



**Avaya™ Computer Telephony 1.2**

**Telephony Services Application  
Programming Interface (TSAPI)**

**Programmer's Reference**

Issue 1  
December 2002

**Copyright © 2002, Avaya, Inc.  
All Rights Reserved  
Printed in U.S.A.**

### **Notice**

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

#### **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 working on your company's behalf). Be aware that there may 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 Web site:

<http://www.avaya.com>

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

#### **Providing Telecommunications Security**

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

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

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

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

### **Your Responsibility for Your Company's Telecommunications Security**

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

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

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

q your Avaya-provided telecommunications systems and their interfaces

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

### **Trademarks**

Adobe, Adobe Acrobat, and the Adobe logo are registered trademarks of Adobe Systems, Inc.

Avaya and MultiVantage are trademarks of Avaya, Inc.

CallVisor, DEFINITY, and the Avaya logotype are registered trademarks of Avaya, Inc.

Internet Explorer is a trademark of SyNet, Inc.

Microsoft, DOS, Windows, Windows NT, Windows 2000, Windows XP, Win32, and the Microsoft logo are registered trademarks and Windows for WorkGroups, Windows 95, and Windows 98 are trademarks of Microsoft.

Netscape Navigator is a registered trademark of Netscape Communications.

Pentium is a registered trademark of Intel Corporation.

Sun, Sun Microsystems and the Sun logo are registered trademarks and Java, Solaris, and Solaris SPARC are trademarks of Sun Microsystems, Inc. in the USA and other countries.

UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company Limited.

UnixWare is a registered trademark of the Santa Cruz Operation, Inc. in the USA and other countries.

All products and company names are trademarks or registered trademarks of their respective holders.

### **Avaya Support**

Avaya provides a telephone number for you to use to report problems or to ask questions about your contact center.

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>

Select **Support**, then select **Escalation Lists US and**

**International**. This Web site includes telephone numbers for escalation within the United States. For escalation telephone numbers outside the United States, select **Global Escalation List**

.

---

# Contents

---

<b>1</b>	<b>Abstract</b>	1-1
	■ About Telephony Services	1-1
	■ About TSAPI	1-2

---

<b>2</b>	<b>Introduction</b>	2-1
	■ Reason for Reissue	2-1
	Impact of Dial Plan Expansion	2-1
	■ Native Network Operating Systems and APIs	2-1
	■ Purpose	2-2
	■ Product Architecture	2-2
	Telephony Services Applications	2-5

---

<b>3</b>	<b>TSAPI Call Model</b>	3-1
	■ Terminology	3-1
	Definitions	3-1
	Acronyms	3-3
	■ Architecture	3-4
	Distribution of Computing and Switching Functions	3-4
	API Services	3-5
	Services and Objects	3-5
	Functions	3-6
	TSAPI Switching Sub-Domain Model	3-6
	TSAPI Device	3-7
	Call	3-10
	TSAPI Connections	3-11
	Call Status Event Reports	3-12
	TSAPI Call States	3-13
	Dynamic Identifier Management	3-15

---

# Contents

---

<b>4</b>	<b>Control Services</b>	4-1
	■ Opening, Closing and Aborting an ACS Stream	4-2
	Opening an ACS Stream	4-2
	Closing an ACS Stream	4-3
	Aborting an ACS Stream	4-3
	■ Sending CSTA Requests and Responses	4-4
	■ Receiving Events	4-4
	Blocking Event Reception	4-5
	Non-Blocking Event Reception	4-5
	■ TSAPI Version Control	4-6
	■ Private Data Version Control	4-8
	■ Querying for Available Services	4-9
	■ API Control Services (ACS)	
	Functions and Confirmation Events	4-9
	acsOpenStream ( )	4-9
	ACSOpenStreamConfEvent	4-15
	acsCloseStream( )	4-17
	ACSCloseStreamConfEvent	4-19
	ACSUniversalFailureConfEvent	4-20
	acsAbortStream( )	4-22
	acsGetEventBlock( )	4-23
	acsGetEventPoll( )	4-25
	acsGetFile( ) (UnixWare)	4-27
	acsSetESR( ) (Windows)	4-28
	acsSetESR( ) (Win32)	4-30
	acsEventNotify( ) (Windows 3.1)	4-32
	acsEventNotify( ) (Win32)	4-35
	acsFlushEventQueue( )	4-37
	acsEnumServerNames( )	4-38
	acsQueryAuthInfo( )	4-39
	■ ACS Unsolicited Events	4-43
	ACSUniversalFailureEvent	4-43
	■ ACS Data Types	4-53
	ACS Common Data Types	4-54
	ACS Event Data Types	4-56

---

## Contents

■ CSTA Control Services and Confirmation Events	4-57
cstaGetAPICaps( )	4-57
CSTAGetAPICapsConfEvent	4-59
cstaGetDeviceList( )	4-61
CSTAGetDeviceListConfEvent	4-63
cstaQueryCallMonitor( )	4-66
CSTAQueryCallMonitorConfEvent	4-67
■ CSTA Event Data Types	4-68

---

<b>5</b>	<b>Switching Function Services</b>	5-1
■	Basic Call Control Services	5-1
	CSTAUniversalFailureConfEvent	5-2
	cstaAlternateCall( )	5-12
	CSTAAlternateCallConfEvent	5-14
	cstaAnswerCall( )	5-16
	CSTAAnswerCallConfEvent	5-17
	cstaCallCompletion( )	5-19
	CSTACallCompletionConfEvent	5-20
	cstaClearCall( )	5-22
	CSTAClearCallConfEvent	5-23
	cstaClearConnection( )	5-26
	CSTAClearConnectionConfEvent	5-28
	cstaConferenceCall( )	5-29
	CSTAConferenceCallConfEvent	5-31
	cstaConsultationCall( )	5-33
	CSTAConsultationCallConfEvent	5-35
	cstaDeflectCall( )	5-37
	CSTADeflectCallConfEvent	5-38
	cstaGroupPickupCall( )	5-40
	CSTAGroupPickupCallConfEvent	5-42
	cstaHoldCall( )	5-43
	CSTAHoldCallConfEvent	5-45
	cstaMakeCall( )	5-47

Issue 1 — December 2002

---

## Contents

CSTAMakeCallConfEvent	5-50
cstaMakePredictiveCall( )	5-51
CSTAMakePredictiveCallConfEvent	5-54
cstaPickupCall( )	5-55
CSTAPickupCallConfEvent	5-57
cstaReconnectCall( )	5-58
CSTARReconnectCallConfEvent	5-61
cstaRetrieveCall( )	5-63
CSTARRetrieveCallConfEvent	5-64
cstaTransferCall( )	5-66
CSTATransferCallConfEvent	5-68
■ Telephony Supplementary Services	5-70
cstaSetMsgWaitingInd( )	5-70
CSTASetMwiConfEvent	5-72
cstaSetDoNotDisturb( )	5-73
CSTASetDndConfEvent	5-75
cstaSetForwarding( )	5-76
CSTASetFwdConfEvent	5-78
cstaSetAgentState( )	5-80
CSTASetAgentStateConfEvent	5-82
cstaQueryMsgWaitingInd( )	5-83
CSTAQueryMwiConfEvent	5-85
cstaQueryDoNotDisturb( )	5-86
CSTAQueryDndConfEvent	5-87
cstaQueryFwd( )	5-89
CSTAQueryFwdConfEvent	5-90
cstaQueryAgentState( )	5-93
CSTAQueryAgentStateConfEvent	5-95
cstaQueryLastNumber( )	5-97
CSTAQueryLastNumberConfEvent	5-98
cstaQueryDeviceInfo( )	5-100
CSTAQueryDeviceInfoConfEvent	5-102

---

# Contents

---

<b>6</b>	<b>Status Reporting Services</b>	6-1
	■ Status Reporting Functions and Confirmation Events	6-1
	cstaMonitorDevice( )	6-3
	cstaMonitorCall( )	6-6
	cstaMonitorCallsViaDevice( )	6-8
	CSTAMonitorConfEvent	6-10
	cstaMonitorStop( )	6-12
	CSTAMonitorStopConfEvent	6-14
	cstaChangeMonitorFilter( )	6-15
	CSTACHangeMonitorFilterConfEvent	6-18
	CSTAMonitorEndedEvent	6-19
	■ Call Event Reports (Unsolicited)	6-21
	cstaCallClearedEvent	6-21
	cstaConferencedEvent	6-24
	CSTAConnectionClearedEvent	6-27
	CSTADeliveredEvent	6-29
	CSTADivertedEvent	6-32
	CSTAEstablishedEvent	6-34
	CSTAFailedEvent	6-36
	CSTAHeldEvent	6-38
	CSTANetworkReachedEvent	6-41
	CSTAOriginatedEvent	6-43
	CSTAQueuedEvent	6-45
	CSTARetrieveEvent	6-48
	CSTAServiceInitiatedEvent	6-50
	CSTATransferredEvent	6-52
	■ Feature Event Reports (Unsolicited)	6-55
	CSTACallInfoEvent	6-55
	CSTADoNotDisturbEvent	6-57
	CSTAForwardingEvent	6-58
	CSTAMessageWaitingEvent	6-61
	■ Agent Status Event Reports (Unsolicited)	6-63
	CSTALoggedOnEvent	6-63
	CSTALoggedOffEvent	6-65

---

# Contents

CSTANotReadyEvent	6-66
CSTARReadyEvent	6-68
CSTAWorkNotReadyEvent	6-69
CSTAWorkReadyEvent	6-71
■ Event Report Data Types (Unsolicited)	6-74
CSTAMonitorFilter_t	6-74
CSTAEventCause_t	6-75

---

<b>7</b>	<b>Snapshot Services</b>	7-1
■	Call Snapshot Services	7-3
	cstaSnapshotCallReq( )	7-4
	CSTASnapshotCallConfEvent	7-5
■	Device Snapshot Service	7-7
	cstaSnapshotDeviceReq( )	7-7
	CSTASnapshotDeviceConfEvent	7-9

---

<b>8</b>	<b>CSTA Computing Function Services</b>	8-1
■	TSAPI Version 2 Routing	8-1
■	Application Call Routing	8-2
■	Routing Registration Functions and Events	8-4
	cstaRouteRegisterReq( )	8-4
	CSTARouteRegisterReqConfEvent	8-6
	cstaRouteRegisterCancel( )	8-8
	CSTARouteRegisterCancelConfEvent	8-10
	CSTARouteRegisterAbortEvent	8-11
■	Routing Functions and Events	8-14
	Register Request ID and the Routing Cross-Reference ID	8-14
	CSTARouteRequestEvent	8-15
	CSTAReRouteRequestEvent	8-18
	cstaRouteSelect( ) TSAPI Version 1 Only	8-20

---

## Contents

cstaRouteSelectInv( )	8-23
CSTARouteUsedEvent	8-25
CSTARouteEndEvent	8-27
cstaRouteEnd( ) TSAPI Version 1 Only	8-29
cstaRouteEndInv( )	8-32

---

<b>9</b>	<b>Escape and Maintenance Services</b>	9-1
■	Escape Services	9-1
■	Maintenance Services	9-3
■	Escape Services : Application as Client	9-5
	cstaEscapeService( )	9-5
	CSTAEscapeServiceConfEvent	9-7
	CSTAPrivateEvent	9-8
	CSTAPrivateStatusEvent	9-9
■	Escape Service : Driver/Switch as the Client	9-11
	CSTAEscapeServiceReq	9-11
	cstaEscapeServiceConf( )	9-12
	cstaSendPrivateEvent( )	9-14
■	Maintenance Services: Device Status	9-16
	CSTABackInServiceEvent	9-17
	CSTAOuOfServiceEvent	9-18
■	System Status - Application as the Client	9-20
	cstaSysStatReq( )	9-20
	CSTASysStatReqConfEvent	9-22
	cstaSysStatStart( )	9-26
	CSTASysStatStartConfEvent	9-28
	cstaSysStatStop( )	9-30
	CSTASysStatStopConfEvent	9-32
	cstaChangeSysStatFilter( )	9-33
	CSTACHangeSysStatFilterConfEvent	9-35
	CSTASysStatEvent	9-37
	CSTASysStatEndedEvent	9-39
■	System Status : Driver/Switch as the Client	9-40

Issue 1 — December 2002

---

# Contents

CSTASysStatReqEvent	9-40
cstaSysStatReqConf( )	9-43
cstaSysStatEvent( )	9-44

---

<b>10</b>	<b>Programming Notes</b>	10-1
	■ Introduction	10-1
	■ TSAPI on Win32 Programming Overview	10-1
	Development Platforms	10-1
	Linking to the TSAPI Library	10-1
	Using Application Control Services	10-2
	Event Notification	10-2
	Receiving Events	10-2
	Blocking Versus Polling	10-2
	Receiving Events From Any Stream	10-2
	Sharing ACS Streams Between Threads	10-3
	Message Trace	10-3
	■ TSAPI on UnixWare	10-4
	Development Platforms	10-4
	Linking to the TSAPI Library	10-4
	Using Application Control Services	10-4
	Event Notification	10-4
	Receiving Events	10-5
	Blocking Versus Polling	10-5
	Receiving Events From Any Stream	10-5
	Message Trace	10-5
	Sample Code	10-6
	■ TSAPI on HP-UX	10-7
	Development Platforms	10-7
	Linking to the TSAPI Library	10-7
	Using Application Control Services	10-7
	Event Notification	10-7
	Receiving Events	10-8
	Blocking Versus Polling**	10-8
	Receiving Events From Any Stream	10-8

---

# Contents

Message Trace	10-8
■ Using High Memory on Windows Clients	10-9

---

<b>11</b>	<b>CSTA Data Types</b>	11-1
■	Introduction	11-1
	Device Identifiers	11-1
■	Basic Call Control Confirmation Events	11-2
	CSTAAlternateCallConfEvent structures	11-2
	CSTAAnswerCallConfEvent structures	11-2
	CSTACallCompletionConfEvent structures	11-2
	CSTAClearCallConfEvent structures	11-2
	CSTAClearConnectionConfEvent structures	11-2
	CSTAConferenceConfEvent structures	11-3
	CSTAConsultationCallConfEvent structures	11-3
	CSTADeflectCallConfEvent structures	11-3
	CSTAGroupPickupCallConfEvent structures	11-3
	CSTAHoldCallConfEvent structures	11-3
	CSTAMakeCallConfEvent structures	11-3
	CSTAMakePredictiveCallConfEvent structures	11-4
	CSTAPickupCallConfEvent structures	11-4
	CSTARetrieveCallConfEvent structures	11-4
	CSTAUniversalFailureEvent structures	11-5
■	Telephony Supplementary Confirmation Events	11-6
	CSTASetMsgWaitingConfEvent structures	11-6
	CSTASetDndConfEvent structures	11-6
	CSTASetFwdConfEvent structures	11-6
	CSTASetAgentStateConfEvent structures	11-7
	CSTAQueryMwiConfEvent structures	11-7
	CSTAQueryDndConfEvent structures	11-7
	CSTAQueryFwdConfEvent structures	11-8
	CSTAQueryAgentStateConfEvent structures	11-8
	CSTAQueryDeviceInfoConfEvent structures	11-9
■	Status Reporting Confirmation Events	11-10

---

# Contents

cstaMonitorDevice structures	11-10
cstaMonitorCall structures	11-11
cstaMonitorCallsViaDevice structures	11-11
CSTAMonitorConfEvent structures	11-11
CSTACHangeMonitorFilterConfEvent structures	11-11
CSTAMonitorStopConfEvent structures	11-11
■ Call Event Reports	11-12
Call Event Report data structures	11-12
CSTACallClearedEvent structures	11-13
CSTAConferencedEvent structures	11-13
CSTAConnectionClearedEvent structures	11-14
CSTADivertedEvent structures	11-14
CSTAEstablishedEvent structures	11-14
CSTAFailedEvent structures	11-15
CSTAHeldEvent structures	11-15
CSTANetworkReachedEvent structures	11-15
CSTAOrginatedEvent structures	11-15
CSTAQueuedEvent structures	11-15
CSTARetrievedEvent structures	11-16
CSTAServiceInitiatedEvent structures	11-16
CSTATransferredEvent structures	11-16
■ Feature Event Reports	11-16
CSTACallInformationEvent structures	11-16
CSTADoNotDisturbEvent structures	11-16
CSTAForwardingEvent structures	11-17
■ Agent Status Report Events	11-18
CSTALoggedOnEvent structures	11-18
CSTALoggedOffEvent structures	11-18
CSTANotReadyEvent structures	11-18
CSTAReadyEvent structures	11-18
CSTAWorkNotReadyEvent structures	11-18
CSTAWorkReadyEvent structures	11-18
■ Snapshot Services	11-19
CSTASnapshotDeviceConfEvent structures	11-19
CSTASnapshotCallConfEvent structures	11-19

---

# Contents

CSTASnapshotDeviceConfEvent structures	11-20
■ Computing Function Services	11-21
cstaRouteRegisterReq structures	11-21
cstaRouteRegisterReqConfEvent structures	11-21
cstaRouteRegisterCancel structures	11-21
cstaRouteRegisterCancelConfEvent structures	11-21
cstaRouteRequestEvent structures	11-22
cstaRouteSelect structures	11-22
CSTAReRouteRequestEvent structures	11-23
cstaRouteUsedEvent structures	11-23
cstaRouteEndEvent structures	11-23
cstaRouteEnd structures	11-23
■ Escape Services	11-24
cstaEscapeService structures	11-24
CSTAEscapeServiceConfEvent structures	11-24
PrivateStatusEvent structures	11-24
cstaPrivateStatusEvent structures	11-24
CSTAEscapeServiceEvent structures	11-24
cstaEscapeServiceConf structures	11-24
cstaSendPrivateEvent structures	11-24
■ Maintenance Services	11-25
CSTABackInServiceEvent structures	11-25
CSTAOutOfServiceEvent structures	11-25
cstaSysStatReq structures	11-25
CSTASysStatReqConfEvent structures	11-25
cstaSysStatStart structures	11-26
CSTASysStatStartConfEvent structures	11-26
cstaSysStatStop structures	11-26
CSTASysStatStopConfEvent structures	11-26
cstaChangeSysStatFilter structures	11-26
CSTACHangeSysStatFilterConfEvent structures	11-26
CSTASysStatReqEvent structures	11-27
cstaSysStatReqConf structures	11-27
cstaSysStatEventSend structures	11-27
■ CSTA Control Services	11-28

---

## Contents

cstaGetAPICaps structures	11-28
CSTAGetAPICapsConfEvent structures	11-28
cstaGetDeviceList structures	11-29
CSTAGetDeviceListConfEvent structures	11-30
CSTA Event Structures	11-30
CSTA Request Event structure	11-30
CSTA Event Report structure	11-30
CSTA Unsolicited Event structure	11-31
CSTA Confirmation Event structure	11-32
CSTA Event_t structure	11-33

---

<b>A</b>	<b>References</b>	1
----------	-------------------	---

---

**About Telephony Services**

---

Telephony Services integrates server-based telephony control with desktop (client) or server applications on enterprise-wide networks. More specifically, Telephony Services logically integrates the existing telephones currently on the user's desktop (analog, ISDN, or those specific to a switch) with telephony-enabled applications running in a client or server. Server and client software create and maintain this logical association; no special telephones, special telephone connectors, PC boards, or other hardware are necessary at a client's desktop. Server hardware terminates the physical control link between the server and the switch (typically a Private Branch Exchange, or PBX) that provides telephony services to the user. The link between Telephony Services and the switch is a *Computer Telephony Integration (CTI)* link.

Software integration gives customers flexibility in deploying CTI applications in environments as varied as the multimedia desktop and call centers. The Telephony Services Application Programming Interface (**TSAPI**) supports telephony applications for many different environments.

Telephony Services and TSAPI support telephony control capabilities in a generic, switch-independent way (e.g., support PBXs from various vendors). The architecture allows the incorporation of vendor-specific switch drivers to deliver Telephony Services across various switch environments.

## About TSAPI

---

The Telephony Services API is based on international standards for CTI telephony services. Specifically, the European Computer Manufacturers Association (**ECMA**) CTI standard definition of *Computer-Supported Telecommunications Applications (CSTA)* is the foundation for TSAPI. The CSTA standard is a technical agreement reached by an open, multi-vendor consortium of major switch and computer vendors. Since CSTA Services and Protocol definitions are the basis for TSAPI, TSAPI provides a generic, switch-independent API.

Various vendors may support a subset of the TSAPI programming interface on their CTI links. Programmers should consult corresponding vendor documentation for application development.

The TSAPI programming interface definition incorporates ECMA CSTA telephony call control services, call/device monitoring services, query services, and services. CSTA services logically integrate the two most common pieces of equipment on users' desktops: the telephone and personal computer.

Security administration for Telephony Services allows administrators to restrict user access to TSAPI features in various ways. For example, an administrator may restrict a user to control and monitoring of the telephone at their desktop. Similarly, an administrator can restrict a user to call control and monitoring of the telephone at any desktop where they log in. Expanded security permissions can increase a user's control in support of work group or departmental telephony applications. Administrators can expand user permissions even further to include any telephone or device that it is possible to control on a CTI link. An administrator might assign an unrestricted security permission level to a server application that processes calls before call delivery to user desktops in a call center environment. Of course, an administrator can assign different users different permissions.

The Telephony Services API is compatible with the following clients: Microsoft Windows, Microsoft Windows 95, Microsoft Windows NT (4.0 and higher), Windows 2000, and UnixWare clients. The Telephony Services API is compatible with Microsoft Windows NT (4.0 and higher) and Windows 2000 server environments. The Telephony architecture permits future growth into other client and server environments while preserving the TSAPI programming interface.

Future releases might extend Telephony Services to include desktop communication services available from non-switch servers. Possible enhancements could provide desktop voice messaging capabilities from a voice messaging server, facsimile capabilities from a fax server, voice response capabilities from a voice response server, etc.

---

## Introduction

# 2

---

### Reason for Reissue

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

### Impact of Dial Plan Expansion

When Avaya Computer Telephony (Avaya CT) cannot determine if a 7-digit extension is internal, it will be reported as an IMPLICIT\_PUBLIC number. Any extension less than 7 digits will be designated as IMPLICIT\_PRIVATE or EXPLICIT\_PRIVATE\_LOCAL\_NUMBER. See “CSTAConferencedEvent structures” on page 11-13.

### Native Network Operating Systems and APIs

The native network operating systems (NOS) provide APIs that allow client and server applications to offer users a variety of computing services (for example, file and print services). Telephony Services expand the set of NOS services and bring together the two most common pieces of equipment on an end user's desktop, the telephone and personal computer. These two environments are integrated using TSAPI, a switch-independent standards-based API.

## **Purpose**

---

This document specifies Telephony Services Application Programming Interface (**TSAPI**) services and C programming language syntax. TSAPI is compatible with Microsoft Windows, Microsoft Windows 95, Microsoft Windows NT, UnixWare, and HP-UX UNIX clients. TSAPI is also compatible with the Microsoft Windows NT server environments. The Telephony architecture permits future growth into other client and server environments while preserving the TSAPI programming interface.

The reader should be familiar with telephony and the ECMA Computer-Supported Telecommunications Application (CSTA) service, and protocol definitions, i.e., ECMA-179 and ECMA-180. The ECMA CSTA standard is the basis for TSAPI service and parameter definition. Write to ECMA at the address below to obtain copies of these standards.

European Computer Manufacturers Association  
114 Rue du Rhône  
CH-1204 Geneva  
Switzerland  
Telephone: 41 22 735 36 34

 **NOTE:**

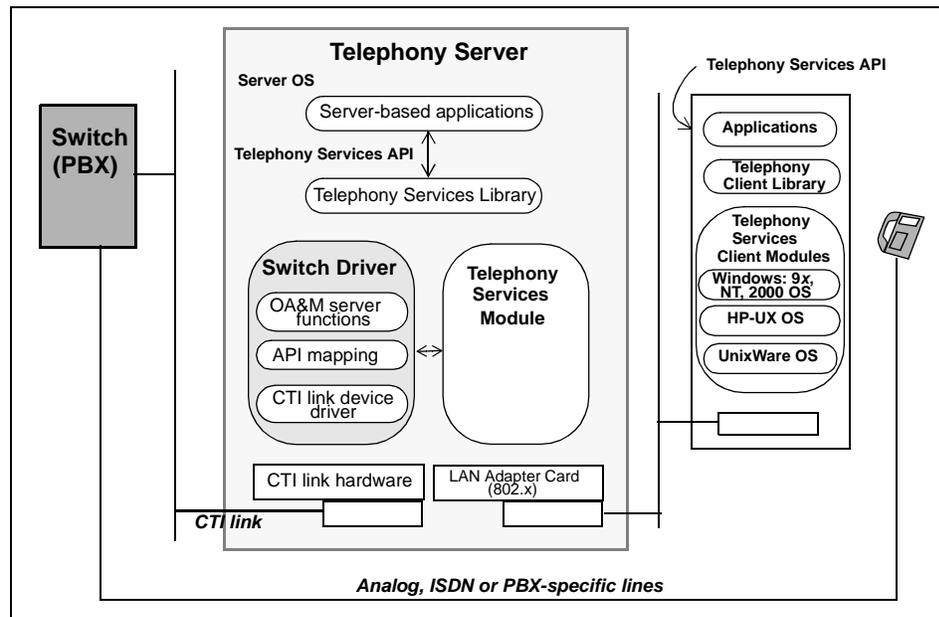
“41” above is the United States International Calling code for Switzerland.

## **Product Architecture**

---

This NOS environment, along with Telephony Services, supports stand-alone or distributed telephony applications on multiple clients/servers. In the case of a distributed application, client and server platforms contain the appropriate application components. For example, the client platform may embody the user interface and local control of the user's telephone. The server platform may use call information to route calls or look up caller information before the switch delivers calls to users' telephones. The NOS supports the client/server communication and interaction between Telephony Services and the application (client or server based).

Since Telephony Services provides a standard API (TSAPI), it facilitates the development and use of telephony-enabled client and server applications. Figure 2-1 illustrates the architecture of Telephony Services. Note that a switch-specific driver terminates the CTI link, thereby making the other Telephony Services modules switch-independent.



**Figure 2-1. Telephony Services Architecture**

Telephony Services consists of the following hardware and software components (refer to Figure 2-1):

- **CTI Link** — a link that supports the Computer Telephony Integration (CTI) protocol that logically integrates the telephone and the client workstation at users' desktops. A CTI link connects link-specific hardware in the server and the switch. It provides sessions between TSAPI applications and the call processing software within the switch. The CTI link is PBX-specific.
- **CTI Link Hardware** — server hardware that terminates the CTI link to a switch. This hardware is PBX-vendor-specific.
- **Switch Driver** — a set of software modules that supports and terminates the switch-specific CTI link and protocol; maps the CTI protocol to the TSAPI (if required); supports any Administration and Maintenance capabilities of the switch driver (if any); and supports a driver interface to the CTI Link Hardware. The Switch Driver modules are PBX-vendor-specific.
- **Switch Driver Interface** — a software interface between the Switch Driver and the Telephony Services module that passes messages between applications and the switch driver. Typically, the messages consist of CSTA service requests, responses, and events for TSAPI clients. The messages may also be administration and maintenance requests, responses, and events for an application doing PBX Driver Administration or Maintenance. The Switch Driver Interface is PBX-independent and supports any Telephony Services compliant driver.

- **Telephony Services module** — a software module that provides communication between multiple telephony-enabled applications and the switch driver. The Telephony Services module routes messages from the Switch Driver to the applications waiting for telephony events and passes the messages received from applications (TSAPI Service Requests) to the Switch Driver. All messages between client applications and a PBX Switch Driver pass across the Switch Driver Interface. The Telephony Services module enforces user restrictions administered in the Telephony Server's security database. This module is switch-independent and supports any Telephony Services-compliant driver.
- **Telephony Server** — an instance of the Telephony Services module software running on a particular NOS server.
- **Telephony Services API (TSAPI)** — the CSTA-based, C language definition of the functions (services), data types (parameters and structures), and event messages that telephony-enabled applications use to access Telephony Services. This document specifies TSAPI. TSAPI is switch-independent and supports any Telephony Services-compliant driver.
- **Telephony Services Library** — server-based applications use this software module to access TSAPI functions. This library accepts TSAPI Service Requests and delivers responses and events to server applications. This library can run on the same physical NOS server as Telephony Services or on any NOS server in the network. This module is switch-independent and supports any Telephony Services-compliant driver.
- **Telephony Client Library** — client-based applications use this software module to access TSAPI functions. This library accepts TSAPI Service Requests and delivers responses and events to client applications. This module is switch-independent and supports any Telephony Services-compliant driver.

In summary, the following software modules and interfaces are switch-independent and support any Telephony Services-compliant driver:

- Switch Driver Interface
- Telephony Services
- Telephony Services API (TSAPI)
- Telephony Services Library
- Telephony Client Library

The following software/hardware modules are switch-specific:

- CTI Link
- CTI Link Hardware, and the
- Switch Driver

## **Telephony Services Applications**

Telephony Services supports a variety of telephony-enabled applications. It can support voice control applications that allow the user to manage and control incoming and outgoing calls at the desktop or more complex applications for the office work group or call center environment. Applications can provide a variety of features to enhance user telephone control from the client workstation.

Application features may include:

- call management
- call screening
- call logging
- directory dialing from personal (client), workgroup (server), and corporate directories
- dialing and integration with other applications, and
- integration of message waiting indicator for email or other messaging applications

Telephony-enabled applications can meet customer needs in many markets. For example, a customer service application can allow agents to interact with both the "telephony" and "caller database" aspects of the job. An easy-to-use graphical user interface (GUI) application can include the caller information on a "screen pop" at agents' PCs. Any server on the network (PC, mini-computer, or mainframe) can contain a caller database. The application integrates access to all the different information (voice and data) needed to support an inbound customer service center. Another example of such an application is call routing. Here, the switch requests that the server provide a destination for incoming calls. Applications may also request outbound calls for Outbound Call Management (OCM) applications such as predictive dialing.



---

This chapter describes concepts from the ECMA CSTA standard that are important for TSAPI application programming. The information presented here is summarized from the CSTA specifications. The complete specification is available from ECMA at the address given in Chapter 2.

### Terminology

The following sections provide TSAPI definitions and acronyms.

#### Definitions

**ACD Agent:** A telephony user that is a member of an inbound or outbound Automatic Call Distribution (ACD) group. ACD Agents first sign on (Login) to an ACD group and then the ACD will distribute calls to the agent.

**Active Call:** The call (at a station) that is connected (in a talking state) at that station. More specifically, the Connection (see *Connection*) for the Active Call is in the Connected State (see *TSAPI Connections* section, *Connection State* definition).

**Alerting Call:** A call that is ringing at a Device. More specifically, the Connection (see *Connection*) for an Alerting Call is in the Alerting State. When the Device is a telephone, the Alerting Call is ringing the telephone instrument.

**Application:** A cooperative process distributed between a Switching Function (see *Switching Function*) and a Computing Function (see *Computing Function*).

**Application Domain:** The union of one Switching Sub-Domain (see *Switching Sub-Domain*) and one Computing Sub-Domain (see *Computing Sub-Domain*).

**Basic Call:** A Call (see *Call*) between exactly two Devices (see *Device*).

**Call (TSAPI programming object):** A Switching Function communications relationship. Typically, a Call is a communications relationship between two or more Devices. Note, however, during call set-up and release, there may be only one Device on the Call. A Call is a TSAPI programming object.

**Call Identifier:** A TSAPI programming handle that identifies a Call.

**Complex Call:** A Call between more than two Devices.

**Computing Domain:** Those computers (and their Objects) accessible from a Switching Function. Where a switch has multiple CTI links to multiple computers, the Computing Domain is the union of all computers connected to the switch.

**Computing Function:** A computer or other resource in a Computing Sub-domain.

**Computing Sub-Domain:** Those computers (and their Objects) accessible from the Switching Function using a specific CTI link. Where a switch has multiple CTI links to multiple computers, the Computing Sub-Domain is a subset of the Computing Domain. Where the switch has a single CTI link, the Computing Sub-Domain is equivalent to the Computing Domain.

**Connection (TSAPI programming object):** A relationship between a Call and a Device. A Connection is in one of a number of states (alerting, held, connected, etc.). Note that when a Call connects (for example) three Devices, there are three Connections for the Call. Each Connection reflects the state of the Call at one of the Devices.

**Connection Identifier (TSAPI programming handle):** A TSAPI programming handle that identifies a Connection. A Call Identifier and a Device Identifier comprise a TSAPI Connection Identifier.

**Device (TSAPI programming object):** An Object that abstracts the interface between a user and the communications signaling in the Switching Function. A Device can be a single endpoint (such as a telephone), or multiple endpoints that form a group (ACD group or trunk group).

**Device Identifier (TSAPI programming handle):** A TSAPI programming handle that identifies a Device.

**Directory Number:** The phone number for a Device. Directory Numbers typically denote telephone station Devices, but ACD groups and other Devices may have Directory Numbers also.

**Domain:** The union of a Switching Domain and a Computing Domain. It is the switching and computing objects that an application can access.

**Event:** A stimulus of interest to an Application that (typically) causes a change in the state of a Device object.

**Event Report:** A message from a Switching Sub-Domain to a Computing Sub-Domain indicating that an Event has occurred.

**Held Call:** A call (at a station) that is held (in a hold state) at that station. More specifically, the Connection (see *Connection*) for a Held Call is in the Hold State (see *TSAPI Connections* section, *Connection State* definition).

**Interconnection Service Boundary:** An abstraction of the boundary between the Switching Domain and the Computing Domain. In practice, CTI links bridge the Interconnection Service Boundary.

**Object:** TSAPI programming objects include Connections, Calls, and Devices. Each has a corresponding programming handle, or identifier.

**Party:** A telephony user. A Party may be a human, application, or other resource (such as a port on a voice response unit).

**Service:** The benefit provided by an Application to a User.

**Service Boundary:** A specific CTI interface between a Computing Function and a Switching Function. All Service boundaries cross the Interconnection Service Boundary.

**State:** An object's current condition. Specifically, TSAPI Connections have an associated state.

**Switching Domain:** Those switches (and their Objects) accessible from a Computing Function. Where a computer has multiple CTI links to multiple switches, the Switching Domain is the union of all switches connected to the computer.

**Switching Function:** A switch in a Switching Sub-domain.

**Switching Sub-Domain:** Those switches (and their Objects) accessible from a Computing Function using a specific CTI link. Where a computer has multiple CTI links to multiple switches, the Switching Sub-Domain is a subset of the Switching Domain. Where the computer has a single CTI link, the Switching Sub-Domain is equivalent to the Switching Domain.

**User:** A person, process or piece of equipment that receives direct benefit (e.g. new feature, improved performance) from an Application's Services.

## **Acronyms**

---

**ACD:** Automatic Call Distributor

**CSTA:** Computer-Supported Telecommunications Applications

**ID:** Identifier

**ISDN:** Integrated Services Digital Network

## **Architecture**

---

This section summarizes the functional architecture underlying ECMA CSTA and TSAPI. CSTA defines the interworking between Computing and Switching Functions in a way that is independent of their physical implementation. This section introduces the concepts of

- distribution of Computing and Switching Functions
- TSAPI Service
- client/server model
- TSAPI objects

An Application is a cooperative process distributed between a Switching Function (switch) and a Computing Function (computer). This section describes the interactions between them.

### **Distribution of Computing and Switching Functions**

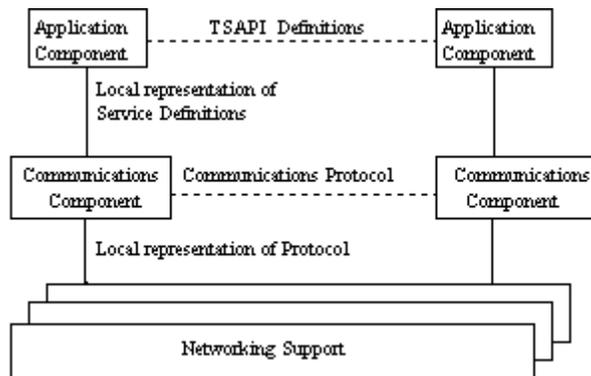
---

One (or several) computers in a computing network provide the Computing Functions and one (or several) switches provide the Switching Functions for a TSAPI Application. The TSAPI application appears to a User (human or machine) to be a single application, not as two separate functions on two separate networks (as it is, in fact, implemented).

Since the applications use distributed resources, communications must occur between the distributed entities. Figure 3-1 shows an abstract communications model. Note that each of the distributed functions is expanded into:

- a client application component that provides the TSAPI interactions
- a server communications component that exchanges messages
- networking support, or lower layer interconnection

■



**Figure 3-1. Model Showing the Relationship Between TSAPI Elements**

Figure 3-1 shows that distributed Application components use TSAPI definitions to interact with their peers. TSAPI defines the Service descriptions and provides the service interface between the Application functions and the Server providing communications with the switch. TSAPI supports various switches, and as a consequence, some of the TSAPI elements are optional and their use is implementation-dependent.

### **API Services**

---

Unless otherwise qualified, the TSAPI definition uses the term 'Service' to refer to the benefit that an application server provides to a client application. TSAPI Services are independent of the specific CTI link connecting the switch with the application server. Since TSAPI is independent of the particular telephone terminal types, the Switching Function must determine how to support a given TSAPI request for its specific telephone types. For example, TSAPI does not specify how to provide the Make Call Service for analog or ISDN telephones. A Switching Function will use its existing service definitions to provide TSAPI Services on telephones where that service already exists.

TSAPI definitions do not embody the specific details of how the Switching Function accomplishes TSAPI Services. The Switching Function does provide an abstraction to the TSAPI Service requester, via Event Reports, of the steps taken to accomplish the Service.

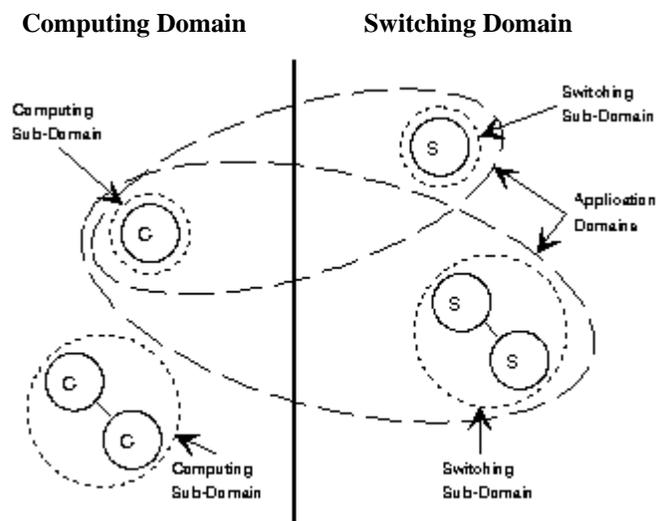
### **Services and Objects**

---

A server provides services to a client that consist of monitoring and controlling Switching Sub-Domain objects. TSAPI defines the client application interface for monitoring and controlling these objects.

## Functions

A Domain is the union of a Switching Domain and a Computing Domain. In other words, it is those switching and computing objects that an application can access. Figure 3-2 shows an example of a Domain. A heavy line divides the Domain into Switching and Computing Domains. The Switching Domain consists of Switching Functions S1, S2, and S3. Similarly, the Computing Domain consists of Computing Functions C1, C2, and C3. Each Function has a view of the Domain where it resides. Note that if multiple Functions provide an application with the same view, then the Functions are in the same Sub-Domain. TSAPI Applications (shown in Figure 3-2 as “Application Domains”) are distributed across at least one Switching Sub-Domain and at least one Computing Sub-Domain.



**Figure 3-2. Domains and Sub-Domains**

## TSAPI Switching Sub-Domain Model

The Switching Sub-Domain Model defines an abstract view of a Switching Function. TSAPI defines several Switching Sub-Domain Model Objects for use in Application programming, Call, and Connection.

## TSAPI Device

A TSAPI application can monitor and control Devices of various types (including telephones). However, a TSAPI application may not be able to monitor or control all Devices. In CSTA, a Device can refer to either a physical device (such as buttons, lines, trunks, and stations) or a logical device (such as groups of devices, pilot numbers, and ACDs). Devices have associated attributes, which allow applications to monitor and control them.

TSAPI Device attributes are:

1. **Device Type** - A Device has one of the following types:

- *ACD* - An Automatic Call Distributor (ACD) is a Switching Function mechanism that distributes calls to ACD agents. An ACD (as opposed to ACD group) consists only of the distribution mechanism and not the ACD agents (or their Devices) to which the mechanism can distribute calls.
- *ACD group* - An Automatic Call Distributor (ACD) group is the mechanism that distributes calls within a Switching Function as well as the ACD agent Devices to which that mechanism distributes calls.
- *Button* - an instance of a call manipulation point at an individual station. Simple analog stations often have no physical buttons but behave as if they have one. Some advanced stations can emulate several analog stations and often represent those stations with several buttons. In some situations it is desirable to identify a given button on a multi-button station. Note that a station with several line appearance buttons could have either the same telephone number or different telephone numbers assigned to those buttons.
- *Button group* - two or more instances of a Button at an individual station.
- *Line* - a communications interface to one or more stations typically associated with a directory number. In some situations it may be impossible to identify individual stations that share a line (a single directory number).
- *Line group* - a set of communications interfaces to one or more stations.
- *Operator* - also known as Attendant, a device used to interact with a party to assist in call setup or to provide other telecommunications service. This device is different from other devices in that it is often involved in setting up other calls, and is usually not part of the call after the call is connected.
- *Operator group* - two or more operator devices used interchangeably or addressed identically.
- *Station* - the traditional telephone device. A station is a physical unit of one or more buttons and one or more lines.

Issue 1 — December 2002

- *Station group* - two or more stations used interchangeably or addressed identically.
  - *Trunk* - a device that spans switching sub-domains. In order to monitor and control calls that cross switching sub-domains, it may be desirable to address the point at which the call crosses the boundary. This point is generally a trunk or trunk group.
  - *Trunk group* - often, many trunks connect to the same place. These trunks are often placed in groups and accessed using a single identifier. In such a configuration the individual trunks are used interchangeably.
2. **Device Class** - An application may monitor or control TSAPI Devices in the various Device Classes in different ways. A Device must belong to one, and may belong to more than one, of the following classes:
- *Data* - a device that is used to make digital data calls (either circuit switched or packet switched). This class includes computer interfaces and G4 facsimile machines.
  - *Image* - a device that is used to make digital data calls involving imaging, or high-speed circuit-switched data in general. This class includes video telephones and CODECs.
  - *Voice* - a device that is used to make audio calls. This class includes all normal telephones, as well as computer modems and G3 facsimile machines.
  - *Other* - a type of device not covered by data, image, or voice.
3. **Device Identifier** - a TSAPI programming handle for a Device that allows an application to identify uniquely each device at the API. Devices are identified using static and/or dynamic identifiers:
- *Static Device Identifier* - A Static Device Identifier is stable over time. It remains constant and unique over calls. A Static Device Identifier is typically the dialed number for the Device known by both the Computing and Switching Functions.

It is sometimes useful for the Switching Function to convert long phone number identifiers to another, usually shorter, static form for subsequent use in service interaction. An example of this would be the transformation of a Public Directory Number to a Private Directory Number.

This transformation allows service interactions to be independent of the identification mechanism and allows reduction in the amount of data exchanged. This transformed number is known as a Short Form Static Device Identifier.

Some Switching Functions allow the same dialed number to be

assigned to Devices of different types. Thus, a TSAPI application may also need to use the Device Type to uniquely address the Device.

- *Dynamic Device Identifier* - A Switching Function may not always make a Static Device Identifier available for every Device on a call. This may occur because a static identifier may not be available (there is no dialed number identifier for the device), or because a dialed number does not unambiguously refer to a single device (i.e. a group identifier). In these cases, the Switching Function assigns a Dynamic Device Identifier as a handle for the Device for the duration of the call. Management of the Dynamic Device Identifier is discussed in a later section, *Dynamic Identifier Management*.
4. **Device State** - a list of the Connection States for all the calls associated with the Device. For information about Connection states see *TSAPI Connections* later in this chapter.

## Call

TSAPI applications can monitor and control calls (including call establishment and release). In certain operations, such as conference and transfer, one Device in a Call is replaced with another Device or two Calls merged into a single Call. In these situations, the TSAPI Call object is maintained as long as the communications relationship remains across each operation (i.e., the call survives transfer, conference, and forwarding operations). TSAPI Call object attributes are:

1. **Call Identifier** - a TSAPI programming handle that the Switching Function assigns to each Call. The Call ID may or may not be unique among all calls within a Switching Sub-Domain, but coupled with a Device ID, the pair will form a unique Connection ID within a Switching Sub-Domain. To allow reference to a nascent call, the switch will assign a Call ID before a call is fully established. For example, a switch will assign a Call ID to an incoming call when the called Device is Alerting (the assignment is done before the call is answered).

Certain Services merge multiple calls into a single call. Examples of such TSAPI Services are Transfer and Conference. During operations of Services that merge multiple calls, the call identifier may change, but the call continues as a TSAPI object. The management of the call identifier is described in a later section, *Dynamic Identifier Management*.

2. **Call state** - a list of the Connection states for all the Devices that are a part of the Call.

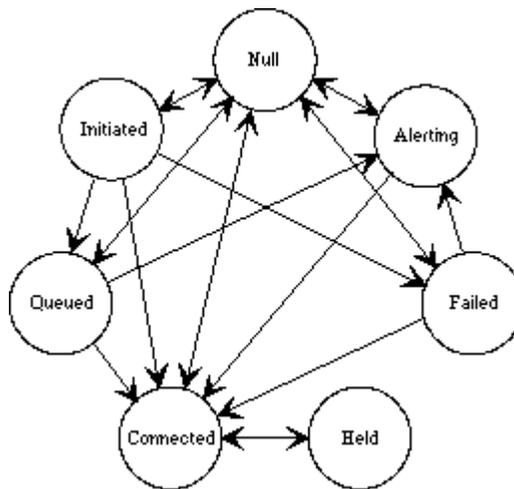
For simplicity, common call states for two-party calls have been given a single descriptive name. For example, a two-party call with a Connection State of "Connected" at one station and a Connection State of "Alerting" at the other has a Call State of "Delivered". Table 3-1 gives the mapping of descriptive names to Connection State lists for two-party calls. Simple call states are provided as single values, whereas uncommon call states are

provided as a list. For more information on Connection States, see the following section, *TSAPI Connections*. The *Call States* section of this chapter gives further information about Call States.

## TSAPI Connections

A Connection is a relationship between a Call and a Device. A TSAPI application can monitor or control a Connection. For example, TSAPI Services Hold Call, Reconnect Call, and Clear Call all control Connections. Connections are TSAPI programming objects with the following attributes:

1. **Connection Identifier** – is a TSAPI handle that is made up of a Call Identifier and Device Identifier. For a call, there are as many Connection identifiers as there are associated devices. Similarly, for a device there are as many Connection identifiers as there are associated calls. The Connection Identifier is unique within a Switching Sub-Domain and within a single TSAPI server. A TSAPI application cannot use a Connection Identifier until it has received the identifier from the Switching Function.
2. **Connection State** - is the state of a call at a Device. The Connection state always refers to a single Call/Device relationship. Snapshot Services report Connection States for Calls and Devices. Monitors report Events, which are changes in Connection States for the monitored entity. Figure 3-3 shows a sample Connection state model. Note that since TSAPI is switch-independent, and since switch features vary from switch to switch (and therefore interact differently on different switches), there is no definitive TSAPI Connection State model to which all switches comply.



**Figure 3-3. Sample Connection State Model**

The transitions between states, shown by arrows, form the basis for providing Event Reports.

The TSAPI Connection states are defined as follows.

- *Null* - the state where there is no relationship between the call and device.
- *Initiated* - the state where the device is requesting service. Usually this results in the creation of a call. Often this is thought of as the “dialing” state.
- *Alerting* - the state where a device is alerting (ringing).
- *Connected* - the state where a device can communicate with other Devices on a call (cannot be a held call).
- *Held* - the call is “on hold” at the Device.
- *Queued* - the state where normal state progression has been stalled. For example, a call being processed by an ACD that is waiting for an ACD agent to become available is “queued.”
- *Failed* - the state where normal state progression has been aborted. A “Failed” state can result because of failure to connect to the calling (originator) device, failure to connect the called (destination) device, failure to create the call, and other reasons.

## Call Status Event Reports

The Switching Sub-Domain model is an abstract view of call states and events. This abstract view is probably more detailed than most applications require, but it introduces a precise language for describing Event Reports, Call States, and Service functional descriptions. Connection state changes correspond to telecommunications signaling at a Device.

ISDN specifications model network access as a distributed state machine. ECMA CSTA borrows from this ISDN model. One part of this access state machine resides in the Device. There is another similar distributed access state machine which resides across the ISDN network at the egress device.

Thus, a call can be modeled as a collection of Connection state machines. Network signaling causes changes in the state machines across the network. When signaling occurs, a state change occurs at the affected Connection.

Many simple Connection events are of interest to applications. Certain telecommunications operations involve changes to many Connections. TSAPI reports these compound events (such as Transfer, Conference and Clear Call) in a single Event Report. Each TSAPI Event Report defines which Connections have changed state.

### **TSAPI Call States**

A Call State is defined as the list of Connection states for all the Devices involved in the call. This list is also called the **Compound Call State**. Listing the Connection states can describe any possible call state. However, most calls are often in a small number of widely recognized states.

TSAPI defines those states as the **Simple Call States** shown in Table 3-1. Note that Simple Call States can differ by the order of the Connection state list. Alerting-Connected is not equal to Connected-Alerting. The first is the Simple Call State “Received” and the second is “Delivered”.

Null can be a known Connection state, so for a “newly created” call it is possible to have a Call state with only one non-Null Connection (see Table 3-1).

For calls with two Connections, Table 3-1 summarizes the Simple Call States assigned to the combinations of Connection States. If there is no entry in Table 3-1 for the combination of Connection states, then TSAPI provides the list.

For calls with more than two non-Null Connection states, the Call State is a compound call state. TSAPI (at times) simplifies the compound call state by relating it to a particular device. The Connection State related to a particular device in this way is called the **Local Connection State**. Other Connection States are not differentiated from one another. A three-party conference call that is on hold at a given Device and connected to the other two devices has a Local Connection State of “Held” at that given Device.

**Table 3-1. TSAPI Simple Call States**

<b>Local Connection State</b>	<b>Other Connection State</b>	<b>Simple Call State</b>
Alerting	Connected	Received
Alerting	Hold	Received-On-Hold
Connected	Alerting	Delivered
Connected	Connected	Established
Connected	Failed	Failed
Connected	Hold	Established-On-Hold
Connected	Null	Originated
Connected	Queued	Queued
Hold	Alerting	Delivered-Held
Hold	Connected	Established-Held
Hold	Failed	Failed-Held
Hold	Queued	Queued-Held
Initiated	Null	Pending
Null	Null	Held

## **Dynamic Identifier Management**

Since Connection Identifiers comprise a Device ID and a Call ID, proper management of Connection Identifiers will, in turn, provide proper management of Dynamic Device Identifiers and Call Identifiers.

The Switching Function provides Connection Identifiers when either a new Call or Device Identifier is needed. When a call is made, the switch provides a Connection Identifier. The switch then provides the Connection ID in any following Event Reports that pertain to that call. Similarly, the switch provides Connection IDs containing a Device ID for a device involved in a call.

The switch updates identifiers when needed. If a Conference or Transfer (merging two calls) changes a Call ID, then the switch provides Event Reports containing Connection IDs that link the old call identifier to the new call identifier. Similarly, if a Dynamic Device Identifier is changed, the switch provides new Connection Identifiers for the devices in the call. Both Service Acknowledgments and Event Reports may contain information necessary to manage identifiers.

Identifiers cease to be valid when their context vanishes. If a call ends, its call identifier is no longer valid. Similarly, if a device is removed from service or from a call, its dynamic device identifier becomes invalid. Many Event Reports and Services specify when a Connection Identifier has lost or will lose its context.

Identifiers can be reused. Once an identifier has lost its context it may be re-used to identify another object. Most implementations will not reuse identifiers immediately.

Call and Device Identifiers can be, but are not guaranteed to be, globally unique. The TSAPI server ensures that the combination of Call and Device Identifier is globally unique within a Switching Sub-Domain. To accomplish this, compliant PBX drivers ensure that either the call identifier or the device identifier (or both) is globally unique. In many cases, the Connection Identifier requires the use of both the Call and Device Identifiers to uniquely refer to Connections in a call.



---

TSAPI provides two kinds of control services: API<sup>1</sup> Control Services, or ACS, and CSTA Control Services. Applications use ACS to manage their interactions with Telephony Services. While most applications will use ACS to access CSTA services, applications that administer PBX drivers use ACS to interface to the PBX Driver. ACS functions manage the interface, while CSTA functions (chapters 5 through 9) provide the CSTA services. Applications use ACS to:

- Open an ACS stream for CSTA services.
- Open an ACS stream to do PBX Driver administration.
- Close an ACS stream.
- Block or poll for events.
- Initialize an operating system event notification facility. On a Windows, Windows NT client, this initializes an Event Service Routine (ESR).
- Get a list of available advertised services (PBX Driver Services and PBX Driver administration services).
- Select the TSAPI version for use on the stream.
- Select a private data version for use on the stream

Applications use the CSTA Control Services, discussed in the later sections of this chapter, to:

- Query for the CSTA Services available on an open ACS Stream
- Query for a list of Devices that CSTA Services can monitor, control or route for on an open ACS Stream
- Query to determine if CSTA Call/Call Monitoring is available on an open ACS Stream.

---

1. An API is an Application Programming Interface.

## Opening, Closing and Aborting an ACS Stream

---

To obtain Telephony Services, 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. Within a Telephony Server, the Telephony Server module and the PBX Driver module cooperate to provide ACS Streams. The Telephony Server also does security checking to ensure that an application receives CSTA services only for permitted Devices. Each application must open an ACS Stream before it requests any services.

An application should only open one stream per advertised service. An application may open multiple ACS streams to multiple advertised services. As PBX drivers initialize, they register the services that they offer (administrative as well as CSTA) with a Telephony Server. The system then advertises these 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. Since the PBX Drivers are switch specific, some drivers may provide services on a single CTI link, while others provide services on multiple CTI links. An application cannot correlate advertised telephony services with underlying physical CTI links.

### Opening an ACS Stream

---

1. The application calls `acsOpenStream()`.

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

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

The application monitors the *acsHandle* (returned from the **`acsOpenStream()`** request) for the corresponding **`ACSOpenStreamConfEvent`**. 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.

#### NOTE:

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.

**⇒ NOTE:**

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

### **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**.

**⇒ NOTE:**

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**

---

1. 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:

- specify 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 window 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 (e.g., Windows NT, Novell, etc.).
- a non-blocking mode that returns control to the application regardless of whether an event is available.

**⇒ NOTE:**

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**

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**

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.

**⇒ NOTE:**

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

In some operating system environments (Windows, Windows NT), an application can use an **Event Service Routine (ESR)** to receive asynchronous notification of arriving events. The ESR mechanism *notifies* the application of arriving events. It does not remove the events from the event queue. The application must use **acsGetEventBlock( )**, **acsGetEventPoll( )**, or **eventNotify( )** 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).

## **TSAPI Version Control**

---

As TSAPI evolves over time to support more services, TSAPI will include new functions and event reports. Similarly, PBX driver manufacturers will enhance their private data over time. To ensure that applications written to earlier versions of the system will continue to operate with newer TSAPI libraries and newer PBX drivers, TSAPI provides Version Control.

An application provides a list of the TSAPI versions that it is willing to accept in the **apiVer** parameter of the **acsOpenStream( )** function. 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.

As the system processes the open stream request, each system component checks to see which of the requested versions it supports. If a component cannot support a requested version, it removes that version from the list before passing the request on to the next component. The system 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 system cannot find a suitable version, the open stream request fails and the application receives an **ACSUniversalFailureConfEvent**.

The system returns the selected TSAPI version in the ***apiVer*** field of the **ACStreamConfEvent**. 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.

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

Releases 2.10 and 2.21 of Telephony Services provide two TSAPI versions: "TS1" and "TS2".

Versions also exist on a stream-type basis. In Releases 2.10 and 2.21, while there are two TSAPI stream versions ("TS1" and "TS2"), there is a single OAM stream version, version "TS1".

## **Private Data Version Control**

---

Private Data versions allow an application to request specific version(s) of a specific vendor(s) private data. The application can also specify that a PBX Driver is not to provide any private data and thereby save the LAN bandwidth that the private data will consume.

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

To request specific version(s) of specific vendor(s) private data, an application passes negotiation information in the private data of **acsOpenStream()**. 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. The string contains vendor/version pairs with the "#" character separating the vendor and version. The "#" character also delimits the vendor/version pairs. The "." and "-" characters are used as they are for TSAPI version negotiation. For example, the string "VENDORA#1-3#VENDORB#1:3-5" indicates that the application requests VENDORA private data versions 1, 2, or 3; or, VENDORB private data versions 1, 3, 4, or 5.

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

An application that does not use Private Data should not pass any private data to the **acsOpenStream()** request. Drivers that do private data version negotiation interpret the lack of private data in the open stream request to mean that the application does not want private data. They will then refrain from sending private data on that stream, thereby saving the LAN bandwidth that the private data would otherwise consume. Applications that do not send any private data in **acsOpenStream()** must be prepared to receive private data since they may request service from a PBX Driver that does not do private data version negotiation.

## **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.

## **API Control Services (ACS) Functions and Confirmation Events**

This section defines the ACS function calls and their confirmation events. Applications use these functions to open ACS streams and to and manage events on ACS Streams between client workstations and the Telephony Server.

### **acsOpenStream()**

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

## 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

### *acsHandle*

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

### invokeIDType

The application sets the type of invoke identifiers used on the stream being opened.

The possible types are: Application-Generated invokeIDs (**APP\_GEN\_ID**) or Library generated invokeIDs (**LIB\_GEN\_ID**).

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

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

### invokeID

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

***streamType***

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

**ST\_CSTA** - requests a CSTA call control stream. This stream can be used for TSAPI service requests and responses which begin with the prefix "csta" or "CSTA".

**ST\_OAM** - requests an OAM stream.

***serverID***

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

***loginID***

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

***passwd***

The application provides a pointer to a null terminated string of maximum size **ACS\_MAX\_PASSWORD**. This string contains the password of the user given *loginID*.

***applicationName***

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

***acsLevelReq***

This version of TSAPI ignores this parameter.

***apiVer***

An application gives the version of TSAPI that it desires in *apiVer*.

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. See the preceding *TSAPI Version Control* section for more information about version negotiation.

Releases 2.10 and 2.21 of Telephony Services support versions “TS1” and “TS2” on CSTA streams and version “TS1” on OAM streams.

Release 1 applications should supply the Version 1 value for **CSTA\_API\_VERSION** in the Version 1 *csta.h* header file in *apiVer*.

#### ***sendQSize***

The application specifies in *sendQSize* the maximum number of outgoing messages the TSAPI Client Library will queue before returning **ACSERR\_QUEUE\_FULL**. If the application supplies a zero (0) value, then a default queue size will be used. The UnixWare TSAPI client library does not use the *sendQSize* parameter.

#### ***sendExtraBufs***

The application specifies the number of additional packet buffers TSAPI allocates for the send queue. If *sendExtraBufs* is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If messages will exceed the size of a network packet, as in the case where private data is used extensively, or the application frequently sees the **ACSERR\_NOBUFFERS** error, then the application does not use *sendExtraBuf* to allocate enough buffers. The UnixWare TSAPI client library does not use the *sendExtraBufs* parameter.

#### ***recvQSize***

The application specifies the maximum number of incoming messages the TSAPI Client Library queues before it ceases acknowledgment to the Telephony Server. TSAPI uses a default queue size when *recvQSize* is set to zero (0). The UnixWare TSAPI client library does not use the *recvQSize* parameter.

#### ***recvExtraBufs***

The application specifies the number of additional packet buffers that TSAPI allocates for the receive queue. If *recvExtraBufs* is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If messages will exceed the size of a network packet, as in the case where private data is used extensively, or the application frequently sees **ACSERR\_STREAM\_FAILED**, then the application does not use *recvExtraBufs* to allocate enough buffers. The UnixWare TSAPI client library does not use the *recvExtraBufs* parameter.

#### ***privateData***

The application may provide a pointer to a data structure that contains any implementation-specific (PBX Driver specific) initialization. TSAPI does not interpret the data in this structure. Some PBX Drivers may use Private Data as an “escape mechanism” to provide implementation specific information between the application and the PBX Driver. An application gives a NULL pointer when Private Data is not present.

In an open stream request, an application may use the private data to negotiate a specific version(s) of a specific vendor(s) private data on the stream. To indicate that the private data is to negotiate the version, the application sets the vendor field in the PrivateData 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. The string contains vendor/version pairs with the "#" character separating the vendor and version. The "#" character also delimits the vendor/version pairs. The ":" and "-" characters are used as they are for TSAPI version negotiation. For example, the string "VENDORA#1-3#VENDORB#1:3-5" indicates that the application requests VENDORA private data versions 1, 2, or 3; or, VENDORB private data versions 1, 3, 4, or 5. No private data on an open stream request is a request to the PBX driver not to send any private data. Only those PBX drivers that support private data version negotiation will honor this request.

See the preceding *Private Data Version Control* section for information on private data version negotiation.

### Return Values

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

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

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

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

**acsOpenStream()** returns the negative error conditions below:

#### **ACSERR\_APIVERDENIED**

The requested API version (*apiVer*) is invalid or the client library does not support it.

#### **ACSERR\_BADPARAMETER**

One or more of the parameters is invalid.

#### **ACSERR\_NODRIVER**

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

**ACSERR\_NOSERVER**

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

**ACSERR\_NORESOURCE**

There are insufficient resources to open a ACS Stream.

**Comments**

An application uses **acsOpenStream()** to open a network or local communication channel (ACS Stream) with an advertised service (PBX Driver). 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 (e.g. for a typical PBX Driver this would include **cstaMakeCall**, **cstaTransferCall**, etc.). **acsOpenStream()** returns an *acsHandle* for the stream. The application uses the *acsHandle* to wait for a **ACSOpenStreamConfEvent**. The application uses the **ACSOpenStreamConfEvent** to determine whether the stream opened successfully. The application then uses the *acsHandle* in any further requests that it sends over the stream. An application should only open one stream for any advertised service.

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

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

**⇒ NOTE:**

The UnixWare TSAPI client library does not use the **sendQsize**, **sendExtraBufs**, **recvQsize**, or **recvExtraBufs** parameters.

**Application Notes**

A Telephony Server advertises services for each registered PBX Driver. A PBX Driver may support a single CTI link or multiple CTI links. Each advertised service name is unique on the network.

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

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

The application must be prepared to receive an **ACSUniversalFailureConfEvent** (for any stream type), **CSTAUniversalFailureConfEvent** (for a CSTA stream type) or an

**ACSUniversalFailureEvent** (for any stream type) anytime after the **acsOpenStream()** function completes. These events indicate that a failure has occurred on the stream.

## **ACSOpenStreamConfEvent**

This event is generated in response to the **acsOpenStream()** function and provides the application with status information regarding the requested open of an ACS Stream with the Telephony Server. 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. See section **4.3, ACS Data Types** and **4.6, CSTA Data Types** for a more 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
            {
                ACSOpenStreamConfEvent_t    acsopen;
            } u;
        } acsConfirmation;
    } event;
} CSTAEvent_t;

typedef struct ACSOpenStreamConfEvent_t
{
    Version_tapiVer;
    Version_tlibVer;
    Version_ttsrvVer;
    Version_tdrvrvVer;
} ACSOpenStreamConfEvent_t;
```

### **Parameters**

***acsHandle***

Issue 1 — December 2002

This is the handle for the ACS Stream.

***eventClass***

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

***eventType***

This is a tag with the value **ACS\_OPEN\_STREAM\_CONF**, which identifies this message as an ACSOpenStreamConfEvent.

***invokeID***

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

***apiVer***

This parameter indicates which version of the API was granted. The version begins with the letters “ST” (the “S” and the “T” are intentionally reversed. Note that the application supplied string had the letters in the order “TS”) followed by a single TSAPI version number. If the contents of the ***apiVer*** field do not begin with the letters “ST”, then the application should assume TSAPI version 1. See the preceding *Private Data Version Control* section for information on private data version negotiation.

***libVer***

This parameter indicates which version of the Library is running.

***tsrvVer***

This parameter indicates which version of the TSERVER is running.

***drvVer***

This parameter indicates which version of the Driver is running.

**Comments**

This message is an indication that the ACS Stream requested by the application via the **acsOpenStream()** function is available to provide communication with the Telephony Server. The application may now request call control services from the Telephony Server on the **acsHandle** identifying this ACS Stream. This message contains the Level of the stream opened, the identification of the server that is providing service, and any Private data returned by the Telephony Server.

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

**Application Notes**

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

## **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_tacsHandle, /* INPUT */
    InvokeID_tinvokeID, /* INPUT */
    PrivateData_t*privateData); /* INPUT */
```

### **Parameters**

#### ***acsHandle***

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

#### ***invokeID***

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

#### ***privateData***

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

## Return Values

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

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

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

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

**acsCloseStream( )** returns the negative error conditions below:

### **ACSERR\_BADHDL**

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

## Comments

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

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

## Application Notes

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

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

The **acsGetEventBlock( )** and **acsGetEventPoll( )** functions can only 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 section **4.2 ACS Data Types** and **4.6 CSTA Data Types** for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            InvokeID_tinvokeID;
            union
            {
                ACSCloseStreamConfEvent_t    acsclose;
            } u;
        } acsConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### **acsHandle**

This is the handle for the opened ACS Stream.

#### **eventClass**

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

***eventType***

This is a tag with the value **ACS\_CLOSE\_STREAM\_CONF**, which identifies this message as an **ACSCloseStreamConfEvent**.

***invokeID***

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

**Comments**

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

**ACSUniversalFailureConfEvent**

This event can occur at any time in place of a confirmation event for any of the CSTA functions which have their own confirmation event and indicates a problem in the processes of the requested function. It does not indicate a failure or loss of the ACS Stream with the Telephony Server. If the ACS Stream has failed, then an ACSUniversalFailureEvent (unsolicited version of this confirmation event) is sent to the application.

**Syntax**

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

```

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

t
typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            union
            {
                ACSUniversalFailureConfEvent_t    failureEvent;
            } u;
        } acsConfirmation;
    } event;
} CSTAEvent_t;

typedef struct
{
    ACSUniversalFailure_terror;
} 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 indicate the cause value for the failure of the original Telephony request. These cause values are the same set as those shown for **ACSUniversalFailureEvent**.

### Comments

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

### acsAbortStream()

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

### Syntax

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

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

### Parameters

#### ***acsHandle***

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

#### ***privateData***

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

### Return Values

This function always returns zero (0) if successful.

The following are possible negative error conditions for this function:

### **ACSERR\_BADHDL**

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

### **Comments**

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

### **acsGetEventBlock()**

---

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

### **Syntax**

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

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

### **Parameters**

#### **acsHandle**

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

#### **eventBuf**

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

#### **eventBufSize**

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

***privateData***

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

***numEvents***

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

On a UnixWare client, *numEvents* will have a value of 0 or 1 indicating that the event queue is empty or non-empty (respectively); this value on a UnixWare client does not indicate the number of events in the queue. On a UnixWare client when *acsHandle* is 0, the *numEvents* value refers only to the stream for which the event was retrieved (and is not an aggregate count).

**Return Values**

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

***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

***ACSERR\_BADHDL***

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

***ACSERR\_UBUFSMALL***

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

### Comments

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

There is no confirmation event for this function.

### Application Notes

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

## 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 than the *eventBuf*, then this parameter will be returned with the size of the buffer required to receive the event. The application should call this function again with a larger buffer.

***privateData***

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

***numEvents***

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

**Return Values**

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

***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

***ACSERR\_BADHDL***

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

***ACSERR\_NOMESSAGE***

There were no messages available to return to the application.

***ACSERR\_UBUFSMALL***

The user buffer size indicated in the *eventBufSize* parameter was smaller than the size of the next available event for the application on the ACS stream. The *eventBufSize* variable has been reset by

the API Library to the size of the next message on the ACS stream. The application should call **acsGetEventPoll()** again with a larger buffer. The ACS event is still on the API Library queue.

### Comments

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

If no events existed on the ACS Stream for the application, this function will return **ACSERR\_NOMESSAGE**.

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

There is no confirmation event for this function.

### Application Notes

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

## acsGetFile() (UnixWare)

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

### Syntax

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

RetCode_t acsGetFile (ACSHandle_t acsHandle);
```

### Parameters

#### **acsHandle**

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

### Return Values

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

### Application Notes

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

There is no confirmation event for this function.

### acsSetESR() (Windows)

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

### Syntax (Windows)

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

#ifdef void (*EsrFunc)(unsigned short esrParam)

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

### Parameters

#### **acsHandle**

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

#### **esr**

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

### ***esrParam***

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

### ***notifyAll***

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

### **Return Values**

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

#### ***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

#### ***ACSERR\_BADHDL***

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

### **Comments**

The ESR mechanism can be used by the application to receive an asynchronous notification of the arrival of an incoming event from the ACS Stream. The ESR routine will receive one user-defined parameter. Windows calls the ESR function with interrupts disabled (not normal application context). The ESR should **not** call ACS functions, TSAPI functions, and many of the Windows APIs 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.

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

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

There is no confirmation event for this function.

### **Application Notes**

The application can use the ESR mechanism to trigger platform specific events (e.g. post a Windows message for the application).

The application may use the ESR mechanism for asynchronous notification of the arrival of incoming events, but most API Library environments provide other mechanisms for receiving asynchronous notification.

The application should not call ACS functions from within the ESR.

The application should complete its ESR processing as quickly as possible.

The ESR function *may* be called while (some level of) interrupts are disabled. This is API implementation specific, so the application programmer should consult the API documentation. Under Windows, the ESR function must be exported and its address obtained from **MakeProcInstance()**.

#### **Windows Client Note:**

Use **acsSetESR()** with care. ESR code and data must be immune to swapping (i.e., fixed and page locked). The ESR must reside in a DLL so as to be fixed. Interrupts are disabled when an ESR is called. Within the ESR, do not call any function that may enable interrupts (including most Windows APIs) or which is not "nailed down".

### **acsSetESR() (Win32)**

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

#### **Syntax**

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

typedef void (*EsrFunc)(unsigned long esrParam)

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

#### **Parameters**

##### ***acsHandle***

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

##### ***esr***

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

##### ***esrParam***

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

***notifyAll***

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

**Return Values**

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

***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

***ACSERR\_BADHDL***

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

**Comments**

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

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

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

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

There is no confirmation event for this function.

## **acsEventNotify() (Windows 3.1)**

---

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

### **Parameters**

#### ***acsHandle***

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

#### ***hwnd***

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

#### ***msg***

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

```
wParamacsHandle
HIWORD(lParam)eventClass
LOWORD(lParam)eventType
```

#### ***notifyAll***

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

### **Return Values**

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

### ***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

### ***ACSERR\_BADHDL***

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

### **Application Notes**

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

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

Applications which process a high volume of incoming events may cause the default application queue (8 messages max) to overflow. In this case, use the Windows API call ***SetMessageQueue()*** to increase the size of the application queue. Also, the rate of notifications may be reduced by setting ***notifyAll*** to **FALSE**.

There is no confirmation event for this function.

### Example

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

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

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

    switch (msg)
    {
        case WM_CREATE:

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

            acsEventNotify (acsHandle, hwnd, WM_ACSEVENT, TRUE);

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

        case WM_ACSEVENT:

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

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

            return 0;

        // process other window messages...
    }
    return DefWindowProc (hwnd, msg, wParam, lParam);
}
```

## **acsEventNotify() (Win32)**

---

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

### **Syntax**

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

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

### **Parameters**

#### ***acsHandle***

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

#### ***hwnd***

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

#### ***msg***

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

<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 (see below).

### **Return Values**

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

#### ***ACSPOSITIVE\_ACK***

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

Issue 1 — December 2002

Possible local error returns are (negative returns):

### ***ACSERR\_BADHDL***

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

## **Application Notes**

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

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

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

There is no confirmation event for this function.

## **Example**

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

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

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

    switch (msg)
    {
    case WM_CREATE:

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

        acsEventNotify (acsHandle, hwnd, WM_ACSEVENT, TRUE);

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

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

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

        return 0;
    }
```

```
        // process other window messages...  
  
    }  
    return DefWindowProc (hwnd, msg, wParam, lParam);
```

## **acsFlushEventQueue()**

---

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

### **Syntax**

```
#include <acs.h>  
#include <csta.h>  
  
RetCode_t ACSFlushEventQueue (ACSHandle_t acsHandle);
```

### **Parameters**

#### ***acsHandle***

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

### **Return Values**

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

#### ***ACSPOSITIVE\_ACK***

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

Possible local error returns are (negative returns):

#### ***ACSERR\_BADHDL***

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

### **Comments**

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

There is no confirmation event for this function.

### **Application Notes**

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

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

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

## **acsEnumServerNames( )**

---

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

### **Syntax**

```
#include <acs.h>

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

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

### **Parameters**

#### ***streamType***

indicates the type of stream requested. The currently defined stream types are **ST\_CSTA** and **ST\_OAM**.

#### ***callback***

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

#### ***IParam***

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

### **Return Values**

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

**ACSPOSITIVE\_ACK**

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

The following are possible negative error conditions for this function:

**ACSERR\_UNKNOWN**

The request has failed due to unknown network problems.

**ACSERR\_NOSERVER**

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

**Comments**

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

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

**acsQueryAuthInfo()**

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

**Syntax**

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

**Parameters****serverID**

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

**authInfo**

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

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

typedef struct
{
    ACSAuthType_t    authType;
    LoginID_t        authLoginID;
} ACSAuthInfo_t;
```

### Return Values

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

***ACSERR\_BADPARAMETER***

One or more of the parameters is invalid.

***ACSERR\_NODRIVER***

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

***ACSERR\_NOSERVER***

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

***ACSERR\_NORESOURCE***

There are insufficient resources to query the advertised service.

### Background

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

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

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

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

## Usage

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

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

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

### REQUIRES\_EXTERNAL\_AUTH:

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

### AUTH\_LOGIN\_ID\_ONLY:

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

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

### AUTH\_LOGIN\_ID\_IS\_DEFAULT:

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

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

## ⇒ NOTE:

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

### NEED\_LOGIN\_ID\_AND\_PASSWD:

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

### ANY\_LOGIN\_ID:

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

## **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 Telephony Server.

#### **Syntax**

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

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

typedef struct
{
    ACSEventHeader_teventHeader;
    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 Tserver operation error (or “cause value”), Tserver 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 logs generated by the Tserver.

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,
```

Issue 1 — December 2002

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,  
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;
```

### Tserver Operation errors

Tserver operation errors indicate that there is an error in the Service Request. These include the following specific error values:

#### ***TSERVER\_STREAM\_FAILED***

The Client Library detected that the ACS Stream failed.

#### ***TSERVER\_NO\_THREAD***

One or more the threads (processes) that make up the Tserver could not be created.

#### ***TSERVER\_BAD\_DRIVER\_ID***

One of the threads (processes) that make up the Tserver encountered a bad Driver Identification number during processing.

#### ***TSERVER\_DEAD\_DRIVER***

A Driver has not sent a heart beat messages to the Tserver form the last three minutes. The Driver may be in an inoperable state.

***TSERVER\_MESSAGE\_HIGH\_WATER\_MARK***

The message rate between a client and the Tserver or the Tserver and a Driver has exceeded the high water mark rate.

***TSERVER\_FREE\_BUFFER\_FAILED***

The Tserver was unable to free Tserver Driver Interface (TDI) memory.

***TSERVER\_SEND\_TO\_DRIVER***

The Tserver was unable to send a message to a Driver.

***TSERVER\_RECEIVE\_FROM\_DRIVER***

The Tserver was unable to receive a message from a Driver.

***TSERVER\_REGISTRATION\_FAILED***

A Driver's attempt to register with the Tserver failed.

***TSERVER\_TRACE***

Used by the Tserver for debugging purposes only.

***TSERVER\_NO\_MEMORY***

The Tserver was unable to allocate a piece of memory.

***TSERVER\_ENCODE\_FAILED***

The Tserver was unable to encode a message for shipment to a client workstation.

***TSERVER\_DECODE\_FAILED***

The Tserver was unable to decode a message from a client workstation.

***TSERVER\_BAD\_CONNECTION***

The Tserver tried to process a request with a bad client connection ID number.

***TSERVER\_BAD\_PDU***

The Tserver's internal table of Protocol Descriptor Units is corrupted.

***TSERVER\_NO\_VERSION***

The Tserver processed a ACSOpenStreamConfEvent from a Driver in which one or more the version fields was not set.

***TSERVER\_ECB\_MAX\_EXCEEDED***

The Tserver can not process a message from the driver because the message is larger than the sum of the ECBs allocated for this driver.

***TSERVER\_NO\_ECBS***

The Tserver has no available ECBs to send events to the client.

***TSERVER\_NO\_RESOURCE\_TAG***

The Tserver was unable to get a resource tag for the purpose of allocating memory.

***TSERVER\_INVALID\_MESSAGE***

The Tserver received an invalid Tserver OAM message.

***TSERVER\_ECB\_OVERDUE***

The Telephony Server uses this error code in the log file. An application will not receive it.

***TSERVER\_BAD\_PW\_ENCRYPTION***

The Telephony Server uses this error code in the log file. An application will not receive it.

***TSERVER\_BAD\_TRANSPORT\_TYPE***

The Telephony Server uses this error code in the log file. An application will not receive it.

***TSERVER\_BAD\_TSERV\_PROTOCOL***

The application has requested a TSAPI protocol version that the Telephony Server does not provide.

***TSERVER\_BAD\_DRIVER\_PROTOCOL***

The application has requested a TSAPI protocol version that the Telephony Server provides but the PBX Driver does not provide.

***TSERVER\_PDU\_VERSION\_MISMATCH***

The application has invoked a service that the stream version does not support. This resulted in the client library sending the Telephony Server a Protocol Data Unit (PDU) that is not supported in the TSAPI version on that stream.

***TSERVER\_BAD\_PROTOCOL\_FORMAT***

The *apiVer* parameter value is not in the proper syntax.

***TSERVER\_OLD\_TSLIB***

The application is using a TSLIB library of an earlier vintage than the version it is requesting in an open stream call.

***TSERVER\_REQUIRED\_MODULES\_NOT\_LOADED***

***TSERVER\_TRANSPORT\_IN\_USE\_BY\_OAM***

***TSERVER\_NO\_NDS\_OAM\_PERMISSION***

***TSERVER\_OPEN\_SDB\_LOG\_FAILED***

***TSERVER\_INVALID\_LOG\_SIZE,***

***TSERVER\_WRITE\_SDB\_LOG\_FAILED***

***TSERVER\_NT\_FAILURE,***  
***TSERVER\_LOAD\_LIB\_FAILED***  
***TSERVER\_INVALID\_DRIVER***  
***TSERVER\_REGISTRY\_ERROR***  
***TSERVER\_DUPLICATE\_ENTRY***  
***TSERVER\_DRIVER\_LOADED***

#### **Tserver Security Data Base errors**

Error values in this category indicate that there is an error in the process of an event which requires a check against the Security Data Base. This type includes one of the following specific error values:

##### ***TSERVER\_NO\_SDB***

One or more the files that makeup the Security Data Base is not present on the server or can not be opened.

##### ***TSERVER\_NO\_SDB\_CHECK\_NEEDED***

The requested service event does not require a Security Data Base check.

##### ***TSERVER\_SDB\_CHECK\_NEEDED***

The requested service event does require a Security Data Base check.

##### ***TSERVER\_BAD\_SDB\_LEVEL***

The Tserver's internal table of API calls indicating which level of security to perform on the request is corrupted.

##### ***TSERVER\_BAD\_SERVERID***

The Tserver rejected an ACSOpenStream request because the Server ID in the message did not match a Driver supported by this Tserver.

##### ***TSERVER\_BAD\_STREAM\_TYPE***

The stream type an ACSOpenStream request was invalid.

##### ***TSERVER\_BAD\_PASSWORD\_OR\_LOGIN***

The Password or Login or both from an ACSOpenStream request did not match an entry in the Bindery on the server the Tserver is running on.

##### ***TSERVER\_NO\_USER\_RECORD***

No user record was found in the Security Data Base for the login specified in the ACSOpenStream request.

##### ***TSERVER\_NO\_DEVICE\_RECORD***

No device record was found in the Security Data Base for the device specified in the API call.

***TSERVER\_DEVICE\_NOT\_ON\_LIST***

The specified device in an API call was not found on any device list administered for this user.

***TSERVER\_USERS\_RESTRICTED\_HOME***

The Tserver is administered to restrict users to home worktops so no checking is done against away worktop devices.

***TSERVER\_NO\_AWAYPERMISSION***

The Tserver rejected a service request because the device did not match a device associated with an away worktop.

***TSERVER\_NO\_HOMEPERMISSION***

The Tserver rejected a service request because the device did not match a device associated with a home worktop.

***TSERVER\_NO\_AWAY\_WORKTOP***

The away worktop the user is working from is not administered in the Security Data Base.

***TSERVER\_BAD\_DEVICE\_RECORD***

The Tserver read a device record from the Security Data Base that contained corrupted information.

***TSERVER\_DEVICE\_NOT\_SUPPORTED***

The device in the API call is administered to be supported by a different Tserver.

***TSERVER\_INSUFFICIENT\_PERMISSION***

The device in the API call is at the users away worktop and the device has a higher permission level than the user, preventing the user from controlling the device.

***TSERVER\_EXCEPTION\_LIST***

The device in the API call is on an exception list which is administered as part of the information for this user.

**Driver Errors**

Error values in this category indicate that the driver detected an error. This type includes one of the following specific error values:

***DRIVER\_DUPLICATE\_ACSHANDLE***

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

***DRIVER\_INVALID\_ACS\_REQUEST***

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

***DRIVER\_ACS\_HANDLE\_REJECTION***

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

***DRIVER\_INVALID\_CLASS\_REJECTION***

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

***DRIVER\_GENERIC\_REJECTION***

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

***DRIVER\_RESOURCE\_LIMITATION***

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

***DRIVER\_ACSHANDLE\_TERMINATION***

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

***DRIVER\_LINK\_UNAVAILABLE***

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

***DRIVER\_OAM\_IN\_USE***

There is already an open OAM-type stream to the PBX driver. A driver can only have a single OAM stream open at any time.

## **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.

## ACS Common Data Types

---

This section specifies the common ACS data types.

```
typedef int RetCode_t;

#define ACSPOSITIVE_ACK 0 /* Successful function return */

/* Error Codes */

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

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

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

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

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

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

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

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

```
#define ACSERR_UNKNOWN-9 /* The ACS Stream has
    * 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.
    */

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 ACSREQUEST0
#define ACSUNSOLICITED1
#define ACSCONFIRMATION2

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

typedef char Boolean;
typedef char Nulltype;
```

```
#define          ACS_OPEN_STREAM1
#define          ACS_OPEN_STREAM_CONF2
#define          ACS_CLOSE_STREAM3
#define          ACS_CLOSE_STREAM_CONF4
#define          ACS_ABORT_STREAM5
#define          ACS_UNIVERSAL_FAILURE_CONF6
#define          ACS_UNIVERSAL_FAILURE7

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 section 4.6).

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

typedef struct
{
    union
    {
        ACSUniversalFailureEvent_t failureEvent;
    } u;
} ACSUnsolicitedEvent;

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

## **CSTA Control Services and Confirmation Events**

---

This section defines the CSTA functions associated with the Telephony Server's Services. These functions are used to determine types and capabilities of Telephony Servers and Drivers connected to Telephony Servers and to determine the set of devices an application can control, monitor and query.

### **cstaGetAPICaps()**

cstaGetAPICaps() obtains the CSTA API function and event capabilities which are supported on an open CSTA stream. The stream could be a local PBX driver or a remote PBX Driver on a network. If a stream provides a CSTA service then it also provides the corresponding CSTA confirmation event.

#### **Syntax**

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

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

## Parameters

### ***acsHandle***

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

### ***invokeID***

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

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

### ***ACSERR\_BADHDL***

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

## Comments

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

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

## CSTAGetAPICapsConfEvent

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

### Syntax

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

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        {
            struct
            {
                InvokeID_tinvokeID;
                union
                {
                    {
                        CSTAGetAPICapsConfEvent_t getAPICaps;
                    } u;
                } cstaConfirmation;
            } event;
        } CSTAEvent_t;

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

```
short      reconnectCall;
short      retrieveCall;
short      setMwi;
short      setDnd;
short      setFwd;
short      setAgentState;
short      transferCall;
short      eventReport;
short      callClearedEvent;
short      conferencedEvent;
short      connectionClearedEvent;
short      deliveredEvent;
short      divertedEvent;
short      establishedEvent;
short      failedEvent;
short      heldEvent;
short      networkReachedEvent;
short      originatedEvent;
short      queuedEvent;
short      retrievedEvent;
short      serviceInitiatedEvent;
short      transferredEvent;
short      callInformationEvent;
short      doNotDisturbEvent;
short      forwardingEvent;
short      messageWaitingEvent;
short      loggedOnEvent;
short      loggedOffEvent;
short      notReadyEvent;
short      readyEvent;
short      workNotReadyEvent;
short      workReadyEvent;
short      backInServiceEvent;
short      outOfServiceEvent;
short      privateEvent;
short      routeRequestEvent;
short      reRoute;
short      routeSelect;
short      routeUsedEvent;
short      routeEndEvent;
short      monitorDevice;
short      monitorCall;
short      monitorCallsViaDevice;
short      changeMonitorFilter;
short      monitorStop;
short      monitorEnded;
short      snapshotDeviceReq;
short      snapshotCallReq;
short      escapeService;
short      privateStatusEvent;
short      escapeServiceEvent;
short      escapeServiceConf;
short      sendPrivateEvent;
short      sysStatReq;
```

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

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_GETAPI\_CAPS\_CONF**, which identifies this message as an **CSTAGetAPICapsConfEvent**.

#### ***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 Telephony Server on a command/event basis. All the commands and events supported by a Telephony Server must be supported as defined in this document.

## **cstaGetDeviceList()**

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

### Syntax

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

RetCode_t      cstaGetDeviceList(
    ACSHandle_t      acsHandle,
    InvokeID_t      invokeID,
```

```

long          index,
CSTALevel_t  level)

```

## Parameters

### *acsHandle*

This is the handle to an active ACS Stream.

### *invokeID*

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

### *index*

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

### *level*

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

```

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

```

## ⇒ NOTE:

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

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

#### **ACSERR\_BADHDL**

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

### **CSTAGetDeviceListConfEvent**

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

#### **Syntax**

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

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            InvokeID_tinvokeID;
            union
            {
                CSTAGetDeviceListConfEvent_t getDeviceList;
            } u;
        }
    }
}
```

Issue 1 — December 2002

```
        } cstaConfirmation;  
    } event;  
} CSTAEvent_t;
```

```
typedef enum SDBLevel_t {  
  
    NO_SDB_CHECKING = -1,  
    ACS_ONLY = 1,  
    ACS_AND_CSTA_CHECKING = 0  
} SDBLevel_t;  
  
typedef struct CSTAGetDeviceList_t {  
    long          index;  
    CSTALevel_t   level;  
} CSTAGetDeviceList_t;  
  
typedef struct DeviceList {  
    short         count;  
    DeviceID_t    device[20];  
} DeviceList;  
  
typedef struct CSTAGetDeviceListConfEvent_t {  
    SDBLevel_t    driverSdbLevel;  
    CSTALevel_t   level;  
    long          index;  
    DeviceList    devList;  
} CSTAGetDeviceListConfEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

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

### ***eventType***

This is a tag with the value **CSTA\_GET\_DEVICE\_LIST\_CONF**, which identifies this message as an **CSTAGetDeviceListConfEvent**.

### ***invokeID***

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

### ***driverSdbLevel***

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

NO\_SDB\_CHECKING Not Used.  
ACS\_ONLY Check ACSOpenStream requests only  
ACS\_AND\_CSTA\_CHECKING Check ACSOpenStream and all applicable CSTA messages

If the SDB database is disabled by administration, and the driver registered with SDB level ACS\_AND\_CSTA\_CHECKING, the Tserver 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 Tserver is using for returning the list of devices. The client application should return this value in the next call to CSTAGetDeviceList to continue receiving devices. A value of (-1) indicates there are no more devices in the list.

***devlist***

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

## **cstaQueryCallMonitor()**

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

**Syntax**

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

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

**Parameters*****acsHandle***

This is the handle to an active ACS Stream.

***invokeID***

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

**Return Values**

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

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

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

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

The following are possible negative error conditions for this function:

#### **ACSERR\_BADHDL**

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

## **CSTAQueryCallMonitorConfEvent**

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

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See the **ACS Data Types** and **CSTA Data Types** sections 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
            {
                CSTAQueryCallMonitorConfEvent_t queryCallMonitor;
            } u;
        } cstaConfirmation;
    }
}
```

```
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_QUERY\_CALL\_MONITOR\_CONF**, which identifies this message as an **CSTAQueryCallMonitorConfEvent**.

#### ***invokeID***

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

#### ***callMonitor***

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

## **CSTA Event Data Types**

---

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

An application always receives a generic *CSTAEvent\_t* event structure. This structure contains an *ACSEventHeader\_t* structure which contains information common to all events. This common information includes:

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

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

```

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

#include <acs.h>
// defines for CSTA event classes

#define CSTAREQUEST3
#define CSTAUNSOLICITED4
#define CSTACONFIRMATION5
#define CSTAEVENTREPORT6

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

typedef struct {
    union
    {
        CSTARouteRegisterAbortEvent_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       ransferCall;
        CSTAUniversalFailureConfEvent_t   universalFailure;
        CSTAMonitorConfEvent_t            monitorStart;
        CSTAChangeMonitorFilterConfEvent_t changeMonitorFilter;
        CSTAMonitorStopConfEvent_t        monitorStop;
        CSTASnapshotDeviceConfEvent_t     snapshotDevice;
        CSTASnapshotCallConfEvent_t       snapshotCall;
        CSTARouteRegisterReqConfEvent_t   routeRegister;
    }
}

```

```
        CSTARouteRegisterCancelConfEvent_trouteCancel;
        CSTAEscapeSvcConfEvent_t          escapeService;
        CSTASysStatReqConfEvent_t         sysStatReq;
        CSTASysStatStartConfEvent_t       sysStatStart;
        CSTASysStatStopConfEvent_t        sysStatStop;
        CSTAChangeSysStatFilterConfEvent_tchangeSysStatFilter;
        CSTAGetAPICapsConfEvent_t         getAPICaps;
        CSTAGetDeviceListConfEvent_t      getDeviceList;
        CSTAQueryCallMonitorConfEvent_t   queryCallMonitor;
    } u;
} CSTAConfirmationEvent;

#define CSTA_MAX_HEAP 1024

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

---

This section describes Telephony Services. Applications use Telephony Services to control calls and activate switch features. Switching Functions Services are divided into **Basic Call Control Services** and **Telephony Supplementary Services**.

### **Basic Call Control Services**

---

Basic Call Control Services allows applications to:

- establish, control, and "tear-down" calls at a device or within the switch,
- answer incoming calls at a device, and
- activate/de-activate switch features.

Each Basic Call Control Service request has an associated confirmation event message. The confirmation message returns the status and other service-specific information to the application. TSAPI always returns confirmation event messages for successful function calls. If TSAPI cannot successfully process a function call then

- TSAPI does not send the service request to the PBX Driver
- TSAPI does not generate a confirmation event

As noted in Chapter 4, section **Sending CSTA Requests and Responses**, the application sets the `invokeID` type (when it opens the stream) to either library generated or application generated. As described in that section, applications may use application generated `invokeIDs` to index into data structures in various ways. The application may also use the `invokeID` to match results with specific service requests.

When TSAPI successfully processes an application request, TSAPI sends the application a confirmation event. This confirmation means that TSAPI has successfully processed the request, not that the PBX driver or PBX has successfully processed the request. For example, TSAPI will send an application a **CSTAMakeCallConfEvent** after TSAPI (not the PBX) successfully processes a

**cstaMakeCall()** request. Further information from the PBX Driver or PBX will arrive in call events or unsolicited status events. An application interested in the results of a request should check for a function confirmation event and any applicable unsolicited status events (see **Status Reporting Services**).

To receive events, an application must have an active ACS Stream and an implement an event handling mechanism. Further, the reception of unsolicited events requires an active monitor. See the **Control Services** and **Status Reporting Services** sections for more information on events.

**⇒ NOTE:**

Not every Driver implementation will support all Telephony functions. The application should use the `cstaGetAPICaps` function to determine which Telephony Services are supported.

## **CSTAUniversalFailureConfEvent**

---

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

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Chapter 4 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
            {
                CSTAUniversalFailureConfEvent universalFailure;
            } u;
        } cstaConfirmation;
    } event;
}
```

```
    } CSTAEvent_t;

typedef struct
{
    UniversalFailure_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,
    UNSPECIFIED_SECURITY_ERROR = 60,
    SEQUENCE_NUMBER_VIOLATED = 61,
    TIME_STAMP_VIOLATED = 62,
```

```
PAC_VIOLATED = 63,  
SEAL_VIOLATED = 64  
GENERIC_UNSPECIFIED_REJECTION = 70  
GENERIC_OPERATION_REJECTION = 71  
DUPLICATE_INVOCATION_REJECTION = 72  
UNRECOGNIZED_OPERATION_REJECTION = 73  
MISTYPED_ARGUMENT_REJECTION = 74  
RESOURCE_LIMITATION_REJECTION = 75  
ACS_HANDLE_TERMINATION_REJECTION = 76  
SERVICE_TERMINATION_REJECTION = 77  
REQUEST_TIMEOUT_REJECTION = 78  
REQUESTS_ON_DEVICE_EXCEEDED_REJECTION = 79  
UNRECOGNIZED_APDU_REJECTION = 80,  
MISTYPED_APDU_REJECTION = 81,  
BADLY_STRUCTURED_APDU_REJECTION = 82,  
INITIATOR_RELEASING_REJECTION = 83,  
UNRECOGNIZED_LINKEDID_REJECTION = 84,  
LINKED_RESPONSE_UNEXPECTED_REJECTION = 85,  
UNEXPECTED_CHILD_OPERATION_REJECTION = 86,  
MISTYPED_RESULT_REJECTION = 87,  
UNRECOGNIZED_ERROR_REJECTION = 88,  
UNEXPECTED_ERROR_REJECTION = 89,  
MISTYPED_PARAMETER_REJECTION = 90,  
NON_STANDARD = 100  
} CSTAUniversalFailure_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This tag with a value, **CSTA\_UNIVERSAL\_FAILURE\_CONF**, identifies this message as an CSTAUniversalFailureConfEvent.

#### ***invokeID***

This parameter specifies the function service request instance that has failed at the server or at the switch. This identifier is provided to the application when a service request is made.

#### ***error***

This parameter contains an error value from one of the following classes: Unspecified, Operation, State Incompatibility, System Resource, Subscribed Resource, Performance Management, or Security. The headings the follow contain the specific errors in these classes.

### Unspecified Errors

Error values in this category indicate that an error has occurred that is not among the other error types. This type includes the following specific error values:

***GENERIC\_UNSPECIFIED***

***GENERIC\_UNSPECIFIED\_REJECTION***

### **Operation errors**

Error values in this category indicate that there is an error in the Service Request. This type includes one of the following specific error values:

#### ***GENERIC\_OPERATION***

#### ***GENERIC\_OPERATION\_REJECTION***

This error indicates that the server has detected an error in the operation class, but that it is not one of the defined errors, or the server cannot be any more specific.

#### ***REQUEST\_INCOMPATIBLE\_WITH\_OBJECT***

The request is not compatible with the object.

#### ***DUPLICATE\_INVOCATION***

The invokeID violates X.208 or X.209 assignment rules.

#### ***UNRECOGNIZED\_OPERATION\_REJECTION***

The operation is not defined in TSAPI.

#### ***VALUE\_OUT\_OF\_RANGE***

The parameter has a value that is not in the range defined for the server.

#### ***OBJECT\_NOT\_KNOWN***

The parameter has a value that is not known to the server.

#### ***INVALID\_CALLING\_DEVICE***

The calling device is not valid.

#### ***INVALID\_CALLED\_DEVICE***

The called device is not valid.

#### ***PRIVILEGE\_VIOLATION\_ON\_SPECIFIED\_DEVICE***

The request cannot be provided because the specified device is not authorized for the Service.

#### ***INVALID\_FORWARDING\_DESTINATION***

The request cannot be provided because the forwarding destination device is not valid.

#### ***PRIVILEGE\_VIOLATION\_ON\_CALLED\_DEVICE***

The request cannot be provided because the called device is not authorized for the Service.

#### ***PRIVILEGE\_VIOLATION\_ON\_CALLING\_DEVICE***

The request cannot be provided because the calling device is not authorized for the Service.

#### ***INVALID\_CSTA\_CALL\_IDENTIFIER***

The call identifier is not valid.

***INVALID\_CSTA\_DEVICE\_IDENTIFIER***

The Device Identifier is not valid.

***INVALID\_CSTA\_CONNECTION\_IDENTIFIER***

The Connection identifier is not valid.

***INVALID\_DESTINATION***

The Service Request specified a destination that is not valid.

***INVALID\_FEATURE***

The Service Request specified a feature that is not valid.

***INVALID\_ALLOCATION\_STATE***

The Service Request indicated an allocation condition that is not valid.

***INVALID\_CROSS\_REF\_ID***

The Service Request specified a Cross-Reference Id that is not in use at this time.

***INVALID\_OBJECT\_TYPE***

The Service Request specified an object type that is outside the range of valid object types for the Service.

***SECURITY\_VIOLATION***

The request violates a security requirement.

**State incompatibility errors**

Error values in this category indicate that the Service Request was not compatible with the condition of a related CSTA object. This type includes the following specific error values:

***GENERIC\_STATE\_INCOMPATIBILITY***

The server is unable to be any more specific.

***INVALID\_OBJECT\_STATE***

The object is in the incorrect state for the Service. This general error value may be used when the server isn't able to be any more specific.

***INVALID\_CONNECTION\_ID\_FOR\_ACTIVE\_CALL***

The Connection identifier specified in the Active Call parameter of the request is not in the correct state.

***NO\_ACTIVE\_CALL***

The requested Service operates on an active call, but there is no active call.

***NO\_HELD\_CALL***

The requested Service operates on a held call, but the specified call is not in the Held state.

***NO\_CALL\_TO\_CLEAR***

There is no call associated with the CSTA Connection identifier of the Clear Call request.

***NO\_CONNECTION\_TO\_CLEAR***

There is no Connection for the CSTA Connection identifier specified as Connection To Be Cleared.

***NO\_CALL\_TO\_ANSWER***

There is no call active for the CSTA Connection identifier specified as Call To Be Answered.

***NO\_CALL\_TO\_COMPLETE***

There is no call active for the CSTA Connection identifier specified as Call To Be Completed.

**System resource availability errors**

Error values in this category indicate that the Service Request cannot be completed because of a lack of system resources within the serving sub-domain. This type includes one of the following specific error values:

***GENERIC\_SYSTEM\_RESOURCE\_AVAILABILITY***

The server is unable to be any more specific.

***SERVICE\_BUSY***

The Service is supported by the server, but is temporarily unavailable.

***RESOURCE\_BUSY***

An internal resource is busy. There is high probability that the Service will succeed if retried.

***RESOURCE\_OUT\_OF\_SERVICE***

The Service requires a resource that is Out Of Service. A Service Request that encounters this condition could initiate system problem determination actions (e.g. notification of the network administrator).

***NETWORK\_BUSY***

The server sub-domain is busy.

***NETWORK\_OUT\_OF\_SERVICE***

The server sub-domain is Out Of Service.

***OVERALL\_MONITOR\_LIMIT\_EXCEEDED***

This request would exceed the server's overall limit of monitors.

***CONFERENCE\_MEMBER\_LIMIT\_EXCEEDED.***

This request would exceed the server's limit on the number of members of a conference.

**Subscribed resource availability errors**

Error values in this category indicate that the Service Request cannot be completed because a required resource must be purchased or contracted by the client system. This type includes the following specific error values:

***GENERIC\_SUBSCRIBED\_RESOURCE\_AVAILABILITY***

The server is unable to be any more specific.

***OBJECT\_MONITOR\_LIMIT\_EXCEEDED***

This request would exceed the server's limit of monitors for the specified object.

***EXTERNAL\_TRUNK\_LIMIT\_EXCEEDED***

The limit of external trunks would be exceeded by this request.

***OUTSTANDING\_REQUEST\_LIMIT\_EXCEEDED***

The limit of outstanding requests would be exceeded by this request.

**Performance management errors**

Error values in this category indicate that an error has been returned as a performance management mechanism. This type includes the following specific error values:

***GENERIC\_PERFORMANCE\_MANAGEMENT***

The server is unable to be any more specific.

***PERFORMANCE\_LIMIT\_EXCEEDED***

A performance limit is exceeded.

**Security errors**

Error values in this category indicate that there is a security error. This type includes the following specific error values:

***UNSPECIFIED\_SECURITY\_ERROR***

The server is unable to be any more specific.

***SEQUENCE\_NUMBER\_VIOLATED***

This error indicates that the server has detected an error in the Sequence Number of the operation.

***TIME\_STAMP\_VIOLATED***

This error indicates that the server has detected an error in the Time Stamp of the operation.

***PAC\_VIOLATED***

This error indicates that the server has detected an error in the PAC of the operation.

***SEAL\_VIOLATED***

This error indicates that the server has detected an error in the Seal of the operation.

**CSTA Driver Interface Errors**

These errors derive from the Remote Operations CCITT Specification X.219 and may occur when a PBX Driver uses the CSTA interface to the Telephony Services.

***UNRECOGNIZED\_APDU\_REJECTION***

The given type of the APDU is not defined in the protocol.

***MISTYPED\_APDU\_REJECTION***

The structure of the APDU does not conform to the protocol.

***BADLY\_STRUCTURED\_APDU\_REJECTION***

APDU does not conform to X.208 or X.209 standard encoding.

***INITIATOR\_RELEASING\_REJECTION***

The requester is not willing to do the invoked operation because it is about to release the stream.

***UNRECOGNIZED\_LINKEDID\_REJECTION***

There is no operation in progress with an invoke ID equal to the specified link ID.

***LINKED\_RESPONSE\_UNEXPECTED\_REJECTION***

The invoked operation that the linked ID refers to is not a parent operation.

***UNEXPECTED\_CHILD\_OPERATION\_REJECTION***

The linked ID refers to a parent operation that does not allow the invoked operation.

***MISTYPED\_RESULT\_REJECTION***

The type of the Result parameter does not conform to the protocol.

***UNRECOGNIZED\_ERROR\_REJECTION***

The reported error is not in the protocol definition.

***UNEXPECTED\_ERROR\_REJECTION***

The reported error is not one that the operation may report.

***MISTYPED\_ARGUMENT\_REJECTION***

***MISTYPED\_PARAMETER\_REJECTION***

The type of a supplied error parameter is not consistent with the protocol specification

## TSAPI

The error codes below can occur within the TSAPI implementation of the ECMA CSTA standards. The ECMA standards do not define these errors.

### **RESOURCE\_LIMITATION\_REJECTION**

A Telephony Server or PBX Driver resource limitation prevents the system from processing the application request

### **ACS\_HANDLE\_TERMINATION\_REJECTION**

### **SERVICE\_TERMINATION\_REJECTION**

### **REQUEST\_TIMEOUT\_REJECTION**

### **REQUESTS\_ON\_DEVICE\_EXCEEDED\_REJECTION**

## Private Data

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## cstaAlternateCall()

The Alternate Call Service provides a higher-level, compound action of the Hold Call Service followed by Retrieve Call Service. This function will place an existing active call on hold and then either retrieves a previously held call or connects an alerting call at the same device.

### Syntax

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

```
RetCode_t    cstaAlternateCall(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *activeCall,
    ConnectionID_t *otherCall,
    PrivateData_t  *privateData);
```

### Parameters

#### **acsHandle**

This is the value of the unique handle to the opened 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.

***activeCall***

This parameter points to the connection identifier for the "Connected" or active call which is to be alternated.

***otherCall***

This parameter points to the connection identifier for the "Alerting" or "Held" call which is to be alternated.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTAAlternateCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

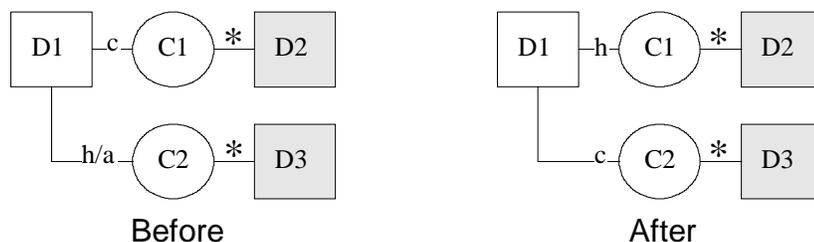
This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

A successful call to this function will cause the held-or-delivered call to be swapped with the active call.

As shown in the figure below, the Alternate Call Service places the user's active call to device D2 on hold and, in a combined action, establishes or retrieves the call between device D1 and 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/active call to device D3.

Figure 5-1 shows the operation of the Alternate Call Service.



**Figure 5-1. Alternate Call Service**

### CSTAAlternateCallConfEvent

The Alternate Call confirmation event provides the positive response from the server for a previous alternate call request.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            InvokeID_tinvokeID;
            union
            {
                CSTAAlternateCallConfEvent_t    alternateCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_ALTERNATE\_CALL\_CONF**, which identifies this message as an **CSTAAlternateCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaAnswerCall()**

---

The Answer Call function will connect an alerting call at the device which is alerting. The call must be associated with a device that can answer a call without requiring physical user manipulation.

### **Syntax**

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

RetCode_t    cstaAnswerCall (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *alertingCall,
    PrivateData_t  *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***alertingCall***

This parameter points to the connection identifier of the call to be answered.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTAAnswerCallConfEvent** 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 return value indicates that a bad or unknown *acsHandle* was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

The Answer Call Service works for an incoming call that is alerting a device. In the following figure the call C1 is delivered to device D1. The **csaAnswerCall()** is typically used with telephones that have attached speakerphone units to establish the call in a hands-free operation.



**Figure 5-2. Answer Call Service**

**CSTAAnswerCallConfEvent**

The Answer Call confirmation event provides the positive response from the server for a previous answer call request.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAAnswerCallConfEvent_t answerCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_ANSWER\_CALL\_CONF**, which identifies this message as an **CSTAAnswerCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaCallCompletion()**

---

The Call Completion Service invokes specific switch features that may complete a call that would otherwise fail. The feature to be activated is passed as a parameter to the function.

### **Syntax**

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

RetCode_t    cstaCallCompletion (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    Feature_t    feature,
    ConnectionID_t *call,
    PrivateData_t *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***feature***

Specifies the call completion feature that is desired. These include:

**CAMP\_ON** - queues the call until the device is available.

**CALL\_BACK** - requests the called device to return the call when it returns to idle.

**INTRUDE** - adds the caller to an existing active call at the called device. This feature requires the appropriate user security level at the server.

```
typedef enum Feature_t {
    FT_CAMP_ON = 0,
    FT_CALL_BACK = 1,
    FT_INTRUDE = 2
} Feature_t;
```

#### ***call***

This is a pointer to a connection identifier for the call to be completed.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

The application should always check the **CSTACallCompletionConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

Generally this Service is invoked when a call is established and it encounters a busy or no answer at the far device.

The Camp On feature allows queuing for availability of the far end device. Generally, Camp On makes the caller wait until the called party finishes the current call and any previously camped on calls. Call Back allows requesting the called device to return the call when it returns to idle. Call Back works much like Camp On, but the caller is allowed to hang up after invoking the service, and the CSTA Switching Function calls both parties when the called party becomes free. Intrude allows the caller to be added into an existing call at the called device.

### CSTACallCompletionConfEvent

The Call Completion confirmation event provides the positive response from the server for a previous call completion request.

## Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            InvokeID_tinvokeID;
            union
            {
                CSTACallCompletionConfEvent_t    callCompletion;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

## Parameters

### *acsHandle*

This is the handle for the newly opened ACS Stream.

### **eventClass**

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

### **eventType**

This is a tag with the value **CSTA\_CALL\_COMPLETION\_CONF**, which identifies this message as an **CSTACallCompletionConfEvent**.

### **invokeID**

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### **privateData**

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaClearCall()**

---

The Clear Call Service releases all of the devices from the specified call, and eliminates the call itself. The call ceases to exist and the connection identifiers used for observation and manipulation are released.

### **Syntax**

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

RetCode_t    cstaClearCall (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    ConnectionID_t *call,
    PrivateData_t *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***call***

This is a pointer to the connection identifier for the call to be cleared.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTAClearCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

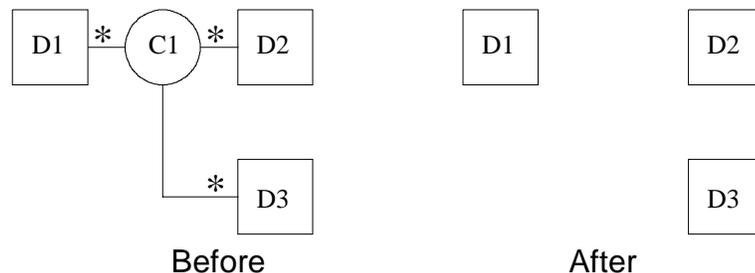
**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

This function will cause each device associated with a call to be released and the CSTA Connection Identifiers (and their components) are freed.

Figure 5-3 illustrates the results of a Clear Call (CSTA Connection ID = C1,D1), where call C1 connects devices D1, D2 and D3.



**Figure 5-3. Clear Call Service**

**CSTAClearCallConfEvent**

The Clear Call confirmation event provides the positive response from the server for a previous clear call request.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAClearCallConfEvent_t clearCall;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_CLEAR\_CALL\_CONF**, which identifies this message as an **CSTAClearCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This confirmation indicates that all instances of the ACS Connection Identifiers for all the endpoints in the call and in the current association have become invalid. The instances of identifiers should not be used to request additional services of the Telephony Server.

### cstaClearConnection()

The Clear Connection Service releases the specified device from the designated call. The Connection is left in the Null state. Additionally, the CSTA Connection Identifier provided in the Service Request is released.

### Syntax

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

RetCode_t    cstaClearConnection (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *call,
    PrivateData_t  *privateData);
```

### Parameters

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***call***

This is a pointer to the connection identifier for the connection to be cleared.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

#### ***acsHandle***

This is the value of the unique handle to the opened ACS Stream.

### Return Values

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

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

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

The application should always check the **CSTAClearConnectionConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

This Service releases the specified Connection and CSTA Connection Identifier instance from the designated call. The result is as if the device had hung up on the call. It is interesting to note that the phone may not 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( )** function is the same as **cstaClearCall( )**.

Figure 5-4 is an example of the results of a Clear Connection (CSTA Connection Id = C1,D3), where call C1 connects devices D1, D2 and D3. Note that it is likely that the call is not cleared by this Service if it is some type of conference.

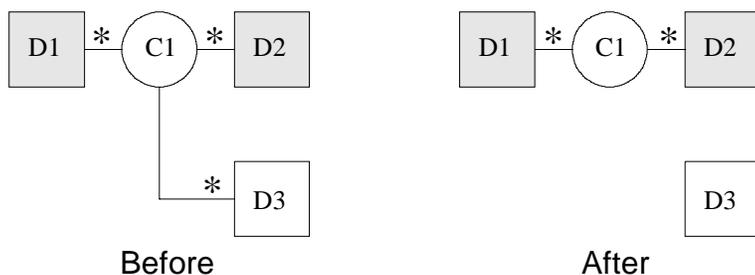


Figure 5-4. Clear Connection Service

### CSTAClearConnectionConfEvent

The Clear Connection confirmation event provides the positive response from the server for a previous clear connection request.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 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
            {
                CSTAClearConnectionConfEvent_t clearConnection;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

***acsHandle***

This is the handle for the newly opened ACS Stream.

***eventClass***

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

***eventType***

This tag with the value **CSTA\_CLEAR\_CONNECTION\_CONF** identifies this message as an CSTAClearConnectionConfEvent.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This confirmation event indicates that the instance of the ACS Connection Identifier for the cleared Connection is released. The identifier should not be used to request additional services of the Telephony Server.

## cstaConferenceCall()

This function provides the conference of an existing held call and another active call at a device. The two calls are merged into a single call and the two Connections at the conferencing device resolve into a single Connection in the Connected state. The pre-existing CSTA Connection Identifiers associated with the device creating the conference are released, and a new CSTA Connection Identifier for the resulting conferenced Connection is provided.

### Syntax

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

RetCode_t cstaConferenceCall (
    ACSHandle_t acsHandle,
    InvokeID_t invokeID,
    ConnectionID_t *heldCall,
    ConnectionID_t *activeCall,
```

```
PrivateData_t *privateData);
```

### Parameters

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***heldCall***

This is a pointer to the connection identifier for the call which is on hold and is to be conferenced with an active call.

#### ***activeCall***

This is a pointer to the connection identifier for the call which is active or proceeding and is to be conferenced with the held call.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

The application should always check the **CSTAConferenceCallConfEvent** 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***

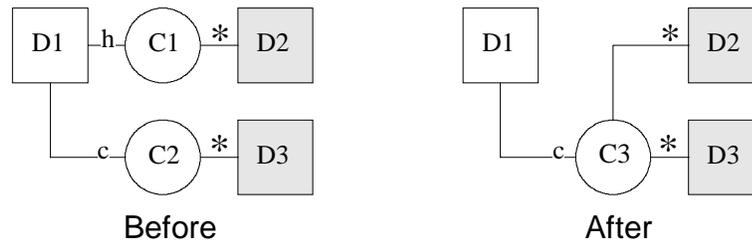
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

Figure 5-5 is an example of the starting conditions for the **cstaConferenceCall()** function, which are: the call C1 from D1 to D2 is in the held state. A call C2 from D1 to D3 is in progress or active.



**Figure 5-5. Conference Call Service**

D1, D2 and D3 are conferenced or joined together into a single call, C3. The value of the Connection identifier (D1,C3) may be that of one of the CSTA Connection Identifiers provided in the request (D1,C1 or D1,C2).

**CSTAConferenceCallConfEvent**

The Conference Call confirmation event provides the positive response from the server for a previous conference call request.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAConferenceCallConfEvent_t conferenceCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

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

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

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_CONFERENCE\_CALL\_CONF**, which identifies this message as an **CSTAConferenceCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***newCall***

This parameter specifies the resulting connection identifier for the calls which were conferenced at the Conferencing device. This connection identifier replaces the two previous connection identifier at that device.

***connList***

Specifies the resulting number of known devices in the conference. This field contains a count (***count***) of the number of devices in the conference and a pointer (***\*connection***) to an array of Connection\_t structures which define each connection in the call.

Each Connection\_t record contains the following:

*Party* - indicates the Connection ID of the party in the conference.

*Device* - provides the static reference for the party in the conference. This parameter may have a value that indicates the static identifier is not known.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

---

**cstaConsultationCall()**

---

The **cstaConsultationCall()** function will provide the compound or combined action of the Hold Call service followed by Make Call service. This service places an existing active call at a device on hold and initiates a new call from the same device using a single function call.

**Syntax**

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

```
RetCode_t    cstaConsultationCall (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    ConnectionID_t    *activeCall,
    DeviceID_t    *calledDevice,
    PrivateData_t    *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***activeCall***

This is a pointer to the connection identifier for the active call which is to be placed on hold before the new call is established.

### ***calledDevice***

This is a pointer to the destination device address for the new call to be established.

### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTAConsultationCallConfEvent** 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***

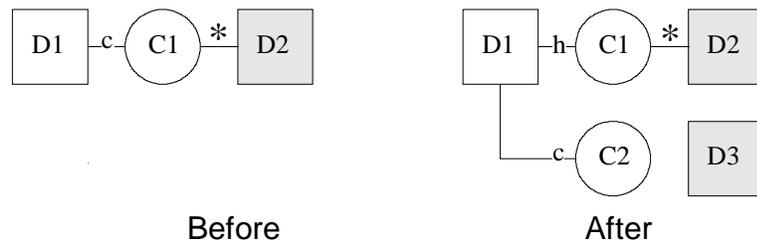
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

This compound service allows the application to place an existing call on hold and at the same time establish a new call to another device. In this case an active call C1 exists at D1 (see Figure 5.7) and a consultative call is desired to D3. After this function is called, the original active call (C1) is placed on hold and a new call, C2, is placed to device D3.



**Figure 5-6. Consultation Call Service**

### CSTAConsultationCallConfEvent

The Consultation Call confirmation event provides the positive response from the server for a previous consultation call request.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAConsultationCallConfEvent_t consultationCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This tag with the value **CSTA\_CONSULTATION\_CALL\_CONF**, identifies this message as an CSTAConsultationCallConfEvent.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***newCall***

Specifies the Connection ID for the originating connection of the new call originated by the Consultation Call request.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaDeflectCall()**

---

The **cstaDeflectCall()** service takes an alerting call at a device and redirects the call to a given number.

### **Syntax**

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

RetCode_t  cstaDeflectCall (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *deflectCall,
    DeviceID_t     *calledDevice,
    PrivateData_t  *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***deflectCall***

This is a pointer to the connection identifier of the call which is to be deflected to another device within the switch.

#### ***calledDevice***

A pointer to the device identifier where the original call is to be deflected.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTADeflectCallConfEvent** 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**

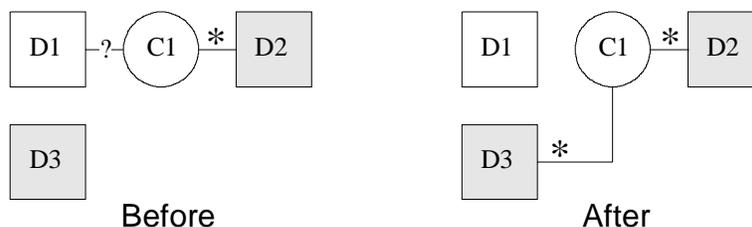
This return value indicates that a bad or unknown *acsHandle* was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

The Deflect Call Service takes a ringing (alerting) call at a device (D1) and sends it to a new destination (D3). This function replaces the original called device, as specified in the *deflectCall* parameter, with a different device within the switch, as specified in the *calledDevice* parameter.



**Figure 5-7. Deflect Call Service**

**CSTADeflectCallConfEvent**

The Deflect Call confirmation event provides the positive response from the server for a previous deflect call request.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 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
            {
                CSTADeflectCallConfEvent_t deflectCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the newly opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_DEFLECT\_CALL\_CONF**, which identifies this message as an **CSTADeflectCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaGroupPickupCall()**

---

The **cstaGroupPickupCall()** service moves an alerting call (at one or more devices in a device pickup group) to a specified device.

Group Pickup is a PBX feature where a “Pickup Group” is defined on the PBX (independent of Telephony Service Device Groups). When a call rings at a station in the pickup group, a pickup group member may invoke the PBX’s “Group Pickup” feature and thereby redirect the ringing call to their phone. The application does not specify the call that is to be redirected.

### **Syntax**

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

RetCode_t    cstaGroupPickupCall (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *deflectCall, /* Version 1 Stream Only*/
    DeviceID_t     *pickupDevice,
    PrivateData_t  *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***deflectCall***

**TSAPI Version 1 Stream:** Pointer to the call being picked up.

**TSAPI Version 2 Stream:** This parameter is ignored.

#### ***pickupDevice***

This is a pointer to the device which is picking up calls from the group.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTAGroupPickupConfEvent** 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**

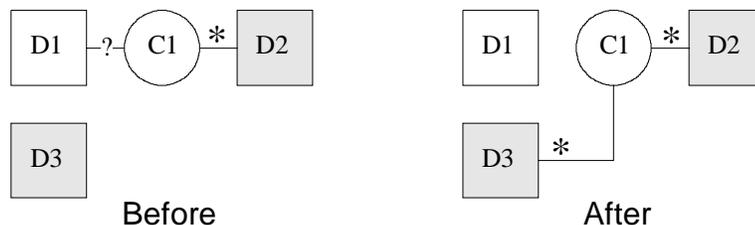
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## Comments

The `cstaGroupPickupCall( )` service redirects an alerting call (at one of more devices in a device pickup) to a specified device, the **pickupDevice**.



**Figure 5-8. Group Pickup Call Service**

## CSTAGroupPickupCallConfEvent

The Group Pickup Call confirmation event provides the positive response from the server for a previous Group Pickup call request.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAGroupPickupCallConfEvent_t groupPickupCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### **acsHandle**

This is the handle for the newly opened ACS Stream.

#### **eventClass**

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

#### **eventType**

This is a tag with the value **CSTA\_GROUP\_PICKUP\_CALL\_CONF**, which identifies this message as an **CSTAGroupPickupCallConfEvent**.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaHoldCall()**

The **cstaHoldCall()** service places an existing Connection in the held state.

**Syntax**

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

RetCode_t cstaHoldCall (
    ACSHandle_t      acsHandle,
    InvokeID_t       invokeID,
    ConnectionID_t   *activeCall,
    Boolean           reservation,
    PrivateData_t    *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***activeCall***

A pointer to the connection identifier for the active call to be placed on hold.

***reservation***

Reserves the facility for reuse by the held call. This option is not appropriate for most non-ISDN telephones. The default is no connection reservation.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

The application should always check the **CSTAHoldCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

A call to this function will interrupt communications for an existing call at a device. The call is usually, but not always, in the active state. A call may be placed on hold by the user some time after completion of dialing. The associated connection for the held call is made available for other uses, depending on the reservation option (ISDN-case). As shown in Figure 5-9, if the Hold Call service is invoked for device D1 on call C1, then call C1 is placed on hold at device D1. The hold relationship is affected at the holding device.



**Figure 5-9. Hold Call Service**

The **cstaHoldCall()** service maintains a relationship between the holding device and the held call that lasts until the call is retrieved from the hold status, or until the call is cleared.

## **CSTAHoldCallConfEvent**

---

The Hold Call confirmation event provides the positive response from the server for a previous Hold call request

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAHoldCallConfEvent_t holdCall;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAHoldCallConfEvent_t {
    Nulltype      null;
} CSTAHoldCallConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_HOLD\_CALL\_CONF**, which identifies this message as an **CSTAHoldCallConfEvent**.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaMakeCall()**

The **cstaMakeCall()** service originates a call between two devices. The originator must be on the switch. The service attempts to create a new call and establish a connection between the calling device (originator) and the called device (destination). The Make Call service also provides a CSTA Connection Identifier that indicates the Connection of the originating device.

**Syntax**

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

RetCode_t  cstaMakeCall (
    ACSHandle_t  acsHandle,
    InvokeID_t   invokeID,
    DeviceID_t   *callingDevice,
    DeviceID_t   *calledDevice,
    PrivateData_t *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***callingDevice***

A pointer to the device identifier of the device which is to originate the new call.

***calledDevice***

A pointer to the device identifier for the destination device for the new call.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTAMakeCallConfEvent** 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***

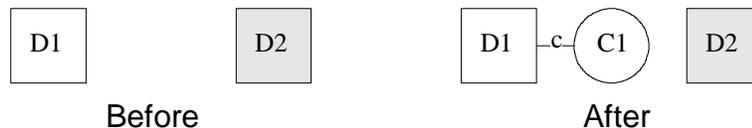
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

The **cstaMakeCall( )** service originates a call between two application designated devices. When the service is initiated, the calling device is prompted (if necessary), and, when that device acknowledges, a call to the called device is originated. Figure 5-11 illustrates the results of a Make Call service request (Calling device = D1, Called device = D2). A call is established as if D1 had called D2, and the client is returned the Connection id: (C1,D1).



**Figure 5-10. Make Call Service**

The establishment of a complete call connection can be a multi-stepped process depending on the destination of the call. Call status event reports (see *Status Reporting Service*) may be sent by the Telephony Server to the service requesting client application as the connection establishment progresses. These events are in addition to the standard confirmation events (e.g. **CSTAMakeCallConfEvent**) which only indicates that the switch is attempting to establish a connection between the two devices. The application should be aware that the requested call is not guaranteed to succeed even after a successful Make Call service confirmation event has been received. The application must monitor status events to be informed of the call status as it progresses. Status event reports can be established by using the **cstaMonitorStart( )** service (see *Status Reporting Services*).

## **CSTAMakeCallConfEvent**

---

The Make Call confirmation event provides the positive response from the server for a previous Make Call service request.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAMakeCallConfEvent_t makeCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t newCall;
} CSTAMakeCallConfEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_MAKE\_CALL\_CONF**, which identifies this message as an **CSTAMakeCallConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***newCall***

Specifies the Connection ID for the originating connection of the new call originated by the Make Call request.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaMakePredictiveCall()**

The **cstaMakePredictiveCall()** service originates a call between a group of devices or a logical device on behalf of an originating (calling) device. The service creates a new call and establishes a Connection with the terminating (called) device. The Make Predictive Call service also provides a CSTA Connection Identifier that indicates the Connection of the terminating (called) device.

### **Syntax**

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

RetCode_t  cstaMakePredictiveCall (
    ACSHandle_t      acsHandle,
    InvokeID_t       invokeID,
    DeviceID_t       *callingDevice,
    DeviceID_t       *calledDevice,
    AllocationState_t allocationState;
    PrivateData_t    *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***callingDevice***

A pointer to the device identifier of the device which is to originate the new call.

### ***calledDevice***

A pointer to the device identifier of the device being call, i.e. the destination device.

### ***allocationState***

This parameter specifies under which condition the connection with the destination is to be connected to the calling or originating device. If this parameter is not specified by the application, the Call Delivered state will be the default. This parameter may be one of the following values:

*Call Delivered:* this value specifies that the switch should attempt to connect the call to the caller (originating device), if the Alerting or Connected state is determined at the called party (destination device).

*Call Established:* this value specifies that the switch should attempt to connect the call to the caller (originating device), if the Connected state is determined at the called party (destination device).

### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTAMakePredictiveCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

This service is often used to initiate a call to a called device (destination) from a group of devices or a logical device without first establishing a connection with a calling device (originator). This service allocates the call to a particular device within that group at some time during the progress of the call.

The **csstaMakePredictiveCall( )** service first initiates a call to the called device (destination). Depending on the call's progress, the call may be connected with the calling device (originator) during the progress of the call. The point at which the switch will attempt to connect the call to the originating device is determined by the **allocation State** parameter. If the allocation parameter is set to Call Delivered, then the call is allocated upon detection of an Alerting (or Connected) Connection state at the destination. If the allocation parameter is set to Call Established, then the call is allocated upon detection of a Connected Connection state at the recipient. In other words, the call is connected to the originating device if the call state is either alerting or connected at the far end or connected, respectively.

Figure 5-12 illustrates the results of a Make Predictive Call (Calling device = group device D1, Called device = D2).



---

**Figure 5-11. Make Predictive Call Service**

## **CSTAMakePredictiveCallConfEvent**

---

The Make Predictive Call confirmation event provides the positive response from the server for a previous **cstaMakePredictiveCall()** request.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAMakePredictiveConfEvent_t makePredictiveCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t newCall;
} CSTAMakePredictiveConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **MAKE\_PREDICTIVE\_CALL\_CONF**, which identifies this message as an **CSTAMakePredictiveConfEvent**.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***newCall***

Specifies the Connection ID for the far-end connection of the new call originated by the Make Predictive Call request.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

---

**cstaPickupCall()**

The **cstaPickupCall()** service takes a ringing (alerting) call at a device and redirects the call to a specified device.

**Syntax**

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

RetCode_t  cstaPickupCall (
    ACSHandle_t  acsHandle,
    InvokeID_t   invokeID,
    ConnectionID_t *deflectCall,
    DeviceID_t   *calledDevice,
    PrivateData_t *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***deflectCall***

This is a pointer to the connection identifier of the call which is to be picked up from another device within the switch.

***calledDevice***

A pointer to the device identifier of the device which is picking up the original call.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTAPickupCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

The **cstaPickupCall()** service takes an alerting call at a device within the switch and brings it to a local device destination. This function picks up a call, **deflectCall**, at the device specified in the **calledDevice** parameter.

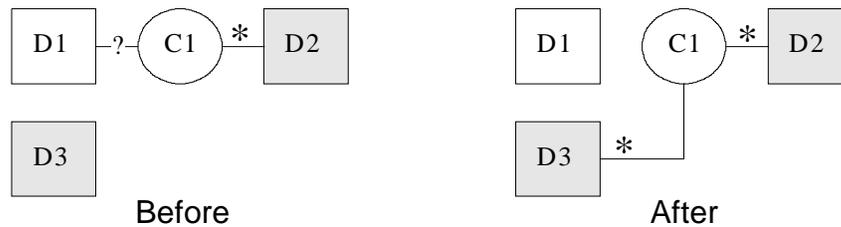


Figure 5-12. Pickup Call Service

### CSTAPickupCallConfEvent

The Pickup Call confirmation event provides the positive response from the server for a previous pickup call request.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAPickupCallConfEvent_t  pickupCall;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

## Parameters

### ***acsHandle***

This is the handle to an active ACS Stream.

### ***eventClass***

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

### ***eventType***

This is a tag with the value **CSTA\_PICKUP\_CALL\_CONF**, which identifies this message as an **CSTAPickupCallConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaReconnectCall()**

The **cstaReconnectCall()** service provides the compound action (combination) of the **cstaClearConnection()** service followed by the **cstaRetrieveCall()** service. The service clears an existing Connection and then retrieves a previously Held Connection at the same device.

### **Syntax**

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

```
RetCode_t  cstaReconnectCall (
    ACSHandle_t  acsHandle,
    InvokeID_t   invokeID,
    ConnectionID_t *activeCall,
    ConnectionID_t *heldCall,
    PrivateData_t *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***activeCall***

A pointer to the connection identifier of the active call which is to be cleared.

### ***heldCall***

A pointer to the connection identifier of the held call which is to be retrieved.

### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTARReconnectCallConfEvent** 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***

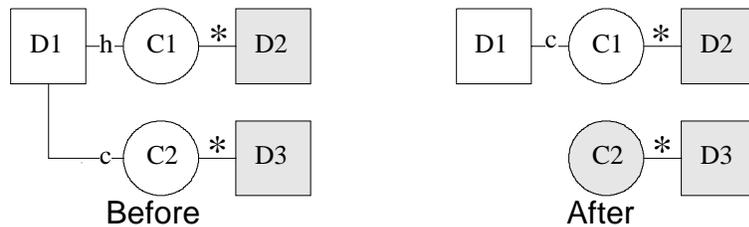
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

A successful request of this service will cause an existing active call to be dropped. Once the active call has been dropped, the specified held call at the device is retrieved and becomes active. This service is typically used to drop an active call and return to a held call; however, it can also be used to cancel of a consultation call (because of no answer, called device busy, etc.) followed by returning to a held call.



**Figure 5-13. Reconnect Call Service**

### CSTARereconnectCallConfEvent

The Reconnect Call confirmation event provides the positive response from the server for a previous Reconnect call request.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAReconnectCallConfEvent_t reconnectCall;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle to an active ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_RECONNECT\_CALL\_CONF**, which identifies this message as an **CSTAReconnectCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaRetrieveCall()**

---

The **cstaRetrieveCall()** service connects an existing Held Connection. The state of the specified call changes from held to active.

### **Syntax**

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

RetCode_t    cstaRetrieveCall (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *heldCall,
    PrivateData_t  *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***heldCall***

A pointer to the connection identifier of the held call which is to be retrieved.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTARRetrieveCallConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

The indicated held Connection is restored to the Connected state (active). The call state can change depending on the actions of far end endpoints. If the **cstaHoldCall()** service reserved the Held Connection and the **cstaRetrieveCall()** service is requested for the same call, then the Retrieve Call service uses the reserved Connection.



**Figure 5-14. Retrieve Call Service**

**CSTARRetrieveCallConfEvent**

The Retrieve Call confirmation event provides the positive response from the server for a previous Retrieve call request.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in Section 4 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
            {
                CSTARetrieveCallConfEvent_t retrieveCall;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle to an active ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_RETRIEVE\_CALL\_CONF**, which identifies this message as an **CSTARRetrieveCallConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaTransferCall()**

---

The **cstaTransferCall()** service provides the transfer of a held call with an active call at the same device. The transfer service merges two calls with Connections to a single common device. Also, both of the Connections to the common device become Null and their Connections Identifiers are released.

### **Syntax**

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

RetCode_t  cstaTransferCall (
    ACSHandle_t      acsHandle,
    InvokeID_t       invokeID,
    ConnectionID_t   *heldCall,
    ConnectionID_t   *activeCall,
    PrivateData_t    *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***heldCall***

A pointer to the connection identifier of the held call which is to be transferred.

#### ***activeCall***

A pointer to the connection identifier of the active call to which the held call is to be transferred.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTATTransferCallConfEvent** 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**

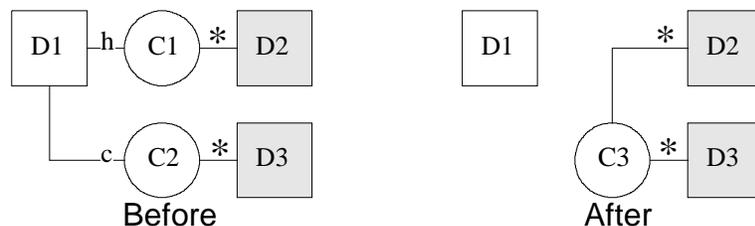
This return value indicates that a bad or unknown **acsHandle** was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

Referring to Figure 5-16, the starting conditions for the **cstaTransferCall()** service are: the call C1 from D1 to D2 is in held state (**heldCall**). A call C2 from D1 to D3 is in progress or active (**activeCall**). This service transfers the existing (held) call between devices D1 and D2 into a new call with a new call identifier from device D2 to a device D3.



**Figure 5-15. Transfer Call Service**

The request is used in the situation where the call from D1 to D3 is established (active) or if the call is in any state other than Failed or Null state. The Transfer Call service successfully completes, and D1 is released from the call.

## CSTATransferCallConfEvent

The Transfer Call confirmation event provides the positive response from the server for a previous transfer call request.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in Section 4 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
            {
                CSTATransferCallConfEvent_t transferCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct Connection_t {
    ConnectionID_t party;
    SubjectDeviceID_t staticDevice; /* NULL for not present */
} Connection_t;

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

typedef struct {
    ConnectionID_t newCall;
    ConnectionList_t connList;
} CSTATransferCallConfEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_TRANSFER\_CALL\_CONF**, which identifies this message as an **CSTATransferCallConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***newCall***

Specifies the resulting Connection Identifier for the transferred call.

### ***connList***

Specifies the resulting number of known devices in the transferred call. This field contains a count (**count**) of the number of devices in the transferred call and a pointer (**\*connection**) to an array of pointers that point to ConnectionID\_t structures which define each connection in the call.

Each ConnectionID\_t record contains the following:

*Party* - indicates the Connection ID of the party in the transferred call.

*Device* - provides the static reference for the party in the transferred call. This parameter may have a value that indicates the static identifier is not known.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Telephony Supplementary Services

---

This section describes those CSTA telephony services that switches typically provide as "features".

### ⇒ NOTE:

Not all switches support all these functions and events.

Applications can use the Telephony Supplementary Services (defined in this section) to manipulate telephony objects.. As with other TSAPI services already described, these function will generate associated confirmation events from the Telephony Server. Similarly (as described in Chapter 4, **Sending CSTA Requests and Responses**) applications can use the *invokeID* to match a specific confirmation event with the specific function call, or they may use application-generated *invokeIDs* to index into a data structure.

To receive events, an application must have an active ACS Stream and an implement an event handling mechanism. Further, the reception of unsolicited events requires an active monitor. See the **Control Services** and **Status Reporting Services** sections for more information on events.

### cstaSetMsgWaitingInd()

---

The **cstaSetMsgWaitingInd()** service provides the application with the ability to turns on and off a message waiting indicator on a telephony device.

#### Syntax

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

```
RetCode_t    cstaSetMsgWaitingInd (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    Boolean        messages,
    PrivateData_t  *privateData);
```

#### Parameters

##### **acsHandle**

This is the value of the unique handle to the opened 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.

***device***

This parameter is a pointer to the device identifier of the device on which to set the message waiting indicator. .

***messages***

This parameter identifies whether to turn on or off the message waiting indicator at the device specified by ***device*** parameter. A value of TRUE indicates that the message waiting indicator should be tuned on, FALSE indicates that the indicator should be turn off.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTASetMwiConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## **CSTASetMwiConfEvent**

---

The Set Message Waiting Indicator confirmation event provides the positive response from the Telephony Server for a previous Set Message Waiting Indicator service request. When the application receives this event the message waiting indicator has been set as requested by the application.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTASetMwiConfEvent_t setMwi;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetMwiConfEvent_t {
    Nulltype    null;
} CSTASetMwiConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_SET\_MWI\_CONF**, which identifies this message as an **CSTASetMessageWaitingIndConfEvent**.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaSetDoNotDisturb()**

The **cstaSetDoNotDisturb()** service activates the switch feature that prevents calls from alerting at a specified device by deflecting the calls from the original destination to other devices.

**Syntax**

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

RetCode_t    cstaSetDoNotDisturb (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    Boolean        doNotDisturb,
    PrivateData_t  *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***device***

This parameter is a pointer to the device identifier of the device on which the Do Not Disturb feature is to be activated. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

***doNotDisturb***

This parameter identifies whether to turn on or off the Do Not Disturb feature at the device specified by ***device*** parameter. A value of TRUE indicates that the Do Not Disturb feature should be tuned on, FALSE indicates that the feature should be turn off.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTASetDndConfEvent** 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 return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

***agentMode***

This parameter specifies the work mode state which the agent will be moved to. This could be one of the following:

- LOG\_IN**
- LOG\_OUT**
- NOT\_READY**
- READY**
- WORK\_NOT\_READY**
- WORK\_READY**

***agentID***

A pointer to the agent identifier of the ACD Agent whose work mode is to be changed.

***agentGroup***

A pointer to the agent group identifier for the ACD group or split in which the agent will be logged into or out of. This parameter is only required when the ***agentMode*** parameter is set for the **LOG\_IN** and **LOG\_OUT** work modes.

***agentPassword***

A pointer to a password that allows the agent to log into an ACD split or group. This parameter is only required when the ***agentMode*** parameter is set for the **LOG\_IN** and **LOG\_OUT** work modes.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTASetAgentStateConfEvent** message to ensure that the service request has been acknowledged and

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaSetForwarding()**

The **cstaSetForwarding()** service activates and deactivates several types of forwarding features on a specified device.

**Syntax**

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

RetCode_t    cstaSetForwarding (

    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    DeviceID_t    *device,
    ForwardingType_t    forwardingType,
    Boolean    forwardingOn,
    DeviceID_t    *forwardingDestination,
    PrivateData_t    *privateData);

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;
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***device***

This parameter is a pointer to the device identifier of the device on which forwarding is to be set. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

### ***forwardingType***

This parameter specifies the type of forwarding to set or clear at the requested device. The possible types include:

<b>Immediate</b>	Forwarding all calls
<b>Busy</b>	Forwarding when busy
<b>No Answer</b>	Forwarding after no answer
<b>Busy Internal</b>	Forwarding when busy for an internal call
<b>Busy External</b>	Forwarding when busy for an external call
<b>No Answer Internal</b>	Forwarding after no answer for an internal call
<b>No Answer External</b>	Forwarding after no answer for an external call.

### ***forwardingOn***

This parameter identifies whether to turn on or off the forwarding feature at the device specified by **device** parameter. A value of TRUE indicates that the forwarding feature should be tuned on, FALSE indicates that the feature should be turn off.

***forwardingDestination***

This is a pointer to the device identifier for the device to which the calls are to be forwarded.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTASetFwdConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**CSTASetFwdConfEvent**

---

The Set Forwarding confirmation event provides the positive response from the server for a previous Set Forwarding service request. When this event is received by the application the forwarding feature has been set as requested.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_tinvokeID;
            union
            {
                CSTASetFwdConfEvent_t    setFwd;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetFwdConfEvent_t {
    Nulltype    null;
} CSTASetFwdConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_SET\_FWD\_CONF**, which identifies this message as an **CSTASetFwdConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaSetAgentState()**

---

The **cstaSetAgentState()** service changes an ACD agents work mode to one specified by this service.

### **Syntax**

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

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 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];
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***device***

This parameter points to the device identifier for the ACD agent for which the work mode is to be changed. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

***agentMode***

This parameter specifies the work mode state which the agent will be moved to. This could be one of the following:

***LOG\_IN***

***LOG\_OUT***

***NOT\_READY***

***READY***

***WORK\_NOT\_READY***

***WORK\_READY***

***agentID***

A pointer to the agent identifier of the ACD Agent whose work mode is to be changed.

***agentGroup***

A pointer to the agent group identifier for the ACD group or split in which the agent will be logged into or out of. This parameter is only required when the ***agentMode*** parameter is set for the ***LOG\_IN*** and ***LOG\_OUT*** work modes.

***agentPassword***

A pointer to a password that allows the agent to log into an ACD split or group. This parameter is only required when the ***agentMode*** parameter is set for the ***LOG\_IN*** and ***LOG\_OUT*** work modes.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTASetAgentStateConfEvent** 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 return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## **CSTASetAgentStateConfEvent**

The Set Agent State confirmation event provides the positive response from the server for a previous Set Agent State service request. When this event is received by the application the agent state has been set as requested.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** and ***CSTA Data Types*** in Section 4 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
            {
                CSTASetAgentStateConfEvent_t setAgentState;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetAgentStateConfEvent_t {
    Nulltype    null;
} CSTASetAgentStateConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_SET\_AGENT\_STATE\_CONF**, which identifies this message as an **CSTASetAgentStateConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaQueryMsgWaitingInd()**

The **cstaQueryMessageWaitingInd()** service provides the current state of the message waiting indicator of a specified device.

### Syntax

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

RetCode_t  cstaQueryMsgWaitingInd (
    ACSHandle_t  acsHandle,
    InvokeID_t   invokeID,
    DeviceID_t   *device,
    PrivateData_t *privateData);
```

### Parameters

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

***device***

This parameter is a pointer to the device identifier of the device on which the message waiting indicator is being queried. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTAQueryMwiConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## CSTAQueryMwiConfEvent

The Query Message Waiting Indicator confirmation event provides the positive response from the server for a previous Query Message Waiting Indicator service request. This event informs the application whether there are any messages waiting, i.e. whether the message waiting indicator is turned on or off.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAQueryMwiConfEvent_t  queryMwi;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryMwiConfEvent_t {
    Boolean    messages;
} CSTAQueryMwiConfEvent_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 CSTA confirmation event.

#### **eventType**

This is a tag with the value **CSTA\_QUERY\_MWI\_CONF**, which identifies this message as an **CSTAQueryMwiConfEvent**.

***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***messages***

This parameter specifies whether there are any messages waiting at the requested device. TRUE indicates that there are messages waiting, FALSE indicates that there are none.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

---

**cstaQueryDoNotDisturb()**

The **cstaQueryDoNotDisturb()** service provides the current state of the do not disturb feature on a specific device.

**Syntax**

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

RetCode_t    cstaQueryDoNotDisturb (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***device***

This parameter is a pointer to the device identifier of the device on which the Do Not Disturb feature is being queried. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

**Return Values**

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

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

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

The application should always check the **CSTAQueryDndConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**CSTAQueryDndConfEvent**

The Query Do Not Disturb confirmation event provides the positive response from the server for a previous Query Do Not Disturb service request. This event informs the application whether the feature is turned on or off.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAQueryDndConfEvent_t    queryDnd;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct
{
    Boolean_t doNotDisturb;
} CSTAQueryDndConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_QUERY\_DND\_CONF**, which identifies this message as an **CSTAQueryDndConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***doNotDisturb***

This parameter specifies whether the Do Not Disturb feature is active at the requested device. TRUE indicates that the feature is turned on. FALSE indicates that the feature is turned off.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaQueryFwd()**

The **cstaQueryFwd()** service provides the current state of the forwarding feature(s) on a specific device.

**Syntax**

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

RetCode_t    cstaQueryFwd (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);
```

**Parameters*****acsHandle***

This is the value of the unique handle to the opened 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.

***device***

This parameter is a pointer to the device identifier of the device on which the forwarding feature is being queried. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

*Application-generated Identifiers* - if the function call completes successfully it will return a zero (0) value. If the call fails a negative error (<0) condition will be returned. For application-generated identifiers the return will never be positive (>0). The application should always check the **CSTAQueryFwdConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### CSTAQueryFwdConfEvent

The Query Forwarding confirmation event provides the positive response from the server for a previous Query Forwarding service request. The event also informs the application of the forwarding type, whether forwarding is on or off, and the forwarding destination for each device requested.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAQueryFwdConfEvent_t queryFwd;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0,
    FWD_BUSY = 1,
    FWD_NO_ANS = 2,
    FWD_BUSY_INT = 3,
    FWD_BUSY_EXT = 4,
    FWD_NO_ANS_INT = 5,
    FWD_NO_ANS_EXT = 6
} ForwardingType_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t forwardingType;
    Boolean          forwardingOn;
    DeviceID_t      forwardDN;
} ForwardingInfo_t;

typedef struct ListForwardParameters_t {
    short          count;
    ForwardingInfo_t param[7];
} ListForwardParameters_t;

typedef struct CSTAQueryFwdConfEvent_t {
    ListForwardParameters_t forward;
} CSTAQueryFwdConfEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_QUERY\_FWD\_CONF**, which identifies this message as an **CSTAQueryFwdConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***queryFwd***

This parameter is a **ListForwardParameters\_t** structure which contains the following:

#### ***count***

This parameter indicates how many forwarding list entries are provided. Each entry corresponds to a different device.

#### ***param***

An array of ForwardingInfo\_t structures, each of which is composed of the following elements.

#### ***forwardingType***

Specifies the type of forwarding set. The types include:

<b>Immediate</b>	Forwarding all calls
<b>Busy</b>	Forwarding when busy
<b>No Answer</b>	Forwarding after no answer
<b>Busy Internal</b>	Forwarding when busy for an internal call
<b>Busy External</b>	Forwarding when busy for an external call
<b>No Answer Internal</b>	Forwarding after no answer for an internal call
<b>No Answer External</b>	Forwarding after no answer for an external call.

***forwardingOn***

Indicates whether forwarding is active or inactive. TRUE indicates forwarding is active. FALSE indicates forwarding is inactive.

***forwardingDN***

Specifies the forward-to destination device for the type of forwarding listed.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**cstaQueryAgentState()**

---

The **cstaQueryAgentState()** service will provide the application with the current agent state at a device.

**Syntax**

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

RetCode_t cstaQueryAgentState (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    DeviceID_t    *device,
```

Issue 1 — December 2002

```
PrivateData_t *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***device***

This parameter is a pointer to the device identifier of the device on which the agent state is being queried. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTAQueryAgentStateConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### CSTAQueryAgentStateConfEvent

The Query Agent State confirmation event provides the positive response from the server for a previous Query Agent State service request. This event will provide the application with the current agent state.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in Section 4 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
            {
                CSTAQueryAgentStateConfEvent_t queryAgentState;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef enum AgentState_t {
    AG_NOT_READY = 0,
    AG_NULL = 1,
    AG_READY = 2,
    AG_WORK_NOT_READY = 3,
    AG_WORK_READY = 4
} AgentState_t;

typedef struct CSTAQueryAgentStateConfEvent_t {
    AgentState_t    agentState;
} CSTAQueryAgentStateConfEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_QUERY\_AGENT\_STATE\_CONF**, which identifies this message as an **CSTAQueryAgentStateConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***agentState***

This parameter specifies the current work mode state of the agent. The possible agent states are:

<b>Null</b>	This indicates that an agent is logged out of the group or device that they serve.
<b>Not Ready</b>	This state indicates that an agent is occupied with some task other than that of serving a call.
<b>Ready</b>	This state indicates that an agent is ready to accept calls.
<b>Work/Not Ready</b>	This state indicates that an agent is occupied with after call work. It also implies that the agent should not receive additional ACD calls.
<b>Work/Ready</b>	This state indicates that an agent is occupied with after call work. It also implies that the agent may receive additional ACD calls.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaQueryLastNumber()**

---

The **cstaQueryLastNumber()** service provides the last number dialed by a specified device.

### **Syntax**

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

RetCode_t    cstaQueryLastNumber (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***device***

This parameter is a pointer to the device identifier of the device on which the last number is being queried. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

#### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### **Return Values**

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

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

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

The application should always check the **CSTAQueryLastNumberConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## **CSTAQueryLastNumberConfEvent**

The Query Last Number confirmation event provides the positive response from the server for a previous Query Last Number request. This event provides the last number that was dialed from the requested device.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 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
            {
                CSTAQueryLastNumberConfEvent_t    queryLastNumber;
            } u;
        }
    }
} cstaConfirmation;
```

```
    } event;  
} CSTAEvent_t;
```

```
typedef struct
{
    DeviceID_t lastNumber,
} CSTAQueryLastNumberConfEvent_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 CSTA confirmation event.

#### ***eventType***

This is a tag with the value **CSTA\_QUERY\_LAST\_NUMBER\_CONF**, which identifies this message as an **CSTAQueryLastNumberConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

#### ***lastNumber***

This parameter indicates the last number dialed at the requested device.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **cstaQueryDeviceInfo()**

The **cstaQueryDeviceInfo()** service provides general information about a device. The confirmation event for this service will include information on the class and type of device being queried.

### **Syntax**

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

RetCode_t cstaQueryDeviceInfo (
    ACSHandle_t acsHandle,
    InvokeID_t invokeID,
    DeviceID_t *device,
    PrivateData_t *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***device***

This parameter is a pointer to the device identifier of the device for which information is being requested. This parameter may be different than the originating device depending on the security level defined for the originating device in the Telephony Server.

### ***privateData***

This is a pointer to the private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

The application should always check the **CSTAQueryDeviceInfoConfEvent** 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 return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## CSTAQueryDeviceInfoConfEvent

The Query Device Info confirmation event provides the positive response from the server for a previous Query Device Info request. This event provides the application with type and class of the requested device.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in Section 4 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
            {
                CSTAQueryDeviceInfoConfEvent_t queryDeviceInfo;
            } u;
        } cstaConfirmation;

        } event;
} CSTAEvent_t;

typedef enum DeviceType_t {
    DT_STATION = 0,
    DT_LINE = 1,
    DT_BUTTON = 2,
    DT_ACD = 3,
    DT_TRUNK = 4,
    DT_OPERATOR = 5,
    DT_STATION_GROUP = 16,
    DT_LINE_GROUP = 17,
    DT_BUTTON_GROUP = 18,
    DT_ACD_GROUP = 19,
    DT_TRUNK_GROUP = 20,
    DT_OPERATOR_GROUP = 21,
    DT_OTHER = 255
}
```

```
} DeviceType_t;
```

```
typedef unsigned char DeviceClass_t;
#define DC_VOICE 0x80
#define DC_DATA 0x40
#define DC_IMAGE 0x20
#define DC_OTHER 0x10

typedef struct CSTAQueryDeviceInfoConfEvent_t {
    DeviceID_t device;
    DeviceType_t deviceType;
    DeviceClass_t deviceClass;
} CSTAQueryDeviceInfoConfEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_QUERY\_DEVICE\_INFO\_CONF**, which identifies this message as an **CSTAQueryDeviceInfoConfEvent**.

### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

### ***deviceIdentifier***

May provide an alternate short-form static device identifier for the device requested.

**deviceType**

This parameter indicates the type of device being queried. The possible device types are:

<b>ACD</b>	Automatic Call Distributor (ACD)
<b>ACD group</b>	Automatic Call Distributor (ACD) group
<b>Button</b>	is one instance of a call manipulation point at an individual station.
<b>Button group</b>	is two or more instances of a call manipulation point at an individual station.
<b>Line</b>	is a communications interface to one or more stations.
<b>Line group</b>	is a set of communications interfaces to one or more stations.
<b>Operator</b>	also known as Attendant
<b>Operator group</b>	two or more operator devices used interchangeably or addressed identically.
<b>Other</b>	is any other type for which there is no enumeration defined.
<b>Station</b>	is the traditional telephone device, either simple or featured.
<b>Station group</b>	is two or more stations used interchangeably or addressed identically.
<b>Trunk</b>	a device used to access other switching sub-domains.
<b>Trunk group</b>	typically, two or more trunks providing connections to the same place.

See the **Functional Call Model** section of this document for more information on device types. Not all switch implementations will support all the device types listed.

***deviceClass***

This parameter indicates the class of device being queried. The possible device classes are:

- Voice** a device that is used to make audio calls. This class includes all normal telephones, as well as computer modems and G3 facsimile machines.
- Data** a device that is used to make digital data calls (either circuit switched or packet switched). This class includes computer interfaces and G4 facsimile machines.
- Image** a device that is used to make digital data calls involving imaging, or high speed circuit switched data in general. This class includes video telephones and CODECs.
- Other** a type of device not covered by data, image, or voice.

See the *Functional Call Model* section of this document for more information on device classes. Not all switch implementations will support all the device classes listed.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

---

TSAPI Status Services are those function calls and events that are related to unsolicited TSAPI event messages. External telephony activity on the switch, External telephony activity at a device, or human user activity can all cause unsolicited TSAPI events. Such events messages are asynchronous in nature. An application typically cannot anticipate their arrival. For example, an event informing the application of an incoming call to a device (e.g. a telephone station) is an unsolicited, asynchronous TSAPI event (since the application did not initiate the call and such a call can arrive at any time).

Applications use the status reporting request function to turn-on or turn-off status event reporting for a CSTA device (e.g. a desktop telephone). An application can use this function to turn-on/turn-off status reporting for any stations on the switch where monitoring is required (assuming proper access permissions are administered at the Telephony Server).

### **Status Reporting Functions and Confirmation Events**

---

Applications may use Event reporting to determine the changes in the state of a call or a connection associated with a device which is of interest to the application. This section describes the functions that an application uses to request unsolicited event reports for a telephony device or for calls. In the case of calls, the application must first control the calls.

Events provide the application with information about the state of calls or connection. The application may keep track of device or call states. If the application needs to maintain call state information for a specific device or call within the switch, it must establish a device or call "monitor" to keep track of the real-time state information for the call or device.

**⇒ NOTE:**

Applications should always be "event driven" and use events received from the Telephony Server to react to changes in call or connection state rather than using a specific switch implementation's call state mode. Following this

guideline will simplify the support of applications across various switch implementations of TSAPI.

An application calls the **cstaMonitorDevice()**, **cstaMonitorCall()**, or **cstaMonitorCallsViaDevice()** function to initiate event reporting for a specific device or call. Event reporting can be provided for a device, a call, or for calls at a monitored device. An application can request two different types of event monitors using these functions. The monitor types are:

- **Call-type monitor** - call-type monitors provide monitoring (event reporting) for unsolicited events about a specific call from "cradle-to-grave". In other words, a call-type monitor provides events for a specific call regardless of the devices at which the call may appear for the duration of the call.

Using call-type monitoring, an application can determine the current state of the call using the TSAPI events. For example, if a call monitor exists for a specific call and that call transfers or forwards to other devices, the sending device ceases to participate in the call, but event reporting continues (telling the application about the new devices participating in the call). Thus, a call-type monitor will provide call state information as the switch, other applications, and human users interact to route a call.

 **NOTE:**

A switch may assign a new call identifier to a call as it is transferred or conferenced. The new call identifier will be provided in the event report associated with the conference or transfer function being requested by the controller of the call.

- **Device-type monitor** - device-type monitors provide the application with call or connection state information about calls at a specific device (the monitored device). TSAPI reports any events about the calls at the monitored device on a device-type monitor. If a call is transferred, drops, or forwards from the monitored device, TSAPI stops reporting events for that call.

If an application begins monitoring a device when call(s) are already in progress at the monitored device, TSAPI may not provide events for those calls (this is switch specific). TSAPI will provide events for calls that arrive at the device after it sends the confirmation event for the device monitor request.

Each monitor must be either a call monitor, a device monitor, or monitor calls via device. An application may request multiple monitors, with various monitors being of various types. An application must setup multiple monitors if it wants to monitor multiple devices or calls at the same time. The switch may impose limitations on the maximum number of simultaneous monitors which can exist for any given switch, call, or device. TSAPI does not place any such restrictions on the application.

When an application requests a device or call monitor, it can also specify an event filter. An event filter causes TSAPI to discard those events which the application is not interested in. An application may specify a filter when it establishes the monitor for a device or call. An application may also use the **cstaChangeMonitorFilter()** to change the filter after the monitor is active.

To receive events, an application must have an active ACS Stream and an implement an event handling mechanism. Further, the reception of unsolicited events requires an active monitor. See the **Control Services** and **Status Reporting Services** sections for more information on events.

## **cstaMonitorDevice()**

---

The Monitor Start service is used to initiate unsolicited event reporting for a device type monitoring on a device object. The unsolicited event reports will be provided for all endpoints within a CSTA switching sub-domain and optionally for endpoints outside of the CSTA switching sub-domain (implementation specific) which are involved with a monitored device.

### **Syntax**

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

RetCode_t cstaMonitorDevice (
    ACSHandle_t      acsHandle,
    InvokeID_t       invokeID,
    DeviceID_t       *deviceID,
    CSTAMonitorFilter_t *monitorFilter,
    PrivateData_t    *privateData),
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***deviceID***

Device ID of the device to be monitored.

#### ***monitorFilter***

This parameter is used to specify a filter type to be used with the object being monitored. Setting a bit to **true** in the *monitorFilter* structure causes the specific event to be **filtered out**, so the application will never see this event. Initialize the structure to all 0's to receive all types of monitor events. See *cstaMonitorDeviceConfEvent* for a definition of a *monitorFilter* structure.

***privateData***

Private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

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

The following are possible negative error conditions for this function:

#### **ACSERR\_BADHDL**

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

This function is used to start a device monitor on a CSTA device . The confirmation event for this function, i.e. **CSTAMonitorConfEvent** will provide the application with the CSTA association handle to the monitored device or call, i.e., the Monitor Cross-Reference Identifier (**monitorCrossRefID**) which defines the CSTA association on which the monitor will exist.

## **cstaMonitorCall()**

---

The Monitor Start service is used to initiate unsolicited event reporting for a call type monitoring on a call object. The unsolicited event reports will be provided for all endpoints within a CSTA switching sub-domain and optionally for endpoints outside of the CSTA switching sub-domain (implementation specific) which are involved with a monitored device.

### **Syntax**

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

RetCode_t cstaMonitorCall (
    ACSHandle_t      acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t   *call,
    CSTAMonitorFilter_t *monitorFilter,
    PrivateData_t    *privateData),
```

### **Parameters**

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***call***

Connection ID of the call to be monitored.

#### ***monitorFilter***

This parameter is used to specify a filter type to be used with the object being monitored. Setting a bit to **true** in the **monitorFilter** structure causes the specific event to be **filtered out**, so the application will never see this event. Initialize the structure to all 0's to receive all types of monitor events. See **CSTAMonitorConfEvent** for a definition of a monitorFilter structure.

#### ***privateData***

Private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

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

The following are possible negative error conditions for this function:

#### **ACSERR\_BADHDL**

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

This function is used to start a call monitor on a CSTA device . The confirmation event for this function, i.e. **CSTAMonitorConfEvent** will provide the application with the CSTA association handle to the monitored device or call, i.e., the Monitor Cross-Reference Identifier (**monitorCrossRefID**) which defines the CSTA association on which the monitor will exist.

## cstaMonitorCallsViaDevice()

The Monitor Start service is used to initiate unsolicited event reporting for a call type monitoring on a device object. The unsolicited event reports will be provided for all endpoints within a CSTA switching sub-domain and optionally for endpoints outside of the CSTA switching sub-domain (implementation specific) which are involved with a monitored device.

### Syntax

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

RetCode_t cstaMonitorCallsViaDevice (
    ACSHandle_t      acsHandle,
    InvokeID_t      invokeID,
    DeviceID_t      *deviceID,
    CSTAMonitorFilter_t *monitorFilter,
    PrivateData_t   *privateData),
```

### Parameters

#### ***acsHandle***

This is the value of the unique handle to the opened 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.

#### ***device***

The deviceID of the device for which call monitoring should be started.

#### ***monitorFilter***

This parameter is used to specify a filter type to be used with the object being monitored. Setting a bit to **true** in the **monitorFilter** structure causes the specific event to be **filtered out**, so the application will never see this event. Initialize the structure to all 0's to receive all types of monitor events. See **cstaMonitorDeviceConfEvent** for a definition of a **monitorFilter** structure.

#### ***privateData***

Private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

### Return Values

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

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

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

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

The following are possible negative error conditions for this function:

#### **ACSERR\_BADHDL**

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

#### **ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### Comments

This function is used to start a monitor on a CSTA object (a device or a call). The confirmation event for this function, i.e. **CSTAMonitorConfEvent** will provide the application with the CSTA association handle to the monitored device or call, i.e. the Monitor Cross-Reference Identifier (**monitorCrossRefID**) which defines the CSTA association on which the monitor will exist. There are two-types of Monitor Service: call-type and device-type.

## CSTAMonitorConfEvent

This event is in response to the **cstaMonitorDevice()**, **cstaMonitorCall** or **cstaMonitorCallsViaDevice** function and contains the association handle being assigned to the CSTA association being used for status reporting.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types and CSTA Data Types** in Section 4 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
            {
                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

typedef unsigned char  CSTAMaintenanceFilter_t;
#define                MF_BACK_IN_SERVICE 0x80
#define                MF_OUT_OF_SERVICE 0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_tcall;
    CSTAFeatureFilter_tfeature;
    CSTAAgentFilter_tagent;
    CSTAMaintenanceFilter_tmaintenance;
    longprivateFilter;
} CSTAMonitorFilter_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 CSTA confirmation event.

### *eventType*

This is a tag with the value **CSTA\_MONITOR\_CONF**, which identifies this message as an **CSTAMonitorConfEvent**.

### *invokeID*

This parameter specifies the requested instance of the function or event. It is used to match a specific functions call request with its confirmation events. Unsolicited events will have this parameter set to zero.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which the requested monitor has been established. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***monitorFilter***

This parameter is used to specify the filter type which is active on the object being monitored by the application. Possible classes of values are: CALL\_FILTER, FEATURE\_FILTER, AGENT\_FILTER, MAINTENANCE\_FILTER, and PRIVATE\_FILTER.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

The application should check this confirmation event to obtain the *monitorCrossRefID* being assigned by the switch and to ensure that the event filter requested has been activated. The event informs the application which filters are active on the given CSTA association.

**cstaMonitorStop()**

The Monitor Stop Service is used to cancel a previously registered Monitor Service on an existing CSTA monitor association, i.e. an active ***monitorCrossRefID***.

**Syntax**

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

RetCode_tcstaMonitorStop (
    ACSHandle_t          acsHandle,
    InvokeID_t          invokeID,
    CSTAMonitorCrossRefID_t monitorCrossRefID,
    PrivateData_t       *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***monitorCrossRefID***

This parameter identifies the original CSTA monitor association for which unsolicited event monitoring is to be canceled. This identifier is provided as a result of a monitor start service request in a **CSTAMonitorConfEvent** for a call or device monitor within the switching domain.

### ***privateData***

Private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_STREAM\_FAILED***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

## Comments

**cstaMonitorStop( )** cancels a previously registered monitor association on a CSTA object (a device or a call object). Once a confirmation event is issued for this function, i.e. a **CSTAMonitorStopConfEvent**; TSAPI terminates the previously active monitoring association and thus end event reporting for the monitored call or device.

## CSTAMonitorStopConfEvent

This event is in response to the **cstaMonitorStop( )** function and provides the application with a confirmation that the monitor association has been canceled. Once this confirmation event is issued all event reporting for the specific monitoring association will be discontinued.

## Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types and CSTA Data Types** in Section 4 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
            {
                CSTAMonitorStopConfEvent
                monitorStop;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_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 CSTA confirmation event.

### ***eventType***

This is a tag with the value **CSTA\_MONITOR\_STOP\_CONF**, which identifies this message as an **CSTAMonitorStopConfEvent**.

### ***invokeID***

This parameter specifies the requested instance of the function or event. It is used to match a specific functions call request with its confirmation events. Unsolicited events will have this parameter set to zero.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This confirmation event indicates a cancellation of a CSTA monitoring association. After this event is issued by the Telephony Server, no further events will be sent to the application on the monitoring association (***monitorCrossRefID***) which was canceled.

## cstaChangeMonitorFilter()

This function is used to request a change in the filter options for CSTA event reporting for a specific CSTA association. It allows the application to specify for which event category the application wishes to receive events.

### Syntax

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

RetCode_t cstaChangeMonitorFilter (
    ACSHandle_t          acsHandle,
    InvokeID_t          invokeID,
    CSTAMonitorCrossRefID_t monitorCrossRefID,
    CSTAMonitorFilter_t *filterlist,
    PrivateData_t       *privateData);
```

## Parameters

### ***acsHandle***

This is the value of the unique handle to the opened 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.

### ***monitorCrossRefID***

This parameter identifies the CSTA association (association handle) for which a change in event filtering is required. The association identifier is provided by the server/switch when the association is established.

### ***filterlist***

This parameter identifies the filter type being requested. Possible classes of values are CALL\_FILTER, FEATURE\_FILTER, AGENT\_FILTER, MAINTENANCE\_FILTER, and PRIVATE\_FILTER. This parameter also identifies the events to be filtered.

### ***privateData***

Private data extension mechanism. Setting this parameter is optional. If the parameter is not used, the pointer should be set to NULL.

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

An application uses **cstaChangeMonitorFilter()** to inform the API Client Library and the server that application wishes to receive only certain events. The server filters out all other events.

**CSTAChangeMonitorFilterConfEvent**

This event occurs as a result of the **cstaChangeMonitorFilter()** function and informs the application which event filter was set by the server.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types and CSTA Data Types** in Section 4 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
            {
                CSTAChangeMonitorFilterConfEvent_t
                changeMonitorFilter;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAChangeMonitorFilterConfEvent_t
{
    CSTAMonitorFilter_t monitorFilter;
} CSTAChangeMonitorFilterConfEvent_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 CSTAConfirmation event.

### ***eventType***

This is a tag with the value **CSTA\_CHANGE\_MONITOR\_FILTER\_CONF**, which identifies this message as an **CSTACHangeMonitorFilterConfEvent**.

### ***invokeID***

This parameter specifies the requested instance of the function or event. It is used to match a specific function's call request with its confirmation events. Unsolicited events will have this parameter set to zero.

### ***monitorFilter***

This parameter identifies the filter type being requested. Possible classes of values are CALL\_FILTER, FEATURE\_FILTER, AGENT\_FILTER, MAINTENANCE\_FILTER, and PRIVATE\_FILTER.

This parameter also identifies the events to be filtered.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This confirmation event should be checked by the application to insure that the event filter requested has been activated and which filters are already active on the given CSTA association.

## CSTAMonitorEndedEvent

This unsolicited indication is sent by the driver/switch to indicate to the application that the monitor associated with the **monitorCrossRefID** has been stopped.

## Syntax

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

```

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAMonitorEnded_t monitorEnded;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAMonitorEndedEvent_t {
    CSTAEventCause_t cause;
} CSTAMonitorEndedEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_MONITOR\_ENDED**, which identifies this message as an **CSTAMonitorEndedEvent**.

#### ***monitorCrossRefID***,

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***cause***

The cause code indicating the reason the monitor was stopped.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### **Comments**

This event is provided by the driver/switch when it can no longer provide the requested events associated with the monitorCrossRefId.

## **Call Event Reports (Unsolicited)**

This section defines the unsolicited TSAPI Event Reports that result from call activity at the Device or the switch. These events provide an application with call status information. Applications, users, and switch administrators may also use switch features that interact with monitored devices and calls, resulting in additional call events. One example of such a feature is call coverage paths.

### **CSTACallClearedEvent**

This event report indicates when a call is torn down. This can occur when the last device has disconnected from the call or when a call is dissolved by another party to the call - like a conference call being dissolved by the conference controller.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t    monitorCrossRefID;
            union
            {
                CSTACallClearedEvent    callCleared;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

typedef enum CSTAEventCause_t {
    ACTIVE_MONITOR = 1,
    ALTERNATE = 2,
    BUSY = 3,
    CALL_BACK = 4,
    CALL_CANCELLED = 5,
    CALL_FORWARD_ALWAYS = 6,
    CALL_FORWARD_BUSY = 7,
    CALL_FORWARD_NO_ANSWER = 8,
    CALL_FORWARD = 9,
    CALL_NOT_ANSWERED = 10,
    CALL_PICKUP = 11,
    CAMP_ON = 12,
    DEST_NOT_OBTAINABLE = 13,
    DO_NOT_DISTURB = 14,
    INCOMPATIBLE_DESTINATION = 15,
    INVALID_ACCOUNT_CODE = 16,
    KEY_CONFERENCE = 17,
    LOCKOUT = 18,
    MAINTENANCE = 19,
    NETWORK_CONGESTION = 20,
    NETWORK_NOT_OBTAINABLE = 21,
    NEW_CALL = 22,
    NO_AVAILABLE_AGENTS = 23,
    OVERRIDE = 24,
    PARK = 25,
    OVERFLOW = 26,
    RECALL = 27,

```

```
    REDIRECTED = 28,  
    REORDER_TONE = 29,  
    RESOURCES_NOT_AVAILABLE = 30,  
    SILENT_MONITOR = 31,  
    TRANSFER = 32,  
    TRUNKS_BUSY = 33,  
    VOICE_UNIT_INITIATOR = 34  
} CSTAEventCause_t;  
  
typedef struct  
{  
    ConnectionID_tclearedCall;  
    LocalConnectionState_tlocalConnectionInfo;  
    CSTAEventCause_tcause;  
} CSTACallClearedEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_CALL\_CLEARED**, which identifies this message as an **CSTACallClearedEvent**.

#### ***monitorCrossRefID***,

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***clearedCall***

This parameter identifies the call which has been cleared.

#### ***localConnectionInfo***

This parameter defines the local connection state of the call after it has been cleared. This could be null, initiated, alerting, connected, held, queued, or failed.

#### ***cause***

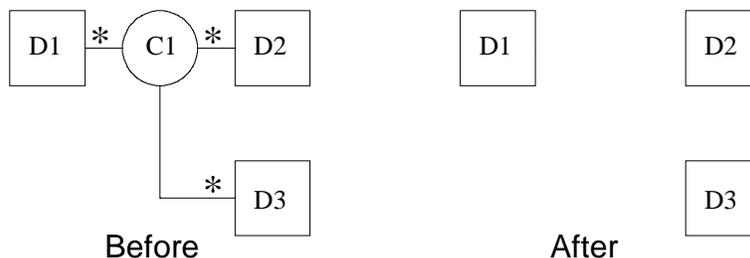
This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is usually provided after the **cstaClearCall()** function has been called by the application. It can also occur, unsolicited, when another endpoint (device) clears a call and the device being monitored by the API is part of the call cleared by the another endpoint. The event is also generated when the last remaining device has disconnected from the call.



**Figure 6-1. Call Cleared Event Report**

### CSTAConferencedEvent

This event report provides indication that two separate calls have been conferenced (merged) into a single call. This occurs without either party being removed from the resulting call.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See *Data Types and CSTA Data Types* in Section 4 for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAConferencedEvent_t conferenced;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t primaryOldCall;
    ConnectionID_t secondaryOldCall;
    SubjectDeviceID_t confController;
    SubjectDeviceID_t addedParty;
    ConnectionList_t conferenceConnections;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAConferencedEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_CONFERENCED**, which identifies this message as an **CSTAConferencedEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***primaryOldCall***

This parameter identifies the primary known call to be conferenced. This is usually the held call pending the conference.

***secondaryOldCall***

This parameter identifies the secondary call (e.g. the consultative call) which is to be conferenced. This is usually the active call which is to be conferenced to the held call pending the conference.

***confController***

This structure identifies the device which is controlling the conference. This is the device which setup the conference. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***addedParty***

This parameter identifies the device which is being added to the conference. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***conferenceConnections***

This is a list of connections (parties) on the call which resulted from the conference. The call ID may be different from either the primary or secondary old call (or both).

***localConnectionInfo***

This parameter defines the local connection state of the call after it has been conferenced. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event provides information regarding a conference after it has been requested by the application using the **CSTAConferenceCall()** function or other endpoints on the switch. The changes in the call states are as follows:

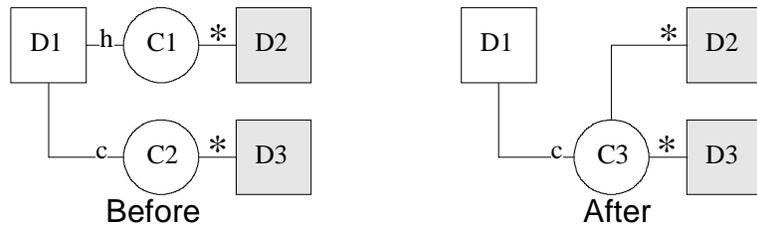


Figure 6-2. Conferenced Event Report

### CSTAConnectionClearedEvent

This event report indicates that a device associated with a call disconnects from the call or is dropped from the call. The event does not indicate that a transferring device has left a call through the act of transferring that call.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *Data Types and CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAConnectionClearedEvent_t connectionCleared;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```

```
typedef struct
{
    ConnectionID_t          droppedConnection;
    SubjectDeviceID_t      releasingDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t       cause;
} CSTAConnectionClearedEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS StreameventClass

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_CONNECTION\_CLEARED**, which identifies this message as an **CSTAConnectionClearedEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***droppedConnection***

This parameter identifies the Connection which was dropped from the call as a result of a device dropping from the call.

### ***releasingDevice***

This parameter identifies the device which dropped the call.

### ***localConnectionInfo***

This parameter defines the local connection state of the call after the connection has been cleared. This could be null, initiated, alerting, connected, held, queued, or failed.

### ***cause***

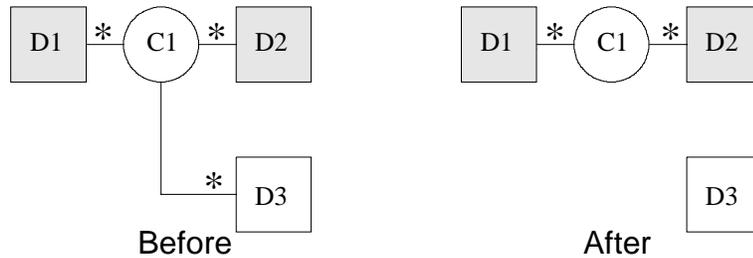
This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the *acsGetEventBlock()* or *acsGetEventPoll()* function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is used to determine which device disconnects from a multi-party call. The deviceID identifies the devices which disconnected or was disconnected from the call. The LocalConnectionInfo defines the state of the call at the monitored device after the device has been dropped from the call.




---

**Figure 6-3. Connection Cleared Event Report**

### CSTADeliveredEvent

This event report indicates that a call is alerting (e.g. ringing) at a specific device or that the server has detected that a call is alerting at a specific device.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See Chapter 4 **Data Types** and **CSTA Data Types** for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTADeliveredEvent_t delivered;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t          connection;
    SubjectDeviceID_t      alertingDevice;
    CallingDeviceID_t      callingDevice;
    CalledDeviceID_t       calledDevice;
    RedirectionDevice_t    lastRedirectionDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t       cause;
} CSTADeliveredEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_DELIVERED**, which identifies this message as an **CSTADeliveredEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***connection***

This parameter identifies the Connection which is alerting

#### ***alertingDevice***

This parameter indicates which device is alerting. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***callingDevice***

This parameter identifies the calling device. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required

***calledDevice***

This parameter identifies the originally called device. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required

***lastRedirectionDevice***

This parameter will identify the previously alerted device in cases where the call was redirected or diverted to the alerting device. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the call after the Connection has alerted. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event provides all the necessary information required when a new call arrives at a device. This will include the calling and called numbers.




---

**Figure 6-4. Delivered Event Report**

## **CSTADivertedEvent**

---

This event report identifies a call which has been deflected or diverted from a monitored device. The call is no longer present or associated with the device.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See *Data Types and CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTADivertedEvent_t diverted;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t connection;
    SubjectDeviceID_t divertingDevice;
    CalledDeviceID_t newDestination;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTADivertedEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

***eventType***

This is a tag with the value **CSTA\_DIVERTED**, which identifies this message as an **CSTADivertedEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***connection***

This parameter indicates the Connection which was previously alerting. This can be the intended Connection for the call before it was diverted.

***divertingDevice***

This parameter indicates the device from which the call was diverted. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***newDestination***

This parameter indicates the device to which the call was diverted. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the device being monitored. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is used to determine information about a call which has been diverted from a monitored device. This includes information on which device the call is being diverted.

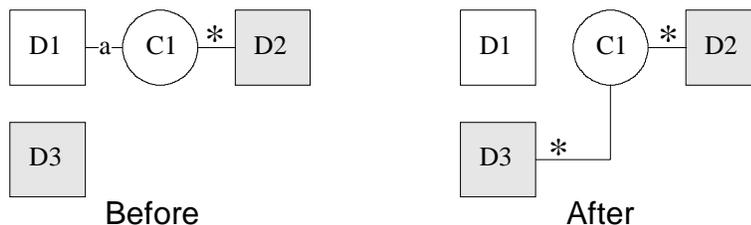


Figure 6-5. Diverted Event Report

### CSTAEstablishedEvent

This event report indicates that a device connects to a call.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *Data Types and CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAEstablishedEvent_t established;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```

```
typedef struct
{
    ConnectionID_t          establishedConnection;
    SubjectDeviceID_t      answeringDevice;
    CallingDeviceID_t      callingDevice;
    CalledDeviceID_t       calledDevice;
    RedirectionDeviceID_t  lastRedirectionDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t       cause;
} CSTAEstablishedEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_ESTABLISHED**, which identifies this message as an **CSTAEstablishedEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***establishedConnection***

This parameter identifies the Connection which joined the call as a result of answering the call.

#### ***answeringDevice***

This parameter indicates the device which has joined the call, i.e. the answering device. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***callingDevice***

This indicates which device made the call, i.e. the calling device. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***calledDevice***

This parameter indicates the originally called device. This may not always be the device answering a call as is the case with call forwarding or coverage, i.e. call redirection. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***lastRedirectionDevice***

This parameter indicates the previously alerted device in cases where a call is redirected. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the device for the call which has been established. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event is typically used to determined when a call is answered by an endpoint being called by the application. This includes the calling and called number identification.




---

**Figure 6-6. Established Event Report**

**CSTAFailedEvent**

---

This event report indicates that a call cannot be completed. The event applies only to a single Connection.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

```

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAFailedEvent_t failed;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t    failedConnection;
    SubjectDeviceID_t failingDevice;
    CalledDeviceID_t  calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t  cause;
} CSTAFailedEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_FAILED**, which identifies this message as an **CSTAFailedEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***failedConnection***

This parameter indicates which Connection has failed.

***failingDevice***

This parameter indicates which device has failed. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***calledDevice***

This parameter indicates which device was called when the call failed. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the call after the Connection has failed. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event occurs anytime a call cannot be completed for any reason (e.g. Stations Busy, Reorder Tone, Trunks Busy, etc.). The **cause** parameter contains the reason why the call failed.




---

**Figure 6-7. Failed Event Report**

**CSTAHeldEvent**

---

This event report indicates that the server has detected that communications on a particular Connection has be interrupted (i.e. put on hold) by one of the devices on the call. This event is usually associated with a call being placed on hold at a device.

## Syntax

The following structure shows only the relevant portions of the unions for this message. See **Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAHeldEvent_t held;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t    heldConnection;
    SubjectDeviceID_t holdingDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAHeldEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_HELD**, which identifies this message as an **CSTAHeldEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***heldConnection***

This parameter identifies the Connection which was put on hold by the device.

***holdingDevice***

This parameter identifies the device which placed the connection on hold. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the call after the Connection has been put on hold. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event occurs after a call has been placed on hold at a specific device. This informs the application what device placed the connection on hold.




---

**Figure 6-8. Held Event Report**

## **CSTANetworkReachedEvent**

---

This event report informs the application that a call has left the switch on an outbound trunk and is being routed through the telephone network.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See *Data Types* and *CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTANetworkReachedEvent_t networkReached;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t      connection;
    SubjectDeviceID_t   trunkUsed;
    CalledDeviceID_t    calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t    cause;
} CSTANetworkReachedEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

***eventType***

This is a tag with the value **CSTA\_NETWORK\_REACHED**, which identifies this message as an **CSTANetworkReachedEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***connection***

This parameter specifies the Connection ID for the outbound connection associated with the trunk and its connection to the network (see figure below).

***trunkUsed***

This parameter specifies the trunk that was used to establish the Connection with the telephone network. If the device (i.e. the trunk) is not specified, then the parameter will indicate that the device was not known or that it was not required.

***calledDevice***

This parameter indicates the destination device for the call. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***localConnectionInfo***

This parameter defines the local connection state of the call after the Connection has cut-through into the telephone network. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

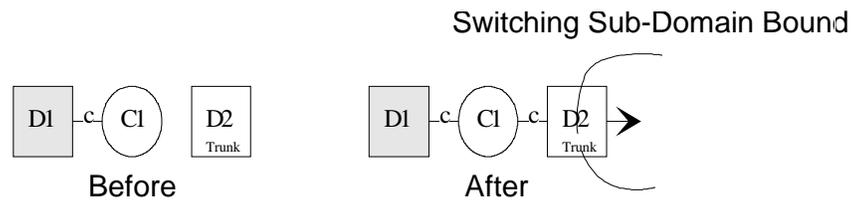
***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

Once this event occurs the level of call related status information may decrease depending on the type of trunk being used to route the call to its destination across the telephone network. The amount of call related status information provided by the network will depend on the type of trunk and telephone network

being used to complete the call. Call status information may be limited to the disconnect or drop event. This only applies for calls to other network endpoints and not to calls within the switch being controlled by the server.



**Figure 6-9. Networked Reached Event Report**

### **CSTAOriginatedEvent**

This event report informs the application that the switch is attempting to establish a call as a result of a completed request from the application.

#### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See *Data Types* and *CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAOriginatedEvent_t originated;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```

```
typedef struct
{
    ConnectionID_t          originatedConnection;
    SubjectDeviceID_t      callingDevice;
    CalledDeviceID_t       calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t       cause;
} CSTAOriginatedEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_ORGINATED**, which identifies this message as an **CSTAOriginatedEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***originatedConnection***

This parameter identifies the Connection where a call has been originated.

### ***callingDevice***

This parameter identifies the device from which the call has been originated. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***calledDevice***

This parameter identifies the device for which the originated call is intended. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***localConnectionInfo***

This parameter defines the local connection state of the call after the Connection has been originated. This could be null, initiated, alerting, connected, held, queued, or failed.

### ***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event indicates that a call is being launched by the switch on behalf of the request from the application. The event only indicates that the switch is attempting to make the call. The application should check for additional events to determine the status of the call as it proceeds either through the switch or out to the telephone network.




---

**Figure 6-10. Originated Event Report**

**CSTAQueuedEvent**

This event report indicates that a call has been queued to an ACD Split, a hunt group, or others devices which support call queues. Call can also be queued during network re-routing without specifying a device.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAQueuedEvent_t queued;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t          queuedConnection;
    SubjectDeviceID_t      queue;
    SubjectDeviceID_t      callingDevice;
    CalledDeviceID_t       calledDevice;
    RedirectionDeviceID_t  lastRedirectionDevice;
    int                    numberQueued;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t       cause;
} CSTAQueuedEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_QUEUED**, which identifies this message as an **CSTAQueuedEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***queuedConnection***

This indicates the Connection was queued to the device.

***queue***

This parameter specifies the device to which the call has been queued. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***callingDevice***

This parameter indicates the device who queued the call. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***calledDevice***

This parameter indicates the device which was called (the intended recipient of the call). If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***lastRedirectionDevice***

This parameter identifies the last device which redirected the call, if the call has been redirected. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***numberQueued***

This parameter indicates how many calls are queued to the queuing device.

***localConnectionInfo***

This parameter defines the local connection state of the call after the call has been queued. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the privateData pointer in the **acsGetEventBlock( )** or **acsGetEventPoll( )** function. If the privateData pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event usually occurs when an application is monitoring a call, a Vector Directory Number (VDN), an ACD Split, or a hunt group. The event also provides information pertaining to the number of calls that have been queued to a device. This information can be useful to applications managing the queue at the device.



**Figure 6-11. Queued Event Report**

## **CSTARRetrieveEvent**

This event report identifies a call which was previously on hold and has been retrieved at a device. This is equivalent to taking the call off the hold state and into the active state.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **Data Types** and **CSTA Data Types** in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTARetrievedEvent_t retrieved;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_RETRIEVED**, which identifies this message as an **CSTARetrievedEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***retrievedConnection***

This parameter specifies the Connection for which the call has been taken off the hold state.

### ***retrievingDevice***

This specifies the device which de-activated the call from the hold state.

### ***localConnectionInfo***

This parameter defines the local connection state of the call after the call has been retrieved from the hold state. This could be null, initiated, alerting, connected, held, queued, or failed.

### ***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This event informs the application that a call is no longer on hold. This can occur if the end-user physically takes the call off the hold state or in response to the **csaRetrieveCall()** function request.



**Figure 6-12. Retrieved Event Report**

### CSTAServiceInitiatedEvent

This event report indicates to the application that telephony service was initiated at a device. The switch sends this event when it provides “dial tone”. Note that the user may be going off hook to invoke a feature using a feature access code, so call setup events do not necessarily follow.

#### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See *Data Types* and *CSTA Data Types* in Section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_tmonitorCrossRefID;
            union
            {
                CSTAServiceInitiatedEvent_t serviceInitiated;
            }u ;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_SERVICE\_INITIATED**, which identifies this message as an **CSTAServiceInitiatedEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***initiatedConnection***

This parameter indicates the Connection for which service (dial tone) has been established or a feature is invoked. The same Connection identifier will continue to be used if a call is eventually established by the device.

### ***localConnectionInfo***

This parameter defines the local connection state of the call after the service has been initiated. This could be null, initiated, alerting, connected, held, queued, or failed.

### ***cause***

This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

CSTA-179 states that switches are not required to send the service initiated event for calls that are originated from functional type devices (e.g. ISDN BRI devices that do *en bloc* dialing) or for calls that other applications originate using **cstaMakeCall()**. Thus, some PBX drivers may not provide this event in these circumstances.



**Figure 6-13. Service Initiated Event Report**

### CSTATransferredEvent

This event report indicates that an existing call was transferred to another device and that the device which transferred the call is no longer part of the call, i.e. the transferring device has dropped from the call.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTATransferEvent_t transferred;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```

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

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_TRANSFERRED**, which identifies this message as an **CSTATransferredEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***primaryOldCall***

This parameter identifies the primary known call that was transferred.

#### ***secondaryOldCall***

This parameter identifies the secondary call that was transferred. This would identify the consultative call used to make the transfer, after the primary call was placed on hold.

#### ***transferringDevice***

This indicates which device transferred the call. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***transferredDevice***

This indicates to which device the call was transferred. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***transferredConnections***

This is a list of connections (parties) on the call which resulted from the transfer. The call ID may be different from either the primary or secondary old call (or both).

***localConnectionInfo***

This parameter defines the local connection state of the call after the calls have been transferred from the device which performed the transfer. This could be null, initiated, alerting, connected, held, queued, or failed.

***cause***

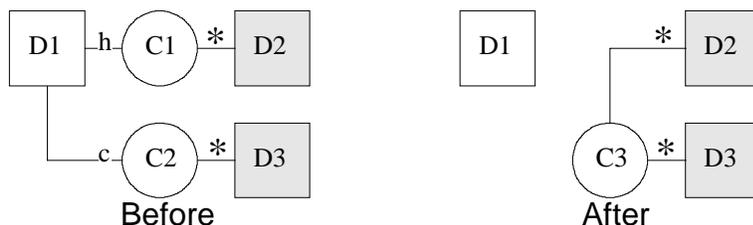
This parameter contains the cause value which indicates the reason or explanation for the occurrence of this event. The possible events are defined by **CSTAEventCause\_t**.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event provides the application with all the information it needs regarding a call which was transferred from one device to another.



**Figure 6-14. Transferred Event report**

## Feature Event Reports (Unsolicited)

TSAPI feature event reports indicate a change in the state of a specific feature operating on a call or a device on the switch. Each feature event gives the current state of the feature regardless of what the state of the feature was before an application receives a feature event.

### CSTACallInfoEvent

This event report is provided when a user account code feature has collected data for a party on the call. The event includes the account code and authorization information which was collected by the switch feature.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *Data Types* and *CSTA Data Types* in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTACallInfoEvent_t callInformation;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ConnectionID_t    connection;
    SubjectDeviceID_t device;
    AccountInfo_t     accountInfo;
    AuthCode_t        authorisationCode;
} CSTACallInfoEvent_t;
```

```
typedef char          AccountInfo_t[32];

typedef char          AuthCode_t[32];
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_CALL\_INFORMATION**, which identifies this message as an **CSTACallInfoEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***connection***

This parameter identifies the party that has entered the account code.

### ***device***

Indicates from which device was the account code information entered. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***accountInfo***

Specifies the account code which was entered at the device.

### ***authorizationCode***

Specifies the authorization code which was entered at the device.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This event informs the application when an account code feature has been activated and what information was collected by the switch as a result of the feature being activated.

## **CSTADoNotDisturbEvent**

---

This event report indicates a change in the status of the Do Not Disturb feature for a specific device. The Do Not Disturb event will result in all calls to a device to be automatically forwarded to the device coverage path.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTADoNotDisturbEvent_t doNotDisturb,
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_t device;
    Boolean            doNotDisturbOn;
} CSTADoNotDisturbEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

***eventType***

This is a tag with the value **CSTA\_DO\_NOT\_DISTURB**, which identifies this message as an **CSTADoNotDisturbEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***device***

Specifies the device for which the DO Not Disturb feature has been activated/deactivated. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***doNotDisturbON***

Specifies whether the DO Not Disturb feature is on (1) or off (0).

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**CSTAForwardingEvent**

This event report will indicate a change in the state of the Forwarding feature for a specific device. The event will also indicate the type of forwarding being invoked when the feature is activated.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See ACS Data Types and CSTA Data Types in section 4 for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_tmonitorCrossRefID;
            union
            {
                CSTAForwardingEvent_t forwarding;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_tdevice;
    ForwardingInfo_tforwardingInformation;
} CSTAForwardingEvent_t;

typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0,
    FWD_BUSY = 1,
    FWD_NO_ANS = 2,
    FWD_BUSY_INT = 3,
    FWD_BUSY_EXT = 4,
    FWD_NO_ANS_INT = 5,
    FWD_NO_ANS_EXT = 6
} ForwardingType_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t forwardingType;
    Boolean forwardingOn;
    DeviceID_t forwardDN; /* NULL for not present */
} ForwardingInfo_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_FORWARDING** which identifies this message as an **CSTAForwardingEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***device***

Specifies the device for which the Forwarding feature has been activated/deactivated. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***forwardingType***

Specifies the type of forwarding being invoked for the specific device. This may include one of the following:

<b>Immediate</b>	Forwarding all calls
<b>Busy</b>	Forwarding when busy
<b>No Answer</b>	Forwarding after no answer
<b>Busy Internal</b>	Forwarding when busy for an internal call
<b>Busy External</b>	Forwarding when busy for an external call
<b>No Answer Internal</b>	Forwarding after no answer for an internal call
<b>No Answer External</b>	Forwarding after no answer for an external call.

***forwardingON***

Specifies whether the Forward feature is on (1) or off (0).

***forwardDN***

Specifies the destination device to which the calls are being forwarded. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

The application should be aware that the *forwardingInfo* parameter can indicate any of the defined values depending on the switch implementation of the forwarding feature.

## CSTAMessageWaitingEvent

This event report is used to indicate whether the Message Waiting feature has been activated/deactivated.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAMessageWaitingEvent_t messageWaiting;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    CalledDeviceID_t    deviceForMessage;
    SubjectDeviceID_t   invokingDevice;
    Boolean              messageWaitingOn;
} CSTAMessageWaitingEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_MESSAGE\_WAITING** which identifies this message as an **CSTAMessageWaitingEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***deviceForMessage***

Indicates the device where the message is waiting (i.e. address of device where the message waiting feature was activated). If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***invokingDevice***

Specifies which device invoked the message waiting feature (i.e. address of the device who activated the message waiting feature). If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***messageWaitingOn***

Specifies whether the Message Waiting feature is on (1) or off (0).

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This event can occur for both a device or a call association.

## Agent Status Event Reports (Unsolicited)

---

This section covers event reports which pertain to the use of ACD agent features. The agent feature event reports indicate a change in the state of a specific agent. Each event defines the current state of the agent feature regardless of the state of the feature before the event. Typically, applications in the call center or message center environment use agent status event reports.

### CSTALoggedOnEvent

---

This event report informs the application that an agent has logged into a device (usually an ACD Split).

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTALoggedOnEvent_t loggedOn,
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
    AgentGroup_t       agentGroup;
    AgentPassword_t    password;
} CSTALoggedOnEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_LOGGED\_ON** which identifies this message as an **CSTALoggedOnEvent**.

### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

### ***agentDevice***

Specifies the device from which the agent is logged on to the system. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

### ***agentID***

This parameter specifies the agent identifier of the agent who logged into the system.

### ***agentGroup***

Specifies the group or ACD Split to which the agent is logging into.

### ***password***

This parameter specifies the agent's password used to log into the system.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

In most cases, when an agent logs into a device it usually means that the agent is ready to start receiving calls at the device. This may not be true for some implementations.

## **CSTALoggedOffEvent**

---

This event report indicates that an agent has logged out of the device/ACD Split for which the agent had previously logged in and was providing service.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTALoggedOffEvent_t loggedOff;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_t agentDevice;
    AgentID_t         agentID;
    AgentGroup_t      agentGroup;
} CSTALoggedOffEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_LOGGED\_OFF** which identifies this message as an **CSTALoggedOffEvent**.

***agentDevice***

Specifies the device from which the agent is logged off the system. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***agentID***

This parameter specifies the agent identifier of the agent who logged off the system.

***agentGroup***

Specifies the group or ACD Split from which the agent is logging out.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**CSTANotReadyEvent**

This event report indicates that an agent is busy with tasks other than servicing an ACD call at the device. In most cases this will imply that the agent is not ready to receive a call or that the agent is taking a break.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

```

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTANotReadyEvent_t notReady;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_NOT\_READY** which identifies this message as an **CSTANotReadyEvent**.

#### ***agentDevice***

Specifies the device from which the agent is logged on to the system. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***agentID***

This parameter specifies the identifier of the agent who is not ready to receive calls.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **CSTARReadyEvent**

---

This event report indicates that an agent is ready to receive calls at the device. This event can occur even if the agent is busy on an active call at the device.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See *Data Types* and *CSTA Data Types* in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTARReadyEvent_t ready;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
} CSTARReadyEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_READY** which identifies this message as an **CSTARReadyEvent**.

***agentDevice***

Specifies the device which is ready to receive calls from the ACD. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***agentID***

This parameter specifies the identifier of the agent who is ready to receive calls.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**CSTAWorkNotReadyEvent**

This event report indicates that the agent is in after call work mode completing the tasks involved in servicing a call after the connection has been disconnected. This will imply that the agent is no longer on the call but is completing the servicing of the last call and the agent **should not** receive any additional calls.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_tmonitorCrossRefID;
            union
            {
                CSTAWorkNotReadyEvent_t workNotReady;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```



```
typedef struct
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkNotReadyEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_WORK\_NOT\_READY** which identifies this message as an **CSTAWorkNotReadyEvent**.

#### ***agentDevice***

Specifies the device which has invoked the Work Not Ready mode. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***agentID***

This parameter specifies the identifier of the agent who is in the Work Not Ready mode.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

In the case of this event the agent is still working on completing the after call work for the last call. The difference between this event and the **CSTAWorkReadyEvent** is that the agent has indicated that he/she is not ready to receive additional calls.

## CSTAWorkReadyEvent

This event report indicates that the agent is in "after call work mode" completing the tasks involved in servicing a call after the connection has been disconnected. This implies that the agents is no longer on the call but is completing the servicing of the last call and the agent **may** receive any additional calls.

## Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAWorkReadyEvent_t workReady;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

## Parameters

### **acsHandle**

This is the handle for the ACS Stream.

### **eventClass**

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

### **eventType**

This is a tag with the value **CSTA\_WORK\_READY** which identifies this message as an **CSTAWorkReadyEvent**.

### **agentDevice**

Specifies the device which has invoked the Work Ready mode. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***agentID***

This parameter specifies the identifier of the agent who is in the Work Ready mode.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

In the case of this event the agent is still working on completing the after call work for the last call. The difference between this event and the **CSTAWorkNotReadyEvent** is that the agent has indicated that he/she is ready to receive additional calls.

## Event Report Data Types (Unsolicited)

This section defines the data structures associated with the CSTA Event Reports defined in the *Status Reporting Services* section of this document.

### CSTAMonitorFilter\_t

This structure is used to identify the event type filters requested or available on a monitored CSTA association.

```

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

typedef unsigned char   CSTAMaintenanceFilter_t;
#define                  MF_BACK_IN_SERVICE 0x80
#define                  MF_OUT_OF_SERVICE 0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t call;
    CSTAFeatureFilter_t feature;
    CSTAAgentFilter_t agent;
    CSTAMaintenanceFilter_t maintenance;
    Boolean          private;
} CSTAMonitorFilter_t;

```

### ***CALL\_FILTERS***

These values indicate that a call event filter should be used for processing events. The provided filter may be different than the one requested.

### ***FEATURE\_FILTERS***

These values indicate that a feature event filter should be used for processing events. The provided filter may be different than the one requested.

### ***AGENT\_FILTERS***

These values indicate that an agent event filter should be used for processing events. The provided filter may be different than the one requested.

### ***MAINTENANCE\_FILTERS***

These values indicate that a maintenance event filter should be used for processing events. The provided filter may be different than the one requested.

### ***PRIVATE\_FILTER***

This value indicates that a private filter should be used for processing events. The provided filter may be different than the one requested.

## **CSTAEventCause\_t**

This structure contains an enumerated list of all the possible event causes which can occur with different events. The definitions of these event cause codes are also provided.

```
typedef enum CSTAEventCause_t {
    EC_NONE = -1,
    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
} CSTAEventCause_t;

```

Certain cause codes will appear in events only if they make sense. The Table 6-1 gives cause code definitions. Table 6-2 illustrates which cause codes are possible for the each of the call events.

Cause Code	Definition
Active Monitor	an Active Monitor Feature has occurred. This feature typically allows intrusion by a supervisor into an agent call with the ability to speak and listen. The resultant call can be considered as a conference so this cause code may be supplied with the Conferenced Event Report.
Alternate	the call is in the process of being exchanged. This feature is typically found on single-line telephones, where the human interface puts one call on hold and retrieves a held call or answers a waiting call in an atomic action.
Busy	the call encountered a busy tone or device
Call Back	Call Back is a feature invoked (by a user or via CSTA) in an attempt to complete a call that has encountered a busy or no answer condition. As a result of invoking the feature, the failed call is cleared and the call can be considered as queued. The switch may subsequently automatically retry the call (normally when the called party next becomes free). Consequently, this cause code may appear in Event Reports related to the feature invocation (Call Cleared, Connection Cleared and Queued) or related to the subsequent, retried call (Service Initiated, Originated, Delivered, and Established).
Call Canceled	the user has terminated a call without going on-hook.
Call Forward	the call has been redirected via a Call Forwarding feature set for general, unknown, or multiple conditions.
Call Fd. - Immediate	the call has been redirected via a Call Forwarding feature set for all conditions.

Cause Code	Definition
Call Fd. - Busy	the call has been redirected via a Call Forwarding feature set for a busy endpoint.
Call Fd. - No Answer	the call has been redirected via a Call Forwarding feature set for an endpoint that does not answer.
Call Not Answered	the call was not answered because a timer has elapsed.
Call Pickup	the call has been redirected via a Call Pickup feature.
Camp On	a Camp On feature has been invoked or has matured.
Dest. Not Obtainable	the call could not obtain the destination.
Do Not Disturb	the call encountered a Do Not Disturb condition.
Incompatible Destination	the call encountered an incompatible destination.
Invalid Account Code	the call has an invalid account code.
Key Operation <sup>1</sup>	indicates that the Event Report occurred at a bridged or twin device.
Lockout	the call encountered inter-digit time-out while dialing.
Maintenance	the call encountered a facility or endpoint in a maintenance condition.
Net Congestion	the call encountered a congested network. In some circumstances this cause code indicates that the user is listening to a "No Circuit" Special Information Tone (SIT) from a network that is accompanied by a statement similar to "All circuits are busy..."
Net Not Obtainable	the call could not reach a destination network.
Resources not Available	resources were not available.
Silent Monitor	the event was caused by the invocation of a feature that allows a third party, such as an ACD agent supervisor, to join the call. The joining party can hear the entire conversation, but cannot be heard by either original party. The feature, sometimes called <i>silent intrusion</i> , 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, announcement) rather than the result of action by a human user.

---

1. Telephone numbers associated primarily with one device often appear also on a second device. One example is a secretary who's phone has mirrored or bridged lines of a boss's phone.

**CSTA Event Report - Cause Relationships**

Cause	Call Clr.	Conf	Con. Clr.	Dlv.	Div.	Est.	Fail	Held	Net. Reh	Orig.	Q-ed	Retr.	Svc. Init.	Tran
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			
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							

**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
Maintenance	y						y							
Net Congestion							y				y			
Net Not Obtainable							y				y			
New Call		y		y		y				y				y
No Available Agents				y	y		y				y			
Overflow	y		y	y	y		y		y		y			
Override	y	y	y	y		y	y			y			y	
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

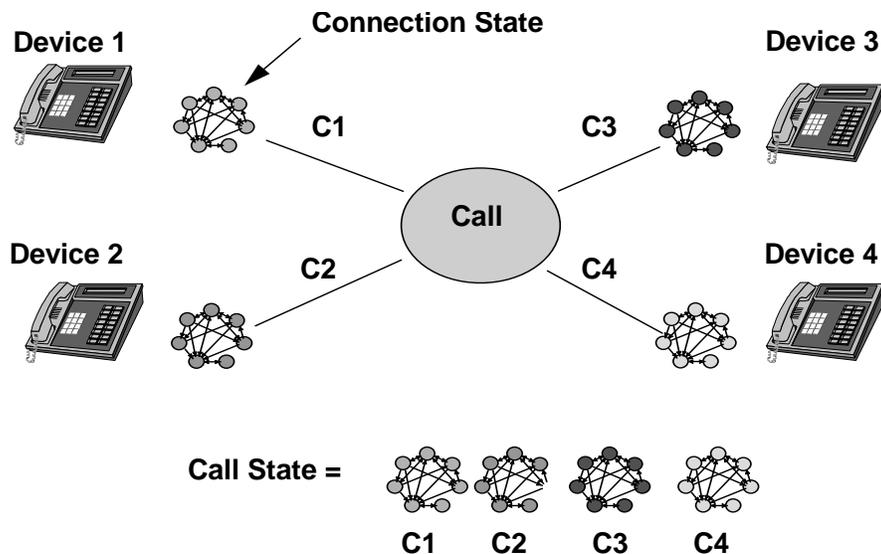


---

An application uses CSTA Snapshot Services to query the current state of a CSTA Call or a Device object. Snapshot services query the switch to provide an application with information about the object. The information is a "*snapshot*" since the state of the Call or Device object changes over time.

The Call Snapshot Services return a list of the Devices and Connections associated with a given Call, and the Connection States for each of those Devices. As Figure 7-1 illustrates, the union of the Connection States for the Call defines the overall Call State. Also refer to the definition of Call State in Chapter 3.

Figure 7-1 shows a Call that has four associated devices. Recall from Chapter 3 that the relationship between a CSTA Call and a CSTA Device is a CSTA Connection (C1, C2, C3, and C4 are Connections). Each Connection has an associated Connection State. The Call Snapshot Services inform an application of each Device that is on a given Call and the associated Connection State for those Devices. The Call State is the union of all the Connection States associated with the Call. The application can use snapshot information to control Connections. For example, if Figure 7-1 shows a four-party conference call, then an application can use the Call Snapshot Services together with the **cstaClearConnection()** service to disconnect any party from the conference. To disconnect connection C4, an application uses the Call Snapshot Services to obtain a Connection Identifier (for C4) that it then passes to the **cstaClearConnection()** service.



**Figure 7-1. Call Snapshot Service**

Device Snapshot Services return information about Calls that are associated with a given CSTA Device object. The information includes a list of Calls associated with the given Device and the Connection State of each Call (at that Device). Note the duality here: Call Snapshot Services return information about Connections at all Devices associated with a given Call, while Device Snapshot Services return information about all Connections at a given Device. Applications use the Device Snapshot Services when they need to know what is happening at a specific Device. As Figure 7-2 shows, Device Snapshot Services do not provide information about the other parties on those Calls connected to the given Device.

An application can use Device Snapshot information to manipulate any Connection, (C1, C2, or C3 in Figure 7-2) at the given Device.

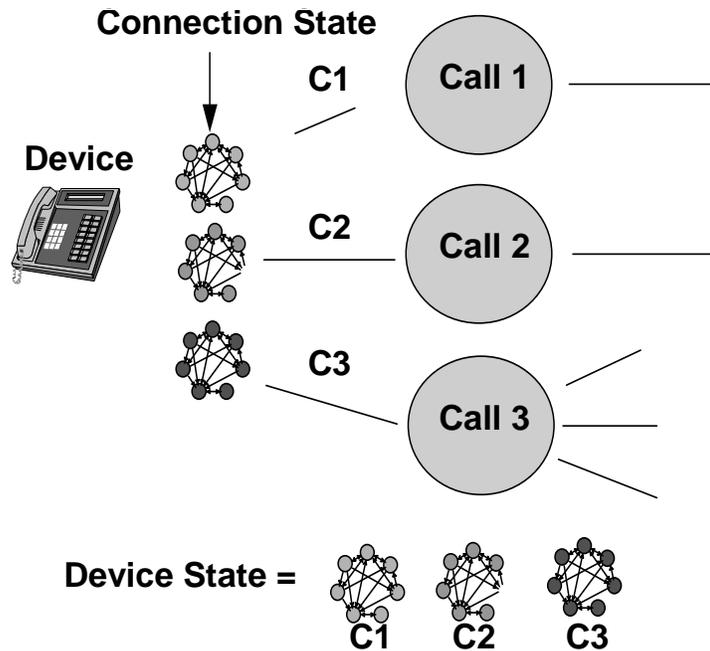


Figure 7-2. Device Snapshot Service

**⇒ NOTE:**

Before an application requests the Call or Device Snapshot Services, it must have previously obtained a Call or Device identifier (that it will use as a parameter to request those services). The identifier specifies a Call or Device in the switching domain. Depending on the implementation, Snapshot Services may not provide information about devices or connections outside of that switching domain (devices not attached to that switch) .

## Call Snapshot Services

This section defines the Call Snapshot Services that query the switch for the status of calls within the switching domain. Call Snapshot Services return information about all Devices and Connections associated with a specified CSTA Call object.

## **cstaSnapshotCallReq()**

---

The **cstaSnapshotCallReq()** service provides information about a Call object in the switching domain. The service will return the Devices associated with a given Call and the Connection State for each Device. The Call State is the union of the Connection States.

### **Syntax**

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

RetCode_t    cstaSnapshotCallReq (
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    ConnectionID_t *snapshotObj,
    PrivateData_t  *privateData),
```

### **Parameters**

#### ***acsHandle***

This is the handle to an active ACS Stream over which the request will be made.

#### ***invokeID***

This is an application provided handle that the application uses to match a specific instance of a service request with its confirmation event. The application supplies this parameter only when the Invoke ID mechanism is set for Application-generated IDs in **acsOpenStream()**. The ACS Library ignores this parameter when the Stream is set for Library-generated invoke IDs.

#### ***snapshotObj***

This is a pointer to the Connection Identifier identifying the Call object for which Snapshot information is requested.

#### ***privateData***

This is an optional pointer to CSTA private data.

### **Return Values**

**cstaSnapshotCallReq()** 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, the invoke identifier. If the call fails it will return a negative error (<0). 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 it will return a negative error (<0). For application-generated identifiers the return is never positive (>0).

An application should always check the **CSTASnapshotCallConfEvent** message to insure that the Telephony Server and switch have acknowledged and processed the **cstaSnapshotCallReq( )** request.

The following are possible negative error conditions for this function:

**ACSERR\_BADHDL**

The application provided a bad or unknown **acsHandle**.

**ACSERR\_STREAM\_FAILED**

A previously active ACS Stream has been abnormally aborted.

### Comments

A call to **cstaSnapshotCallReq( )** results in a confirmation event, **CSTASnapshotCallConfEvent**, that returns the information about the call. **cstaSnapshotCallReq( )** provides information about calls that make further monitoring meaningful. For example, when an application requests **cstaMonitorStart( )**, there may already be active calls at the Device being monitored. The application can use Call Snapshot information to obtain information about those existing calls process additional events about them in a reasonable way.

**cstaSnapshotCallReq( )** is passive and does not affect the state of any object in the switching domain.

## CSTASnapshotCallConfEvent

The Call Snapshot confirmation event returns call related information in response to the **cstaSnapshotCallReq( )** service. The call information includes the static Device Identifiers, the Connection Identifiers, and Connection States for every endpoint in the specified call.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASnapshotCallConfEvent_t snapshotCall;
            } u;
        } cstaConfirmation
    } event;
} CSTAEvent_t;

typedef struct CSTASnapshotCallConfEvent_t {
    CSTASnapshotCallData_t snapshotData;
} CSTASnapshotCallConfEvent_t;

typedef struct CSTASnapshotCallData_t {
    int count;
    struct CSTASnapshotCallResponseInfo_t *info;
} CSTASnapshotCallData_t;

typedef struct CSTASnapshotCallResponseInfo_t {
    SubjectDeviceID_t deviceOnCall;
    ConnectionID_t callIdentifier;
    LocalConnectionState_t localConnectionState;
} CSTASnapshotCallResponseInfoEvent_t;

```

## Parameters

### *acsHandle*

This is the handle for the ACS Stream over which the confirmation arrived. This is the same as the ACS Stream over which the request was made.

### *eventClass*

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

### *eventType*

This is a tag with the value **CSTA\_SNAPSHOT\_CALL\_CONF**, which identifies this message as an CSTASnapshotCallConfEvent.

### *invokeID*

This parameter specifies the service request instance for the **cstaSnapshotCallReq()**. The application uses this parameter to correlate responses with requests.

### *snapshotData*

Contains all the snapshot information for the Call for which the query was made.

**count:** A count of the number of *CSTASnapshotCallResponseInfo\_t* structures. Each structure contains information about one device on the call.

**info:** A pointer to an array of *CSTASnapshotCallResponseInfo\_t* structures, each of which contains the following fields:

**deviceOnCall** — A pointer to the Device Identifier of a device that is a party on the call for which the query was made.

**callIdentifier** — The Connection Identifier for the Connection between the deviceOnCall and the call for which the query was made.

**localConnectionState** — The Connection State for the local Connection in the callIdentifier parameter.

***privateData***

If private data accompanies this event, 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 **CSTASnapshotCallConfEvent** does not deliver private data to the application.

Comments

The **CSTASnapshotCallConfEvent** returns an array or variable length since the number of devices on a call can be greater than one. Each array element identifies a Device on the call, the Connection between the Device and the Call, and the Connection State (see Figure 7-1). An application should use **count** to determine the number of array elements.

## **Device Snapshot Service**

---

This section defines the Device Snapshot Services that query the switch for the status of Devices within the switching domain. Device Snapshot Services return information about Calls (Connections) associated with a specified Device.

### **cstaSnapshotDeviceReq()**

---

The **cstaSnapshotDeviceReq()** service returns information about a Device object in the switching domain. The service returns a list of calls associated with the given Device and the Connection State of each of those calls at that Device.

## Syntax

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

RetCode_t    cstaSnapshotDeviceReq (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    DeviceID_t    *snapshotObj,
    PrivateData_t    *privateData);
```

## Parameters

### *acsHandle*

This is the handle to an active ACS Stream over which the request will be made.

### *invokeID*

This is an application provided handle that the application uses to match a specific instance of a service request with its confirmation event. The application supplies this parameter only when the Invoke ID mechanism is set for Application-generated IDs in **acsOpenStream()**. The ACS Library ignores this parameter when the Stream is set for Library-generated invoke IDs.

### *snapshotObj*

This parameter is a pointer to the Device Identifier for the Device object for which Snapshot information is being requested.

### *privateData*

This is an optional pointer to CSTA private data.

## Return Values

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

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

The application should always check the **CSTASnapshotDeviceConfEvent** message to insure that the Telephony Server and the switch have acknowledged and processed the **cstaSnapshotDeviceReq()** request.

The following are possible negative error conditions for this function:

**ACSERR\_BADHDL**

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

**ACSERR\_STREAM\_FAILED**

This return value indicates that a previously active ACS Stream has been abnormally aborted.

**Comments**

A call to **cstaSnapshotDeviceReq()** results in a confirmation event, **CSTASnapshotDeviceConfEvent**, which returns information about the Device. **cstaSnapshotDeviceReq()** provides information about Devices that permit an application to synchronize state with the switching domain. For example, an application can call **cstaSnapshotDeviceReq()** to find out about the Calls present at a Device, then call **cstaMonitorStart()** to monitor the Device. The information from the Device query permits the application to process the monitoring events in a proper context.

The **cstaSnapshotDeviceReq()** is passive and does not affect the state of any object within the switching domain.

**CSTASnapshotDeviceConfEvent**

The Device Snapshot confirmation event returns Device related information in response to the **cstaSnapshotDeviceReq()** service. The Device information includes a Connection Identifier for each Call at the Device and the Connection State for each Call at the Device.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASnapshotDeviceConfEvent_t snapshotDevice;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASnapshotDeviceConfEvent_t {
    CSTASnapshotDeviceData_t snapshotData;
} CSTASnapshotDeviceConfEvent_t;

typedef struct CSTASnapshotDeviceData_t {
    int    count;
    struct CSTASnapshotDeviceResponseInfo_t *info;
} CSTASnapshotDeviceData_t;

typedef struct CSTASnapshotDeviceResponseInfo_t {
    ConnectionID_t callIdentifier;
    CSTACallState_t localcallState;
} CSTASnapshotDeviceResponseInfo_t;

typedef struct CSTACallState_t {
    int    count;
    LocalConnectionState_t *state;
} CSTACallState_t;

typedef enum CSTASimpleCallState_t {
    CALL_NULL = 0,
    CALL_PENDING = 1,
    CALL_ORIGINATED = 3,
    CALL_DELIVERED = 35,
    CALL_DELIVERED_HELD = 36,
    CALL_RECEIVED = 50,
    CALL_ESTABLISHED = 51,
    CALL_ESTABLISHED_HELD = 52,
    CALL_RECEIVED_ON_HOLD = 66,
    CALL_ESTABLISHED_ON_HOLD = 67,
    CALL_QUEUED = 83,
    CALL_QUEUED_HELD = 84,
    CALL_FAILED = 99,
    CALL_FAILED_HELD = 100
} CSTASimpleCallState_t;

```

```

/*
 *
 */
#define SIMPLE_CALL_STATE(ccs) (ccs.stat[0]+(ccs.state[1] << 4))

typedef struct CSTACallState_t {
    int    count;
        LocalConnectionState_t*state;
} CSTACallState_t;

```

## Parameters

### ***acsHandle***

This is the handle for the ACS Stream over which the confirmation arrived. This is the same as the ACS Stream over which the request was made.

### ***eventClass***

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

### ***eventType***

This is a tag with the value **CSTA\_SNAPSHOT\_DEVICE\_CONF**, which identifies this message as an **CSTASnapshotDeviceConfEvent**.

### ***invokeID***

This parameter specifies the service request instance for the **cstaSnapshotDeviceReq()**. The application uses this parameter to correlate responses with requests.

### ***snapshotData***

Contains all the snapshot information for the Device for which the query was made.

**count:** A count of the number of *CSTASnapshotDeviceResponseInfo\_t* structures. Each contains information about one Call at the Device.

**info:** A pointer to an array of *CSTASnapshotDeviceResponseInfo\_t* structures, each of which contains the following fields:

**callIdentifier** — A pointer to a Connection Identifier for each call at the device. For some implementations, this parameter points to the device's dynamic device identifier for the call object.

**callState** — The CSTA Call State. The Call State is returned as a list of local Call States.

### ***privateData***

If private data accompanies this event, 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 **CSTASnapshotDeviceConfEvent** does not deliver private data to the application.

### Comments

The **CSTASnapshotDeviceConfEvent** returns a linked list since the number of calls on a device can be greater than one. Each member of the list identifies a call at the device, and the local call state of the Connection for that call at the device (see Figure 7-2). An application should be aware that the number of members on the list is not fixed. The pointer, *\*next*, will be *NULL* for the last member (call) on the list.

---

CSTA Computing Functions are those functions where the switching domain is the client (service requester) and the computing domain is the server. Presently, Application Call Routing is the only CSTA Computing Function. A switch uses application call routing when it needs the application to supply call destinations on a call-by-call basis. Applications can use internal databases together with call information to determine a destination for each call. For Example, an application might use the caller's number, caller-entered digits (provided as private data), or information in an application database to route incoming calls.

### TSAPI Version 2 Routing

TSAPI Version 2 more closely aligns with the CSTA standard. Version 2 adds two new TSAPI functions and provides additional information in two event reports. The new function calls are **cstaRouteSelectInv( )** and **cstaRouteEndInv( )**, and the two enhanced events are **CSTARouteRequestEvent** and **CSTARouteUsedEvent**.

The new functions, **cstaRouteSelectInv( )** and **cstaRouteEndInv( )**, allow an application to supply an invoke identifier (*invokeID*) in these requests. Although TSAPI does not define a confirmation event for these services, the CSTA standard does define a negative acknowledgement. By using the Version 2 functions, the application can associate a potential **CSTAUniversalFailureConfEvent** with the original request. This is not possible with the Version 1 APIs because they did not include an *invokeID* parameter (a PBX Driver may instead send a **CSTARouteEndEvent** to indicate an error, but doing so is not strictly in compliance with the CSTA standard).

The enhancements in the routing event reports, **CSTARouteRequestEvent** and **CSTARouteUsedEvent** allow inclusion of *ExtendedDeviceID\_t* type fields for certain event data instead of the currently defined *DeviceID\_t* types. In version 2, the **currentRoute** and **callingDevice** members of **CSTARouteRequestExtEvent** are defined as *CalledDeviceID\_t* and *CallingDeviceID\_t* types, respectively. The **routeUsed** and **callingDevice** members of **CSTARouteUsedEvent** are defined as *CalledDeviceID\_t* and *CallingDeviceID\_t* types, respectively. The version 2 definitions align with the CSTA standard. Note that a routing server application that requests and gets the TSAPI version 2 (i.e., "TS2") will always receive the version 2 events. Existing applications that do not use version control to request the Version 2 API version will continue to receive the Version 1 events.

## **Application Call Routing**

---

Application call routing requires that the switch be configured to direct calls to a special type of device known as the "routing device". When a call arrives at a routing device, the switch sends a message to the Telephony Server requesting a route for the call.

Before an application can route calls, it must register with the Telephony Server as a routing server. The application may either register as the routing server for a specific routing device or as the default routing server for an advertised service. Recall that a PBX driver advertises its services. Often these services correspond to a CTI link, so an application can, in effect, register to be the default routing server for a CTI link. An application uses **costaRouteRegisterReq( )** to register as a routing server. This request has an associated confirmation event, **CSTARouteRegisterReqConfEvent**, that contains the routing cross-reference identifier (*routingCrossRefID*) that the application uses to identify requests that arrive on this registration.

### **⇒ NOTE:**

At any one time, one, and only one application can be the routing server for a routing device. Similarly, one, and only one application can be the default routing server for an advertised service.

### **Routing Procedure**

The registration above must occur before this procedure can take place. This procedure description uses the version 2 functions.

1. The switch queues an incoming call at a special device object, the routing device. The routing device may be a "soft" extension on the switch for application-based routing, or similar device defined within the switching domain.
2. When the call arrives at the routing device, the switch and the Telephony Services PBX driver create a CSTA routing dialog for the call. The PBX driver allocates a *routing cross-reference identifier* (*routingCrossRefID*) that references this routing dialog.

3. The PBX Driver directs the route request to the application registered as the routing server for the routing device. If no application is registered for that specific routing device, then the PBX Driver directs the route request to the default routing server application for its advertised service. The routing application receives an unsolicited *route request* event (CSTARouteRequestEvent) for the call. This event contains the routing cross-reference identifier and call information (such as calling and called numbers). The application which provides the call routing destination is called the "routing server" for the routing device.
4. the routing server sends the switch a *route select* message (cstaRouteSelectInv) containing a destination for the call. The routing server typically uses information from the *route request* event together with information from an application database to determine this destination. The routing server may include an optional flag in the *route select* (*routeUsedReq*) instructing the switch to inform it of the final destination for the call. The final destination may be different than the application-provided destination when switch features such as call forwarding redirect the call.
5. the switch receives the *route select* message (cstaRouteSelectInv) and attempts to route the call to the application-provided destination. If the destination is valid, the switch routes the call to that destination and sends the application a *route end* event (cstaRouteEndInv). This terminates the routing dialog for that call. If the application-provided destination is not valid (an invalid extension number, the destination is busy, etc.), then the switch may send a *re-route* event (CSTAReRouteRequestEvent) to the application to request another route to a different destination.
6. If the application receives a *re-route* event (CSTAReRouteRequestEvent) it can select a different destination for the call and send the switch another *route select* message (cstaRouteSelectInv). Depending on the switch implementation, the re-routing message exchange can repeat until the application provides a valid route. The routing server application will find out about a successful route if the switch sends a *route end* event (cstaRouteEndInv) or if the application included the *routeUsedReq* flag in its last *route select* message (cstaRouteSelectInv).

Either the switch or the routing server (application) may send a *route end* event (**cstaRouteEndInv**) to end the routing process and terminate the CSTA routing dialog (this releases the *routing cross-reference identifier*, **routingCrossRefID** for use in the future). Either endpoint may send a route end at any time. This message indicates that the routing server does not want to route the call, or the switch (usually in the absence of a **cstaRouteSelectInv** message) routed the call using some mechanism within the switching domain.

 **NOTE:**

Certain switch implementations may not support the optional flags described above.

Figure 8-17 illustrates the Routing Procedure.

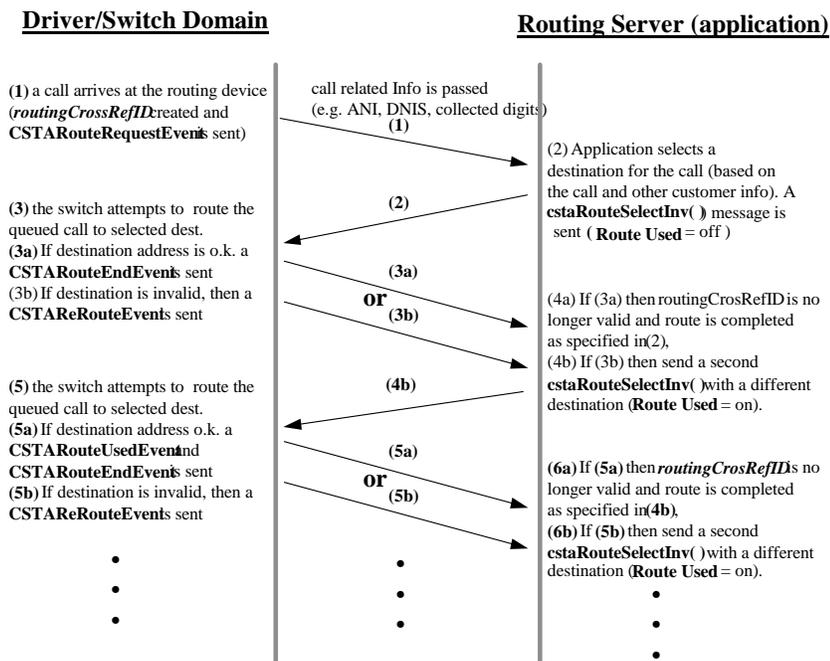


Figure 8-1. Routing Procedure

## Routing Registration Functions and Events

This section describes the service requests and events that an application uses to register with the Telephony Server as a call routing server

### cstaRouteRegisterReq()

An application uses `cstaRouteRegisterReq()` to register as a routing server for a specific routing device or as a default routing server for an advertised service. The application must register for routing services before it can receive any route requests for a routing device. An application may be a routing server for more than one routing device. However, only one application may be a routing server for any given routing device. Similarly, only one application may register as the default routing server for an advertised service.

## Syntax

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

RetCode_t cstaRouteRegisterReq (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    DeviceID_t    *routingDevice,
    PrivateData_t *privateData);
```

## Parameters

### ***acsHandle***

This is the handle to an active ACS Stream over which the routing dialog will take place.

### ***invokeID***

This is an application provided handle that the application uses to match a specific instance of **cstaRouteRegisterReq( )** request with its **CSTARouteRegisterReqConfEvent** confirmation event. The application supplies this parameter only when the Invoke ID mechanism is set for Application-generated IDs in **acsOpenStream( )**. The ACS Library ignores this parameter when the Stream is set for Library-generated invoke IDs.

### ***routingDevice***

This is a pointer to a device id for the routing device for which the application requests to be the routing server. The routing device can be any device type which the switch implementation supports as a routing device. A NULL value indicates that the requesting application will be the default routing server for the **ServerID** associated with the **acsHandle** in the **cstaRouteRegisterReq( )**. The default routing server will receive switch routing requests for any routing devices making routing requests on that advertised service that do not have registered routing servers. Thus, the default routing server will receive route requests when a routing device sends a route request and there is no corresponding registered routing server for that routing device.

### ***privateData***

This is an optional pointer to CSTA private data.

## Return Values

**cstaRouteRegisterReq( )** 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, the invoke identifier. If the call fails it will return a negative error (<0). 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 it will return a negative error (<0). For application-generated identifiers the return is never positive (>0).

An application should always check the **CSTARouteRegisterReqConfEvent** message to ensure to ensure that the Telephony Server and switch have acknowledged the **cstaRouteRegisterReq( )**.

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown ***acsHandle***.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

### Comments

In order for an application to route calls the application must successfully call **cstaRouteRegisterReq( )**. An application can register as:

- a routing server for the specified routing device, or
- as the default routing server for all routing devices making requests of a specific Telephony Server.

To register as a default routing server, an application sets the ***routingDevice*** parameter to NULL. One, and only one, application is allowed to register as the routing server for a ***routingDevice***, or as the default routing Server for an advertised service. Applications may register for routing services for a specific device even when a default routing server has registered. A default routing server will not receive any routing requests from any routing device for which there is a registered routing server. Once a routing server is registered, **CSTARouteRequestEvents** convey the route requests to the routing server.

## CSTARouteRegisterReqConfEvent

The **RouteRegisterReqConfEvent** indicates successful registration to an application. That application is now the call routing server for the requested routing device (or is the default routing server for the advertised service).

### Syntax

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** and ***CSTA Data Types*** in section 4 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
            {
                CSTARouteRegisterReqConfEvent_t routeRegister;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct {
    RouteRegisterReqID_t routeRegisterReqID;
} CSTARouteRegisterReqConfEvent_t;

typedef long      RouteRegisterReqID_t;
```

### Parameters

#### *acsHandle*

This is the handle for the ACS Stream over which the **RouteRegisterReqConfEvent** confirmation arrived. This is the same as the ACS Stream over which the application made the corresponding **cstaRouteRegisterReq()** request.

#### *eventClass*

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

#### *eventType*

This is a tag with the value **CSTA\_ROUTE\_REGISTERREQ\_CONF**, which identifies this message as an **CSTARouteRegisterReqConfEvent**.

#### *invokeID*

This parameter specifies the service request instance for the **cstaRouteRegisterReq()**. The application uses this parameter to correlate **RouteRegisterReqConfEvent** responses with requests.

***routeRegisterReqID***

This parameter contains a handle to the routing registration session for a specific routing device (or for the default routing server depending on the registration request). All routing dialogs (**routingCrossRefIDs**) for a routing device occur over this routing registration session. The PBX Driver selects **routeRegisterReqIDs** so that they will be unique within the **acsHandle**.

***privateData***

If private data accompanies this event, 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 **CSTARouteRegisterReqConfEvent** does not deliver private data to the application.

**Comments**

This event provides the application with a positive confirmation to a request for routing registration.

**cstaRouteRegisterCancel()**

Applications (routing servers) use **cstaRouteRegisterCancel()** to cancel a previously registered routing server session. This request terminates the routing session and the application receives no further routing messages for that session once it receives the confirmation to the cancel request.

**Syntax**

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

RetCode_t cstaRouteRegisterCancel (
    ACSHandle_t      acsHandle,
    InvokeID_t      invokeID,
    RouteRegisterReqID_t routeRegisterReqID,
    PrivateData_t   *privateData);
```

**Parameters*****acsHandle***

This is the handle to an active ACS Stream over which the **cstaRouteRegisterCancel()** request will be made.

***invokeID***

This is an application provided handle that the application uses to match a specific instance of a **cstaRouteRegisterCancel()** request with its confirmation event. The application supplies this parameter only when the

Invoke ID mechanism is set for Application-generated IDs in **acsOpenStream()**. The ACS Library ignores this parameter when the Stream is set for Library-generated invoke IDs.

***routeRegisterReqID***

This parameter is the handle to the routing registration session which the application is canceling. The application received this handle in the confirmation event for the route register service request, a **CSTARouteRegisterReqConfEvent**.

***privateData***

This is an optional pointer to CSTA private data.

**Return Values**

**cstaRouteRegisterCancel()** 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, the invoke identifier. If the call fails it will return a negative error (<0). 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 it will return a negative error (<0). For application-generated identifiers the return is never positive (>0).

The application should always check the **CSTARouteRegisterCancelConfEvent** message to ensure that the Telephony Server and switch have acknowledged and processed the **cstaRouteRegisterCancel()** request.

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown **acsHandle**.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

**Comments**

The application must continue to process outstanding routing requests from the routing device until it receives **CSTARouteRegisterCancelConfEvent**. The Telephony Server will not send any further requests once it has sent this confirmation event.

## CSTARouteRegisterCancelConfEvent

**CSTARouteRegisterCancelConfEvent** confirms a previously issued **cstaRouteRegisterCancel()** request for a routing registration. Once an application receives this event, it invalidates the routing registration session.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTARouteRegisterCancelConfEvent_t routeCancel;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct {
    RouteRegisterReqID_t routeRegisterReqID;
} CSTARouteRegisterCancelConfEvent_t;

typedef long          RouteRegisterReqID_t;
```

### Parameters

#### **acsHandle**

This is the handle for the ACS Stream over which the **CSTARouteRegisterCancelConfEvent** confirmation arrived. This is the same as the ACS Stream over which the **cstaRouteRegisterCancel()** request was made.

#### **eventClass**

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

***eventType***

This is a tag with the value **CSTA\_ROUTE\_REGISTER\_CANCEL\_CONF**, which identifies this message as an **CSTARouteRegisterCancelConfEvent**.

***invokeID***

This parameter specifies the service request instance for the **cstaRouteRegisterCancel()**. The application uses this parameter to correlate the **CSTARouteRegisterCancelConfEvent** responses with requests.

***routeRegisterReqID***

This parameter contains the handle to a routing registration for which the application is providing routing services. The application obtained this handle from a **CSTARouteRegisterReqConfEvent**. This **routeRegisterReqID** handle is no longer valid once the Telephony Server sends **CSTARouteRegisterCancelConfEvent**.

***privateData***

If private data accompanies this event, 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 **CSTASnapshotCallConfEvent** does not deliver private data to the application.

**Comments**

**CSTARouteRegisterCancelConfEvent** confirms an application's **cstaRouteRegisterCancel()** service request, which cancels a routing registration session. The Telephony Server will send any further requests from the routing device to the default routing server.

**CSTARouteRegisterAbortEvent**

The Telephony Server sends an application an unsolicited **CSTARouteRegisterAbortEvent** to cancel an active routing dialog. This event invalidates a routing registration session.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTARouteRegisterAbortEvent_t registerAbort;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct {
    RouteRegisterReqID_t    routeRegisterReqID;
} CSTARouteRegisterAbortEvent_t;

typedef long             RouteRegisterReqID_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream. The routing dialog being canceled is occurring on this ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAEVENTREPORT**, which identifies this message as an CSTA event report.

#### ***eventType***

This is a tag with the value **CSTA\_ROUTE\_REGISTER\_ABORT**, which identifies this message as an **CSTARouteRegisterAbortEvent**.

#### ***routeRegisterReqID***

This parameter is the handle to a routing registration for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent**. The **CSTARouteRegisterAbortEvent** invalidates this handle.

#### ***privateData***

If private data accompanies **CSTARouteRegisterAbortEvent**, 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 **CSTARouteRegisterAbortEvent** does not deliver private data to the application.

### Comments

**CSTARouteRegisterAbortEvent** notifies the application that the PBX driver or switch aborted a routing registration session.

## Routing Functions and Events

This section defines the CSTA call routing services for application call routing. The switch queues calls at the routing device until the application provides a destination for the call or a time-out condition occurs within the switching domain. Figure 8-17 shows the Application-based call routing dialog between a switch and the routing server (the application).

Once an application registers as a routing server, the application uses the services in this section to route calls. The application receives a **CSTARouteRequestEvent** for each call which requires a routing destination. The application sends the switch a destination in **cstaRouteSelectInv()**. The switch then attempts to route the call to that application-provided destination. The switch will respond with a **CSTARouteEndEvent** and/or a **CSTARouteUsedEvent**. If the application-provided destination is invalid, the switch may send a **CSTARouteRequestEvent** to request an additional destination. See Figure 8-17 for a typical sequence of these events and service requests.

### Register Request ID and the Routing Cross-Reference ID

The routing services use two handles (identifiers) to refer to different software objects in the Telephony Server. The register request identifier (**routeRegisterReqID**) identifies a routing session over which an application will receive routing requests. This handle is tied to a routing device on the switch, or it may indicate that the application is the default routing server for an advertised service. When the application uses **cstaRouteRegisterReq()** to register for routing services, it receives a **routeRegisterReqID** in the confirmation. The **routeRegisterReqID** is valid until the registration is canceled or aborted.

Within a routing session (**routeRegisterReqID**) the switch may initiate many routing dialogs (shown in Figure 8-17) to route multiple calls. An application uses a routing cross reference-identifier (**routingCrossRefID**) to refer to each routing dialog. The application receives a **routingCrossRefID** in each **CSTARouteRequestEvent**. The **CSTARouteRequestEvent** initiates a routing dialog. The **routingCrossRefID** is valid for the duration of the call routing dialog.

The routing cross-reference identifier (**routingCrossRefID**) is unique within the routing session (**routeRegisterReqID**). Some switch implementations may provide the additional benefit of a unique routing cross reference-identifier across the entire switching domain. Routing session identifiers (**routeRegisterReqIDs**) are unique within an ACS Stream (**acsHandle**).

**⇒ NOTE:**

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

**CSTARouteRequestEvent**

---

A routing server application receives a **CSTARouteRequestEvent** when the switch requests a route for a call. 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 for the advertised service. The **CSTARouteRequestEvent** event includes call related information (such as the called and calling number, when available). A routing server application typically combines the call related information with an application database to determine a destination for the call. A routing server application receives a **CSTARouteRequestEvent** for every call queued at the routing device.

**TSAPI Version**

Applications using TSAPI Version 1 must use the **CSTARouteRequestEvent\_t** structure; applications using any other version must use the **CSTARouteRequestExtEvent\_t** structure.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTARouteRequestEvent_t    routeRequest;

                /* version 2 union only */
                CSTARouteRequestExtEvent_t routeRequestExt
            }
        }
    }
}
```

```

        } u;
    } cstaRequest;
} event;
CSTAEvent_t;

TSAPI Version 1:
typedef struct {
    RouteRegisterReqID_t    routeRegisterReqID;
    RoutingCrossRefID_t    routingCrossRefID;
    DeviceID_t             currentRoute;
    DeviceID_t             callingDevice;
    ConnectionID_t         routedCall;
    SelectValue_t          routedSelAlgorithm;
    Boolean                 priority;
    SetupValues_t          setupInformation;
} CSTARouteRequestEvent_t;

TSAPI Version 2:

typedef struct {
    RouteRegisterReqID_t    routeRegisterReqID;
    RoutingCrossRefID_t    routingCrossRefID;
    CalledDeviceID_t        currentRoute;           /* V2 */
    CallingDeviceID_t       callingDevice;         /* V2 */
    ConnectionID_t         routedCall;
    SelectValue_t          routedSelAlgorithm;
    Boolean                 priority;
    SetupValues_t          setupInformation;
} CSTARouteRequestExtEvent_t; /* V2 */

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 {
    int             length;
    unsigned char  *value;
} SetupValues_t;

```

## Parameters

### ***acsHandle***

This is the handle for the opened ACS Stream on which the route request event arrives.

### ***eventClass***

This is a tag with the value **CSTAREQUEST**, which identifies this message as an CSTA request message.

***eventType***

This is a tag with the value **CSTA\_ROUTE\_REQUEST** (version 1) or **CSTA\_ROUTE\_REQUEST\_EXT** (version 2), which identifies this message as an **CSTARouteRequestEvent**.

***routeRegisterReqID***

This parameter contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent** confirmation to a route register service request.

***routingCrossRefID***

The application receives this new handle for the routing dialog for this call. This identifier has a new, unique value within the scope of the routing session (***routeRegisterReqID***).

***currentRoute***

This parameter indicates the originally requested destination for the call being application routed. Often, this is the DNIS, or dialed number.

***callingDevice***

This is the originating device of the call, i.e., the calling party number (when available. If not available, it may be trunk information).

***routedCall***

This parameter is a CSTA Connection ID that identifies the call being routed.

***routedSelAlgorithm***

This parameter identifies the routing algorithm being used.

***priority***

This parameter indicates the priority of the call.

***setupInformation***

This parameter includes an ISDN call setup message, if available.

***privateData***

If private data accompanies **CSTARouteRequestEvent**, 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 **CSTARouteRequestEvent** does not deliver private data to the application.

### Comments

CSTARouteRequestEvent informs the routing server (application) that the switch is requesting a destination for a call queued at the routing device. The application uses `cstaRouteSelectInv()` or `cstaRouteSelect()` to respond with a destination.

### ⇒ NOTE:

CSTA requires that all events contain an invoke ID. During routing, the `RouteRegisterReqID` and the `RoutingCrossRefID` identify the routing dialogue. The `invokeID` is not used.

## CSTAReRouteRequestEvent

The switch sends an unsolicited **CSTAReRouteRequestEvent** to request another destination for a call. Typically, the destination that the application previously sent was invalid or busy. The switch previously sent call related information (such as the called and calling numbers) in the **CSTARouteRequestEvent**; Call related information is not re-sent in the **CSTAReRouteRequestEvent**. The routing server application responds using the `cstaRouteSelectInv()` or `cstaRouteSelect()` service.

### Syntax

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

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t invokeID;
            union
            {
                CSTAReRouteRequest_t reRouteRequest;
            } u;
        } cstaRequestEvent;
    } event;
} CSTAEvent_t;

typedef struct
{
    RouteRegisterReqID_t routeRegisterReqID;
    RoutingCrossRefID_t routingCrossRefID;
} CSTAReRouteRequest_t;
```

**Parameters*****acsHandle***

This is the handle for the opened ACS Stream on which the re-route request arrives.

***eventClass***

This is a tag with the value **CSTAREQUEST**, which identifies this message as a CSTA request message.

***eventType***

This is a tag with the value **CSTA\_RE\_ROUTE\_REQUEST**, which identifies this message as a **CSTAReRouteRequestEvent**.

***routeRegisterReqID***

This parameter contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent** confirmation to a route register service request.

***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for this call. The application previously received this handle in a **CSTARouteRequestEvent** for the call.

***privateData***

If private data accompanies **CSTAReRouteRequestEvent**, 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 **CSTAReRouteRequestEvent** does not deliver private data to the application.

**Comments**

The switch can send **CSTAReRouteRequestEvent** to the routing server application when the application previously sent a destination that was invalid or other circumstances exist where routing of the call to the destination is not possible (e.g. the destination is busy). The switch uses **CSTAReRouteRequestEvent** to request another destination for the call queued at the routing device. The application uses **cstaRouteSelect( )** to provide the new destination.

The number of re-route requests that a switch may send depends on the implementation or administration within the switch. The application should be prepared to respond to all re-route requests or terminate the routing dialog by using the **cstaRouteEnd( )** service request when it cannot provide additional destinations.

**⇒ NOTE:**

CSTA requires that all events contain an invoke ID. During routing, the `RouteRegisterReqID` and the `RoutingCrossRefID` identify the routing dialogue. The `invokeID` is not used.

**cstaRouteSelect() TSAPI Version 1 Only**

The routing server application uses `cstaRouteSelect` to send a routing destination to the switch in response to a `CSTARouteRequestEvent` for a call.

**TSAPI Version**

Applications using TSAPI Version 1 must use the `CSTARouteSelect()` function; applications using any other version must use the `CSTARouteSelectInv()` function.

**Syntax**

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

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);
```

**Parameters*****acsHandle***

This is the handle to the ACS Stream on which the routing dialog for the call is taking place.

***routeRegisterReqID***

This parameter contains the active handle to the routing registration session for which the application is providing routing services. The application received this handle in the confirmation event for the route register service request, `CSTARouteRegisterReqConfEvent`, for the call.

***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for this call. The application previously received this handle in the `CSTARouteRequestEvent` for the call.

***routeSelected***

The application provides this parameter containing a Device ID that specifies the destination for the call.

***remainRetry***

The application indicates the number of times it is willing to receive a **CSTAReRouteRequestEvent** for this call in the case that the switch needs to request an alternate route. TSAPI provides #define values that an application may use to indicate that it does not keep count, or that there is no limit.

***setupInformation***

The application provides this optional parameter that contains information for the ISDN call setup message that the switch will use to route the call. Some switches may not support this option.

***routeUsedReq***

The routing application uses this parameter to request a **CSTARouteUsedEvent** for the call. The route used event informs the application of the final destination of the call once it has been routed.

***privateData***

This is an optional pointer to CSTA private data.

**Return Values**

**cstaRouteSelect( )** returns a non-zero value if it completes successfully.

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown **acsHandle**.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

**⇒ NOTE:**

There is no confirmation event for this service request, however this service request can generate a universal failure event.

**Comments**

An application should call **cstaRouteSelect** only in response to a **CSTARouteRequestEvent** or **CSTAReRouteRequestEvent**. The **cstaRouteSelect** service request will fail if the application does not provide valid identifiers from a previous **CSTARouteRequestEvent**, (**acsHandle**, **routeRegisterReqID**, and **routingCrossRefID**). The application should check the return value for this function and any resulting universal failure event for errors.

## **cstaRouteSelectInv()**

---

The routing server application uses **cstaRouteSelectInv** to send a routing destination to the switch in response to a **CSTARouteRequestEvent** for a call.

The invoke identifier parameter lets the application correlate the route selection with a **CSTAUniversalFailureConfEvent** in the case of a failure.

### **TSAPI Version**

Applications using TSAPI Version 1 must use the **CSTARouteSelect()** function; applications using any other version must use the **CSTARouteSelectInv()** function.

### **Syntax**

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

```
RetCode_t cstaRouteSelectInv (
    ACSHandle_t          acsHandle,
    InvokeID_t invokeID, /* Version 2 */
    RouteRegisterReqID_t routeRegisterReqID,
    RoutingCrossRefID_t routingCrossRefID,
    DeviceID_t          *routeSelected,
    RetryValue_t        remainRetry,
    SetUpValues_t       *setupInformation,
    Boolean              routeUsedReq,
    PrivateData_t       *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the handle to the ACS Stream on which the routing dialog for the call is taking place.

#### ***invokeID***

A handle provided by the application to be used for matching a specific instance of a function service request with any resulting **CSTAUniversalFailureConfEvent**. 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.

#### ***routeRegisterReqID***

This parameter contains the active handle to the routing registration session for which the application is providing routing services. The application received this handle in the confirmation event for the route register service request, **CSTARouteRegisterReqConfEvent**, for the call.

***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for this call. The application previously received this handle in the **CSTARouteRequestEvent** for the call.

***routeSelected***

The application provides this parameter containing a Device ID that specifies the destination for the call.

***remainRetry***

The application indicates the number of times it is willing to receive a **CSTARouteRequestEvent** for this call in the case that the switch needs to request an alternate route. TSAPI provides #define values that an application may use to indicate that it does not keep count, or that there is no limit.

***setupInformation***

The application provides this optional parameter that contains information for the ISDN call setup message that the switch will use to route the call. Some switches may not support this option.

***routeUsedReq***

The routing application uses this parameter to request a **CSTARouteUsedEvent** for the call. The route used event informs the application of the final destination of the call once it has been routed.

***privateData***

This is an optional pointer to CSTA private data.

**Return Values**

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown **acsHandle**.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

**⇒ NOTE:**

There is no confirmation event for this service request, however this service request can generate a universal failure event.

**Comments**

An application should call **cstaRouteSelectInv** only in response to a **CSTARouteRequestEvent** or **CSTAReRouteRequestEvent**. The **cstaRouteSelectInv** service request will fail if the application does not provide valid identifiers from a previous **CSTARouteRequestEvent** or **CSTAReRouteRequestEvent** (**acsHandle**, **routeRegisterReqID**, and **routingCrossRefID**). The application should check the return value for this function and any resulting universal failure event for errors.

**CSTARouteUsedEvent**

The **CSTARouteUsed** event provides a routing server application with the actual destination of a call for which the application previously sent a **cstaRouteSelect()** or **cstaRouteSelectInv()**. To receive a **CSTARouteUsed**, the application must set the **cstaRouteSelect()** or **cstaRouteSelectInv()** parameter **routeUsedReq** to TRUE.

**TSAPI Version**

Applications using TSAPI Version 1 must use the **CSTARouteUsedEvent\_t** structure; applications using any other version must use the **CSTARouteUsedExtEvent\_t** structure.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTARouteUsedEvent_t routeUsed;
            }
        }
    }
}
```

```

        /* version 2 only */
        CSTARouteUsedExtEvent_t routeUsedExt
    } u;
    } cstaEventReport;
    } event;
} CSTAEvent_t;

TSAPI Version 1:
typedef struct
{
    RouteRegisterReqID_t    routeRegisterReqID;
    RoutingCrossRefID_t    routingCrossRefID;
    DeviceID_t             routeUsed;
    DeviceID_t             callingDevice;
    Boolean                 domain;
} CSTARouteUsedEvent_t;

TSAPI Version 2:
typedef struct
{
    RouteRegisterReqID_t    routeRegisterReqID;
    RoutingCrossRefID_t    routingCrossRefID;
    CalledDeviceID_t       routeUsed; /* Version 2 */
    CallingDeviceID_t     callingDevice; /* Version 2 */
    Boolean                 domain;
} CSTARouteUsedExtEvent_t;

```

## Parameters

### *acsHandle*

This is the handle to the ACS Stream on which the routing dialog for the call is taking place.

### *eventClass*

This is a tag with the value **CSTA\_EVENTREPORT**, which identifies this message as an CSTA unsolicited event.

### *eventType*

This is a tag with the value **CSTA\_ROUTE\_USED** (version 1) or **CSTA\_ROUTE\_USED\_EXT** (version 2), which identifies this message as an **CSTARouteUsedEvent**.

### *routeRegisterReqID*

This parameter contains the active handle to the routing registration session for which the application is providing routing services. The application received this handle in the confirmation event for the route register service request, **CSTARouteRegisterReqConfEvent**, for the call.

***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for this call. The application previously received this handle in the **CSTARouteRequestEvent** for the call.routeUsed. This parameter identifies the selected and final destination for the call.

***callingDevice***

This parameter contains the originating device of the call, i.e. the calling party number (when available).

***domain***

This parameter will indicate whether the call has left the switching domain accessible to the Telephony Server (the **ServerID** defined in the active **acsHandle**). Typically, a call leaves a switching domain when it is routed to a trunk connected to another switch or to the public switched network.

***privateData***

If private data accompanies **CSTARouteUsedEvent**, 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 **CSTARouteUsedEvent** does not deliver private data to the application.

**Comments**

An application uses **CSTARouteUsedEvent** to determine the final destination of a call that it routed using the **cstaRouteSelect( )** or **cstaRouteSelectInv( )**. Switch features such as forwarding or routing tables may direct the call to a device other than the application supplied destination. The **CSTARouteUsedEvent** indicates the final destination for the call.

**CSTARouteEndEvent**

---

The switch sends **CSTARouteEndEvent** to terminate a routing dialog. The event includes a cause value giving the reason for the dialog termination.

Note that when an application wishes to terminate a routing interaction, it should use **CSTARouteEnd( )** (version 1) or **CSTARouteEndInv( )** (version 2).

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

```

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

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTARouteEndEvent_t    routeEnd,
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct
{
    RouteRegisterReqID_t    routeRegisterReqID;
    RoutingCrossRefID_t    routingCrossRefID;
    CSTAUniversalFailure_t    errorValue;
} CSTARouteEndEvent_t;

```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream on which routing dialog is ending.

#### ***eventClass***

This is a tag with the value **CSTAEVENTREPORT**, which identifies this message as a CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_ROUTE\_END**, which identifies this message as a **CSTARouteEndEvent**.

#### ***routeRegisterReqID***

This parameter contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent** confirmation to a route register service request.

#### ***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for this call. The application previously received this handle in the **CSTARouteRequestEvent** for the call.

***errorValue***

This parameter contains a cause code which giving the reason why the routing dialog ended.

***privateData***

If private data accompanies **CSTARouteEndEvent**, 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 **CSTARouteEndEvent** does not deliver private data to the application.

**Comments**

The switch sends **CSTARouteEndEvent** when a call has been successfully routed, cleared, or when the routing server has failed to provide a route select within the switch's time limit. This event is unsolicited and can occur at any time.

**cstaRouteEnd() TSAPI Version 1 Only**

A routing application uses **cstaRouteEnd( )** to cancel an active routing dialog for a call. The service request includes a cause value giving the reason for the routing dialog termination. Note that when the switch terminates an active routing dialog, it uses **cstaRouteEndEvent**.

**TSAPI Version**

Applications using TSAPI Version 1 must use the **CSTARouteEnd( )** function; applications using any other version must use the **CSTARouteEndInv( )** function.

**Syntax**

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

RetCode_t cstaRouteEnd (
    ACSHandle_t           acsHandle,
    RouteRegisterReqID_t  routeRegisterReqID,
    RoutingCrossRefID_t   routingCrossRefID,
    CSTAUniversalFailure_t errorValue;
    PrivateData_t         *privateData);
```

**Parameters**

***acsHandle***

This is the handle for the opened ACS Stream on which the application is terminating a routing dialog for a call.

***routeRegisterReqID***

This parameter contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent** confirmation to a route register service request.

***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for a call. The application previously received this handle in the **CSTARouteRequestEvent** for the call. This is the routing dialog that the application is ending.

***errorValue***

The application supplies this cause code giving the reason why it is ending the routing dialog.

***privateData***

This is an optional pointer to CSTA private data.

**Return Values**

**cstaRouteEnd()** returns a non-negative value when successful.

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown **acsHandle**.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

**Comments**

A routing application can use **cstaRouteEnd()** when it cannot route a call. This can occur if:

- the application receives a routing request for a call without sufficient call information and it cannot determine a routing destination.
- the application has already routed calls to all available destinations and those calls remain active at those destinations.
- the application does not have access to a database necessary to route the call

In these cases, the application uses **cstaRouteEnd()** to inform the switch that it will not route the call in question. **cstaRouteEnd()** will terminate the CSTA routing dialog (**routingCrossRefID**) for the call. **cstaRouteEnd()** does not clear the call. The switch will continue to process the call using whatever default routing algorithm is available (implementation specific).

Note that when the switch terminates an active routing dialog, it uses **cstaRouteEndEvent**.

## **cstaRouteEndInv()**

---

The routing server (application) uses **cstaRouteEnd()** to cancel an active routing dialog for a call. The service request includes a cause value giving the reason for the routing dialog termination. Note that when the switch terminates an active routing dialog, it uses **cstaRouteEndEvent**.

### **TSAPI Version**

Applications using TSAPI Version 1 must use the **CSTARouteEnd()** function; applications using any other version must use the **CSTARouteEndInv()** function.

### **Syntax**

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

RetCode_t cstaRouteEnd (
    ACSHandle_t          acsHandle,
    InvokeID_invokeID;   /* Version 2 */
    RouteRegisterReqID_t routeRegisterReqID,
    RoutingCrossRefID_t  routingCrossRefID,
    CSTAUniversalFailure_t errorValue;
    PrivateData_t        *privateData);
```

### **Parameters**

#### ***acsHandle***

This is the handle for the opened ACS Stream on which the application is terminating a routing dialog for a call.

#### ***invokeID***

A handle provided by the application to be used for matching a specific instance of a function service request with any resulting **CSTAUniversalFailureConfEvent**. 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.

#### ***routeRegisterReqID***

This parameter contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a **CSTARouteRegisterReqConfEvent** confirmation to a route register service request.

#### ***routingCrossRefID***

This parameter contains the handle to the CSTA call routing dialog for a call. The application previously received this handle in the **CSTARouteRequestEvent** for the call. This is the routing dialog that the application is ending.

***errorValue***

The application supplies this cause code giving the reason why it is ending the routing dialog.

***privateData***

This is an optional pointer to CSTA private data.

**Return Values**

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

The application provided a bad or unknown ***acsHandle***.

***ACSERR\_STREAM\_FAILED***

A previously active ACS Stream has been abnormally aborted.

**Comments**

A routing server application can use **cstaRouteEndInv( )** when it cannot route a call. This can occur if:

- the application receives a routing request for a call without sufficient call information and it cannot determine a routing destination.
- the application has already routed calls to all available destinations and those calls remain active at those destinations.
- the application does not have access to a database necessary to route the call

In these cases, the application uses **cstaRouteEnd( )** to inform the switch that it will not route the call in question. **cstaRouteEnd( )** will terminate the CSTA routing dialog (***routingCrossRefID***) for the call. **cstaRouteEnd( )** does not clear the call. The switch will continue to process the call using whatever default routing algorithm is available (implementation specific).

An application can use this function to respond to either a route request or a re-route request.

Note that when the switch terminates an active routing dialog, it uses **cstaRouteEndEvent**.



---

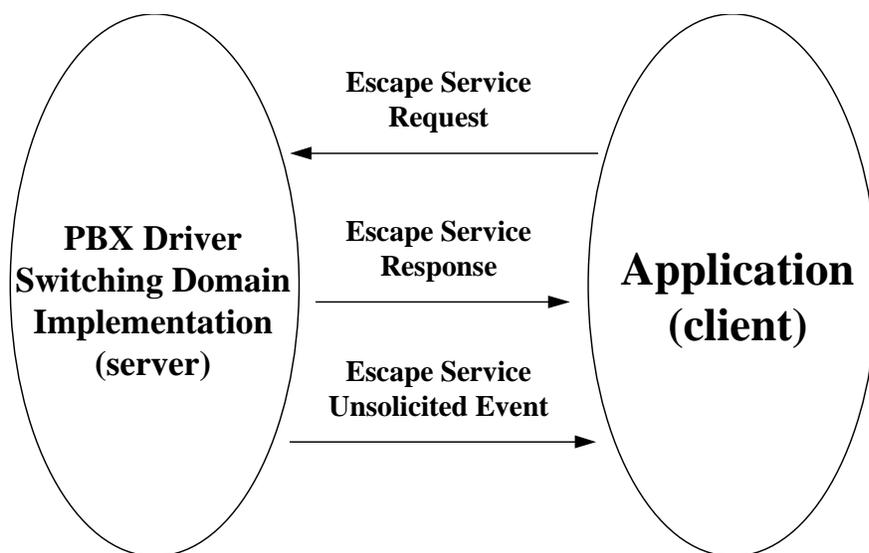
This chapter describes the CSTA Escape and Maintenance Services.

### Escape Services

Switching domains use Escape Services to enhance TSAPI functions with "private" services which are specific to the switch or PBX Driver implementation (see Section 2.3). Each switch vendor may define functions within the CSTA private services framework, even though CSTA does not incorporate these services. Although the functions defined within escape services can vary from one implementation to the next, the way the application accesses these functions is consistent. Escape Services use the same programming model as all other CSTA services. Figure 9-18 illustrates this model.

When an application requests an escape service from a server it receives a confirmation event or a Universal Failure in the same fashion as for other TSAPI services. The Escape Service request parameters are an ***acsHandle*** (to the open stream), an ***invokeID*** and a private data parameter. The confirmation event includes the ***acsHandle***, the ***invokeID***, and the private data response.

Escape Services also includes an unsolicited private event which a server can send to an application at any time a CSTA monitor association exists on a CSTA call or device object (see Section 6 - ***cstaMonitorStart()***).




---

**Figure 9-1. Escape Services Model**

Applications can also send Escape Services requests to a switch. For most CSTA services the application is always a client in the computing domain. However, an escape service could operate in the opposite direction (such as routing does). Although the client/server role may change, services are always uni-directional where either the switch or application is always the requester for a service.

TSAPI includes escape service definitions for both the "*Application as the Client*" and the "*Switch as the Client*".

**⇒ NOTE:**

See vendor specific documentation for more information on what, if any, Escape Services are supported by a specific vendor. Escape Service Functions are generally not portable across different vendor implementations. Some implementations may support Escape Services either bi-directionally or uni-directionally (one-way only) depending on the needs and capabilities of the switch driver

## Maintenance Services

---

There are two different types of CSTA maintenance services:

- device status maintenance events which provide status information for device objects, and
- bi-directional system status maintenance services which provide information on the overall status of the system.

The device status events inform the application when the switch places a monitored device in or out of service. When a device object is removed from service, the application may monitor the device (e.g. **cstaMonitorStart( )** or **cstaDevSnapshotReq( )**) but may not request services for that device. For example, an application request for a **cstaMakeCall( )** returns an error when the device is out of service.

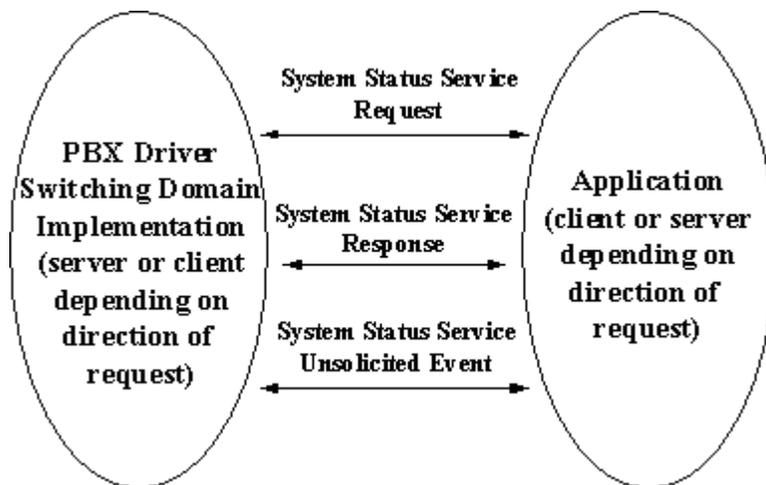
System Status services inform applications or switches about the status of the switching or computing domains, respectively. Table 9-3 shows the System Status Services' system status information (cause codes).

### System Status Cause Codes

System Status Cause Code	Cause Code Definitions
Initializing	the system is re-initializing or restarting. This status indicates that the system is temporarily unable to respond to any requests. If provided, this status message shall be followed by an Enable status message that indicates that the initialization process is completed.
Enabled	request and responses are enabled, usually after a disruption or restart. This status indication shall be sent after an initializing status indicator has been sent and may be sent under other conditions. This status indicates that there are no outstanding monitor requests.
Normal	a System Status Event with this cause value can be sent at any time to indicate that the status is normal. This status has no effect on other services.
Message Lost	this status indicates that a request, response, or event report may have been lost.
Disabled	this cause value indicates that active <b>cstaMonitorStart( )</b> monitor requests via have been disabled. Other requests and responses may also be disabled, but, unlike monitors, reject responses are provided for those.
Overload Imminent	the system (driver, switch, or application) is about to reach an overload condition. The "client" should shed load to remedy the situation.

Overload Reached	the system (driver, switch, or application) has reached an overload condition and may take action to shed load. The server (the application, driver, or switch) may then take action to decrease message traffic. This may include stopping existing monitors or rejecting any new requests sent by the client.
Overload Relieved	the system (driver, switch, or application) has determined that the overload condition has passed and normal application operation may resume.

The System Status services are bi-directional and thus can originate at the application domain or at the driver/switch domain. Figure 9-19 shows System Status Maintenance Services.




---

**Figure 9-2. System Status Maintenance Services**

An application can obtain System Status information in one of two different ways :

- the client can ask for the information using a request to the "server" and obtain the information in a confirmation event, or
- the client can register for System Status messages and receive unsolicited events containing system status changes.

A switch or application may issue the System Status request (**cstaSysStatReq()**) to obtain status information from the "server" (the application or switch, respectively, depending on the direction of the request). A System Status response (**CSTASysStatReqConfEvent**) provides the "client" with the current system status information for the "server". The "server" may send unsolicited events the client if the client used the **cstaSysStatStart()** service to register for System Status events. The System Status unsolicited event (**CSTASysStatEvent**) is the same in structure as the confirmation event (**CSTASysStatReqConfEvent**) except that the "server" sends it to the "client" automatically.

## **Escape Services : Application as Client**

This section defines escape services for situation where the application is the service requester in the client/server relationship (see Figure 9-18). The services include an escape service request, a confirmation event to the request, and an unsolicited escape service event that originates at the driver or switching domain.

### **cstaEscapeService()**

This service allows the application to request a service which is not defined by the CSTA Standard but rather by a switch vendor. A service request made by this function will be specific to an implementation.

#### **Syntax**

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

RetCode_t    cstaEscapeService (
    ACSHandle_t    acsHandle,
    InvokeID_t    invokeID,
    PrivateData_t    *privateData);
```

#### **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 valid when the Invoke ID mechanism is set for Application-generated IDs in the **acsOpenStream()**. The parameter is ignored by the ACS Library when the Stream is set for Library-generated invoke IDs.

***privateData***

This is a pointer to the CSTA private data extension mechanism. This parameter is NOT optional for this function and must be passed by the application. If the parameter is NULL an error will be returned to the application and the API Client Library Driver will reject the service request.

**Return Values**

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

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

***ACSERR\_NULLPARAMETER***

This error indicates that the pointer to the CSTA Private Data information is NULL and thus no private data is available to send to the driver/switch. No action is taken by the API Client Library Driver.

**Comments**

This function is used to send private data information to the driver/switch.

## **CSTAEscapeServiceConfEvent**

---

This confirmation event is sent in response to the **cstaEscapeService()** service and provides the positive acknowledgment to the request. The event includes any private information that is to be provided as part of a confirmation event to the service request.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

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

#### ***eventType***

This is a tag with the value **CSTA\_ESCAPE\_SERVICE\_CONF** which identifies this message as an **CSTAEscapeServiceConfEvent**.

#### ***invokeID***

This parameter specifies the function service request instance for the service which was processed at the Telephony Server or at the switch. This identifier is provided to the application when a service request is made.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event always occurs as a result of a normal (positive) service request made through the **cstaEscapeService()** service. The information contained in the ***privateData*** parameter is implementation specific.

**CSTAPrivateEvent**

This event report allows for unsolicited, implementation specific event reporting. The informational contents of this event will be defined by a specific implementation.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAPrivateEvent_t privateData;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;
```

### Parameters

***acsHandle***

This is the handle for the opened ACS Stream.

***eventClass***

This is a tag with the value **CSTAEVENTREPORT**, which identifies this message as an CSTA unsolicited event.

***eventType***

This is a tag with the value **CSTA\_PRIVATE**, which identifies this message as an **CSTAPrivateEvent**.

***monitorCrossRefID***

Does not apply to this event.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is typically used for providing unsolicited, implementation specific event information. This event can occur at any time and does not have to be specific to a monitored object. The event can be sent by the driver/switch even though the application does not have a monitored object. When a monitor exists, the **PrivateStatusEvent** is used by the driver/switch to send private status information pertaining to a monitored object. The **PrivateEvent** is used for all other cases of unsolicited private events and is not associated with a monitoring association.

## CSTAPrivateStatusEvent

This event report allows for unsolicited, implementation specific event reporting for a monitored object. The informational contents of this event will be defined by a specific implementation.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_tacsHandle;
    EventClass_teventClass;
    EventType_teventType;
```

```
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_tmonitorCrossRefID;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_PRIVATE\_STATUS**, which identifies this message as an **CSTAPrivateStatusEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is typically used for providing implementation specific event information which is not defined in any other event in the API. The event is always used for private information on a monitoring association. A monitor must be active [**csaMonitorDevice()**, **csaMonitorCall()**, **csaMonitorCallsViaDevice()**] before this event can be sent to the application by the driver/switch. This event is always sent from the driver/switch to the application and it is not bi-directional.

## Escape Service : Driver/Switch as the Client

---

This section defines escape services for cases where the Driver/Switch is the service requester in the client/server relationship (see Figure 9-1). The services include an escape service request event, a confirmation function for the request, and an unsolicited escape service event that originates at the application domain.

### CSTAEscapeServiceReq

---

This unsolicited event is sent by the driver/switch to request a private service from the application. The event includes the service request as private information for which the application must provide a positive or negative acknowledgment.

#### Syntax

The following structure shows only the relevant portions of the unions for this message. See **4.3 ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTAEscapeSvcReqEvent_t escapeSvcRequest;
            } u;
        } cstaRequestEvent;
    } event;
} CSTAEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the opened ACS Stream.

### ***eventClass***

This is a tag with the value **CSTAREQUEST**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_ESCAPE\_SVC\_REQ**, which identifies this message as an **CSTAEscapeServiceReq**

### ***invokeID***

This parameter defines the invoke identifier selected by the driver/switch for the specific private request. This parameter *must* be returned, unchanged, in the response to this request in order for the driver/switch to match a service request with a service response.

### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## Comments

This event is sent by the driver/switch to request an escape or private service when the application is providing the "server" function in the client/server relationship. The response to this event will be accomplished via the **cstaEscapeServiceConf()** service.

## **cstaEscapeServiceConf()**

This service allows the application to respond to a **CSTAEscapeServiceEvent** which originated at the driver/switch. A service response made by this function will be specific to an implementation.

### Syntax

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

RetCode_t  cstaEscapeServiceConf (
    ACSHandle_t          acsHandle,
    InvokeID_t           invokeID,
    CSTAUniversalFailure_t  error,          /* negative ACK */
    PrivateData_t        *privateData), /* positive ACK */
```

## Parameters

### ***acsHandle***

This is the handle to an active ACS Stream.

### ***invokeID***

The invoke identifier used in this function *must* be the same value (unchanged) as that provided in the **CSTAEscapeServiceReq** for which this services request is being called. The same ***invokeID*** value must be used in order for the driver/switch to match the instances of a previous service event and the service confirmation to that event provided by this function call.

### ***error***

This parameter is used to provide a negative acknowledgment to the **CSTAEscapeServiceReq**. See **CSTAUniversalFailureConfEvent** for a definition of the possible error values for this parameter. If the ***error*** pointer is NULL this will indicate that the event contains a positive acknowledgment.

### ***privateData***

This is a pointer to the CSTA private data extension mechanism which contains the positive acknowledgment to the CSTAEscapeServiceEvent. If the private pointer is NULL this will indicate that the event contains a negative acknowledgment.

## Return Values

This function never returns an invoke identifier since there is no confirmation event for this service. The function does return errors conditions during the processing of the request by the API Client Library. A return value of zero (0) indicates that the request has been accepted by the Library. This function never returns a positive value.

Possible local error returns are (negative returns):

### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

### ***ACSERR\_NONULL***

This error indicates that neither the ***error*** or ***privateData*** pointers are NULL. One of these pointers *must be* NULL to indicate either a positive or negative acknowledgment to the request. No action is taken by the API Client Library.

#### **ACSERR\_NORESPONSE**

This error indicates that both the ***error*** or the ***privateData*** pointer are NULL. In this case the API Client Library has nothing to send to the driver/switch and rejects the response. The request associated with the invoke identifier from the driver/server will still be outstanding and the application must respond by calling this function with acceptable parameters.

#### **ACSERR\_BADINVOKEID**

This error indicates that the invoke identifier being returned by the application is not one that is outstanding from the driver/switch. The API Client Library will keep track of the driver/switch-based invoke id's until the application responds to the specific request from the driver/switch.

#### **Comments**

This function is used to send a response to a private request from the driver/switch. The event supports both a positive and negative acknowledgment to the request. One of the two pointers (***error*** or ***privateData***) must be NULL in order for the request to be successfully processed by the API Client Library. This would indicate a positive or negative acknowledgment to the request made by the driver/switch.

### **cstaSendPrivateEvent()**

This service allows the application to send an unsolicited private event to the driver/switch. An event sent by this function will be specific to an implementation.

#### **Syntax**

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

RetCode_t  cstaSendPrivateEvent (
    ACSHandle_t  acsHandle,
    PrivateData_t *privateData),
```

#### **Parameters**

##### ***acsHandle***

This is the handle to an active ACS Stream.

##### ***privateData***

This is a pointer to the CSTA private data extension mechanism. This parameter is NOT optional for this function and must be passed by the application. If the parameter is NULL an error will be returned to the application and the API Client Library Driver will reject the service request.

### Return Values

This function never returns an invoke identifier since there is no confirmation event for this service. The function does return errors conditions during the processing of the request by the API Client Library. A return value of zero (0) indicates that the request has been accepted by the Library. This function never returns a positive value.

Possible local error returns are (negative returns):

#### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

#### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

#### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

#### ***ACSERR\_NULLPARAMETER***

This error indicates that the pointer to the CSTA Private Data information is NULL and thus no private data is available to send to the driver/switch. No action is taken by the API Client Library Driver.

### Comments

This function is used to send unsolicited, private data information to the driver/switch when the application is supporting the "server" role in the client/server relationship.

## **Maintenance Services: Device Status**

This section describes the CSTA Maintenance Services which provide device status information. To receive device status information, an application must monitor the device(e.g. the application must have an active ***monitorCrossRefID*** for the device). These events are unidirectional and always originate in the switch domain.

## **CSTABackInServiceEvent**

---

This event report indicates that a monitored device object has returned to services and operates normally.

### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTABackInServiceEvent_t backInService;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    DeviceID_t     device;
    CSTAEventCause_t cause;
} CSTABackInServiceEvent_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_BACK\_IN\_SERVICE**, which identifies this message as an **CSTABackInServiceEvent**.

***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

***device***

Specifies the device which is back in service. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

***cause***

This parameter indicates the reason or explanation for the occurrence of this event. See Section 6 for more information.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event indicates that a previously inactive device (a device which is out of service) has resumed normal operation. Once this event has occurred the application can then initiate an active service request (e.g. **cstaMakeCall()**) for that specific device. A passive service request can be done while a device is out of service, i.e. monitoring or Snapshot Services.

**CSTAOutOfServiceEvent**

This event report indicates that a monitored device object has entered a maintenance state and can no longer accept calls or be actively manipulated by the application.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 for a complete description of the event structure.

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

```
typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAOutOfServiceEvent_t outOfService;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    DeviceID_t device;
    CSTAEventCause_t cause;
} CSTAOutOfServiceEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_OUT\_OF\_SERVICE**, which identifies this message as an **CSTAOutOfServiceEvent**.

#### ***monitorCrossRefID***

This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.

#### ***device***

This parameter indicates the device which has been taken out of service. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.

#### ***cause***

This parameter indicates the reason or explanation for the occurrence of this event. See Section 6 for more information.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This event indicates that a previously active device (a device which is in service) has entered into a maintenance state, i.e. the device has been taken out of service. Once this event has occurred the application *can not* initiate any new active service request (e.g. **cstaMakeCall()**) for that specific device. A passive service request (e.g. monitoring or Snapshot Services) can be done while a device is out of service.

## **System Status - Application as the Client**

---

This section defines the services which provide system level status information to the application or the driver/switch. The System Status service is bi-directional and thus the client/server relationship (see Figure 9-2) can be reversed.

### **cstaSysStatReq()**

---

This service allows the application to request system status information from the driver/switch domain.

**Syntax**

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

RetCode_t cstaSysStatReq (
    ACSHandle_tacsHandle,
    InvokeID_tinvokeID,
    PrivateData_t*privateData);
```

**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 valid when the Invoke ID mechanism is set

for Application-generated IDs in the **acsOpenStream()**. The parameter is ignored by the ACS Library when the Stream is set for Library-generated invoke IDs.

***privateData***

This is a pointer to the CSTA private data extension mechanism. This is optional.

**Return Values**

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

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

**Comments**

This function is used to request the current overall system status for the driver/switch.

**CSTASysStatReqConfEvent**

This event is in response to the **ctaSysStatReq( )** service and informs the application of the overall system status of the driver/switch.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASysStatReqConfEvent_t sysStatReq;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASysStatReqConfEvent_t (
    SystemStatus_t systemStatus;
) CSTASysStatReqConfEvent_t;

typedef enum SystemStatus_t {
    SS_INITIALIZING = 0,
    SS_ENABLED = 1,
    SS_NORMAL = 2,
    SS_MESSAGES_LOST = 3,
    SS_DISABLED = 4,
    SS_OVERLOAD_IMMINENT = 5,
    SS_OVERLOAD_REACHED = 6,
    SS_OVERLOAD_RELIEVED = 7
} SystemStatus_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an CSTA unsolicited event.

***eventType***

This is a tag with the value **CSTA\_SYS\_STAT\_REQ\_CONF**, which identifies this message as an **CSTASysStatReqConfEvent.invokeID**

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

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

***systemStatus***

This parameter provides the application with a cause code defining the overall system status as in Table 9-4.

**Overall System Status Codes**

<b>Cause Code</b>	<b>Definition</b>
Initializing	the driver/switch is re-initializing or restarting. This status indicates that the driver/switch is temporarily unable to respond to any requests. If provided, this status message shall be followed by an Enable status message to indicate that the initialization process is completed.
Enabled	request and responses are re-enabled, usually after a disruption or restart. This status indication shall be sent after an initializing status indicator has been sent and may be sent under other conditions. This status indicates that there are no outstanding monitor request.
Normal	this cause value can be sent at any time by the driver/switch to indicate that the status is normal. This status has no effect on other services.
Message Lost	this status indicates that a request and/or responses may have been lost, including Event Reports.
Disabled	this cause value indicates that existing monitor requests via <b>cstaMonitorStart()</b> have been disabled. Other requests and responses may also be disabled, but reject responses should be provided.
Overload Imminent	the driver/switch is about to reach a overload condition and the application should shed load to better the situation.

Overload Reached	the driver/switch has reach overload and may take initiative to shed load. This cause may be followed by action on the part of the driver/switch to decrease message traffic. This may include stopping existing or rejecting any new monitor requests sent by the client, and rejections to additional new service requests.
Overload Relieved	the driver/switch has determined that the overload condition has passed and normal application operation may continue.

### Comments

This confirmation event provides the application with certain information regarding the state of the overall driver/switch system. This event is important for proper application operation and should be processed accordingly. This is especially important for cause values for the overload condition. If the driver/switch has informed the application that an overload condition is imminent all applications should attempt to decrease the overall traffic to the driver/switch. This can be accomplished, for example, by stopping all non-essential monitors on call or device objects on the switch thus reducing the traffic between the server and the switch. Frequent occurrence of the Overload Imminent cause value can be a symptoms of a poorly engineered system which should reviewed for proper loading.

### cstaSysStatStart()

This services allows the application to register for System Status event reporting. It can be used by an application to automatically receive a **CSTASysStatEvent** each time the status of the driver/switch changes.

### Syntax

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

RetCode_t cstaSysStatStart (
    ACSHandle_t          acsHandle,
    InvokeID_t          invokeID,
    SystemStatusFilter_t statusFilter,
    PrivateData_t       *privateData),

typedef unsigned char  SystemStatusFilter_t;
#define                SF_INITIALIZING 0x80
#define                SF_ENABLED 0x40
#define                SF_NORMAL 0x20
#define                SF_MESSAGES_LOST 0x10
#define                SF_DISABLED 0x08
#define                SF_OVERLOAD_IMMINENT 0x04
#define                SF_OVERLOAD_REACHED 0x02
#define                SF_OVERLOAD_RELIEVED 0x01
```

## 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 valid 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.

### ***statusFilter***

This parameter is used to specify a filter for specific cause values in which the application is not interested. The parameter can be used by the application to filter out unwanted status information (e.g. the Normal status)

### ***privateData***

Private data extension mechanism. This is optional.

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

### Comments

This function is used to start a monitor for system status information. The system status information is provided via the **CSTASysStatEvent**. Only one System Status register is allowed per opened ACS Stream.

## CSTASysStatStartConfEvent

This event is in response to the **cstaSysStatStart()** function and confirms an active System Status monitor. Once this event is issued the application will start to automatically receive unsolicited System Status events.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASysStatStartConfEvent_t sysStatStart;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASysStatStartConfEvent_t (
    SystemStatusFilter_t statusFilter;
) CSTASysStatStartConfEvent_t;
```

### Parameters

**acsHandle**

This is the handle for the opened ACS Stream.

**eventClass**

This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an CSTA unsolicited event.

**eventType**

This is a tag with the value **CSTA\_SYS\_STAT\_START\_CONF**, which identifies this message as an **CSTASysStatStartConfEvent**.

**invokeID**

This parameter specifies the requested instance of the function or event. It is used to match a specific functions call request with its confirmation events.

**statusFilter**

This parameter is used to specify the filter type which is active on the System Status monitor requested by the application. The parameter identifies which filter was accepted by the driver/switch. Note that the filter returned by this function may be different than the filter requested in the **cstaSysStatStart()** service request. This can occur when the driver/switch rejected the request filter and selected a default filter.

**privateData**

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This confirmation event should be checked by the application to insure that the System Status monitor has been activated and that the requested filter is active.

**cstaSysStatStop()**

This service is used to cancel a previously registered monitor for System Status information.

**Syntax**

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

RetCode_t cstaSysStatStop (
    ACSHandle_tacsHandle,
    InvokeID_tinvokeID,
    PrivateData_t*privateData),
```

## 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 valid when the Invoke ID mechanism is set for Application-generated IDs in the **acsOpenStream()**. The parameter is ignored by the ACS Library when the Stream is set for Library-generated invoke IDs.

### ***privateData***

Private data extension mechanism. This is optional.

## Return Values

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

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

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

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

The following are possible negative error conditions for this function:

### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

## Comments

This function is used to cancel a previously registered System Status monitor. Once a confirmation event is issued for this function, i.e. a **CSTASysStatStopConfEvent**, the driver/switch will terminate automatic System Status event notification. If required, the application can still continue to poll for system status information using the **cstaSysStatReq()** service, even after a System Status register is closed.

## CSTASysStatStopConfEvent

This event is in response to the **cstaSysStatStop()** function and confirms a cancellation of the active System Status monitor. Once this event is issued the application will not continue to receive unsolicited System Status events.

## Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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_t;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;
```

## Parameters

### ***acsHandle***

This is the handle for the opened ACS Stream.

### ***eventClass***

This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an CSTA unsolicited event.

### ***eventType***

This is a tag with the value **CSTA\_SYS\_STAT\_STOP\_CONF**, which identifies this message as an CSTASysStatStopConfEvent.

***invokeID***

This parameter specifies the requested instance of the function or event. It is used to match a specific functions call request with its confirmation events.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This confirmation event should be checked by the application to insure that the System Status monitor has been deactivated. Once this event is sent, automatic notification of System Status events will be discontinued. The application must poll using the **cstaSysStatReq()** service in order to obtain any System Status information.

**cstaChangeSysStatFilter()**

This function is used to request a change in the filter options for automatic System Status event reporting for a specific ACS Stream. It allows the application to specify which System Status events it requires.

**Syntax**

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

```
RetCode_t cstaChangeSysStatFilter (
    ACSHandle_t          acsHandle,
    InvokeID_t           *invokeID,
    SystemStatusFilter_t statusFilter,
    PrivateData_t        *privateData),
```

**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 valid 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.

***statusFilter***

This parameter identifies the new filter mask to be applied to the existing active System Status monitor. The new mask will replace the existing mask.

***privateData***

Private data extension mechanism. This is optional.

**Return Values**

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

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

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

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

The following are possible negative error conditions for this function:

***ACSERR\_BADHDL***

This return value indicates that a bad or unknown **acsHandle** was provided by the application.

***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

**Comments**

This service is used whenever the application wishes to change a previously defined System Status event filter. Note that application will not receive any System Status message which has its bit mask turned off.

## CSTAChangeSysStatFilterConfEvent

This event occurs as a result of the `cstaChangeSysStatFilter()` service and informs the application which event filter was set by the server.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See *ACS Data Types* and *CSTA Data Types* in section 4 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
            {
                CSTAChangeSysStatFilterConfEvent_t changeSysStatFilter;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAChangeSysStatFilterConfEvent_t (
    SystemStatusFilter_t statusFilterSelected;
    SystemStatusFilter_t statusFilterActive;
) CSTAChangeSysStatFilterConfEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_CHANGE\_SYS\_STAT\_FILTER\_CONF**, which identifies this message as an CSTAChangeSysStatFilterConfEvent.

#### ***invokeID***

This parameter specifies the requested instance of the function or event. It is used to match a specific functions call request with its confirmation events.

***statusFilterSelected***

This parameter specifies the System Status event filters which are active as a result of the **cstaChangeSysStatFilter()** service request. This filter may be different than the one requested by the application. This can occur if the implementation rejects a particular filter request.

***eventFilterActive***

This parameter indicates the filters which are already active on the given CSTA association.

***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

**Comments**

This confirmation event should be checked by the application to insure that the event filter requested has been activated and which filters are already active on the given System Status monitor.

**CSTASysStatEvent**

This unsolicited event informs the application of the overall system status of the driver/switch. The application must register for System Status events before this event is sent to the application.

**Syntax**

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASysStatEvent_t sysStat;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;
```

```
typedef struct
{
    SystemStatus_t systemStatus;
} CSTASysStatEvent_t;

typedef enum SystemStatus_t {
    SS_INITIALIZING = 0,
    SS_ENABLED = 1,
    SS_NORMAL = 2,
    SS_MESSAGES_LOST = 3,
    SS_DISABLED = 4,
    SS_OVERLOAD_IMMINENT = 5,
    SS_OVERLOAD_REACHED = 6,
    SS_OVERLOAD_RELIEVED = 7
} SystemStatus_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAEVENTREPORT**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_SYS\_STAT**, which identifies this message as an **CSTASysStatEvent**.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

#### ***systemStatus***

This parameter provides the application with a cause code defining the overall system status. See Table 9-4 for the possible values of this parameter.

### Comments

This event provides the application with certain information regarding the state of the overall driver/switch system. This event is important for proper application operation and should be processed accordingly. This is especially important for cause values for the overload condition. If the driver/switch has informed the application that an overload condition is imminent all applications should attempt to decrease the overall traffic to the driver/switch. This can be accomplished, for example, by stopping all non-essential monitors on call or device objects on the switch thus

reducing the traffic between the server and the switch. Frequent occurrence of the Overload Imminent event can be a symptoms of a poorly engineered system which should reviewed for proper loading.

PBX drivers that follow driver programming recommendations will send **CSTASysStatEvent** when a CTI link goes up or down. When a link that is in service goes down, the **systemStatus** will be "Initializing". When a link enters service, the **systemStatus** will be "Enabled". If a driver uses multiple CTI links to provide service, then the driver may only send the "Initializing" message when there are no CTI links in service and may send the "Enabled" when at least one link is in service.

Certain, non-essential cause values can be sent at any time or depending on the driver/switch implementation even at regular intervals (e.g. the Normal cause value) to indicate that the system status is O.K. and operating normally. This can be turned off by the application to avoid the overhead associated with processing these normal messages. This is accomplished by changing the event filter type by using the **cstaChangeSysStatFilter()** service. This service can be used to discontinue the delivery of "non-essential" system status events to the application.

## CSTASysStatEndedEvent

The driver uses this event to cancel a previously registered monitor for System Status information.

### Syntax

The following structure shows only the relevant portions of the unions for this message. See **ACS Data Types** and **CSTA Data Types** in section 4 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
            {
                CSTASysStatEndedEvent_t sysStatEnded;
            } u;
        } cstaEventReport;
    }
}
```

```
    } event;  
} CSTAEvent_t;  
  
typedef struct CSTASysStatEndedEvent_t {  
    Nulltype      null;  
} CSTASysStatEndedEvent_t;
```

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAEVENTREPORT**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_SYS\_STAT\_ENDED**, which identifies this message as a CSTASysStatStopEvent.

#### ***monitorCrossRefID***

This parameter is unused in this message.

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

## **System Status : Driver/Switch as the Client**

---

This section defines the services which provide system level status information to the driver/switch from the application. The System Status service is bi-directional and thus the client/server relationship (see Figure 9-2) can be reversed.

### **CSTASysStatReqEvent**

---

This unsolicited event is sent by the driver/switch to request system status information from the application.

#### **Syntax**

The following structure shows only the relevant portions of the unions for this message. See section **4.3 ACS Data Types** and **4.6 CSTA Data Types** 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
            {
                CSTASysStatReqEvent_t sysStatRequest;
            } u;
        } cstaRequestEvent;
    } event;
} CSTAEvent_t;

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

### Parameters

#### ***acsHandle***

This is the handle for the opened ACS Stream.

#### ***eventClass***

This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.

#### ***eventType***

This is a tag with the value **CSTA\_SYS\_STAT\_REQ**, which identifies this message as an **CSTASysStatReqEvent**.

#### ***invokeID***

This parameter identifies the instance of the request generated by the switch/driver. This same value must be used, unchanged, in the response to this event (**cstaSysStatReqConf()**).

#### ***privateData***

If private data accompanied this event, then the private data would be copied to the location pointed to by the *privateData* pointer in the **acsGetEventBlock()** or **acsGetEventPoll()** function. If the *privateData* pointer is set to NULL in these functions, then no private data will be delivered to the application.

### Comments

This event is sent by the driver/switch to request status information pertaining to the application. It is the bi-directional equivalent of the **cstaSysStatReq()** function which is issued by the application to request status information from the driver/switch. The application responds to this unsolicited event request utilizing the **cstaSysStatReqConf()** function.

## **cstaSysStatReqConf()**

---

This service is used to respond to a **CSTASysStatReqEvent** unsolicited event from the driver/switch. It provides the driver/switch with information regarding the status of the application.

### **Syntax**

```
#include <csta.h>

RetCode_t cstaSysStatReqConf (

    ACSHandle_tacsHandle,
    InvokeID_t*invokeID,
    SystemStatus_tsystemStatus,
    PrivateData_t*privateData);

typedef enum SystemStatus_t {
    SS_INITIALIZING = 0,
    SS_ENABLED = 1,
    SS_NORMAL = 2,
    SS_MESSAGES_LOST = 3,
    SS_DISABLED = 4,
    SS_OVERLOAD_IMMINENT = 5,
    SS_OVERLOAD_REACHED = 6,
    SS_OVERLOAD_RELIEVED = 7
} SystemStatus_t;
```

### **Parameters**

#### ***acsHandle***

This is the handle to an active ACS Stream.

#### ***InvokeID***

The value of this parameter must be the same (unchanged) as that provided in the **cstaSysStatReqEvent** so that the driver/switch can match an instance of a service request with the response to that request.

#### ***systemStatus***

This parameter provides the driver/switch with a cause code defining the overall system status. See Table 9-4 for the possible values of this parameter.

#### ***privateData***

Private data extension mechanism. This is optional.

### Return Values

This function never returns an invoke identifier since there is no confirmation event for this service. The function does return errors conditions during the processing of the request by the API Client Library. A return value of zero (0) indicates that the request has been accepted by the Library.

Possible local error returns are (negative returns):

#### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

#### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

#### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

### Comments

This confirmation response provides the driver/switch with certain information regarding the state of the overall application. The information can be used by the driver/switch to determine the overall state of the application. The driver/switch may act on this information in order to insure proper end-to-end system operation and performance. Frequent occurrence of the Overload Imminent cause value can be a symptoms of a poorly engineered application system which should reviewed for proper loading.

### cstaSysStatEvent()

This service is used to send application system status information in the form of an unsolicited event to the driver/switch without a formal request for the information. This status information can be sent at any time.

### Syntax

```
#include <csta.h>

RetCode_t cstaSysStatEvent (
    ACSHandle_tacsHandle,
    SystemStatus_tsystemStatus,
    PrivateData_t*privateData);
```

```
typedef enum SystemStatus_t {
    SS_INITIALIZING = 0,
    SS_ENABLED = 1,
    SS_NORMAL = 2,
    SS_MESSAGES_LOST = 3,
    SS_DISABLED = 4,
    SS_OVERLOAD_IMMINENT = 5,
    SS_OVERLOAD_REACHED = 6,
    SS_OVERLOAD_RELIEVED = 7
} SystemStatus_t;
```

### Parameters

#### ***acsHandle***

This is the handle to an active ACS Stream.

#### ***systemStatus***

This parameter provides the driver/switch with a cause code defining the overall system status. See Table 9-4 for the possible values of this parameter.

#### ***privateData***

Private data extension mechanism. This is optional.

### Return Values

This function never returns an invoke identifier since there is no confirmation event for this service. The function does return errors conditions during the processing of the request by the API Client Library. A return value of zero (0) indicates that the request has been accepted by the Library.

Possible local error returns are (negative returns):

#### ***ACSERR\_BADHDL***

This return value indicates that a bad or unknown ***acsHandle*** was provided by the application.

#### ***ACSERR\_NOCONN***

This return value indicates that a previously active ACS Stream has been abnormally aborted.

#### ***ACSERR\_REQDENIED***

This return value indicates that a ACS Stream is established but a requested capability has been denied by the Client Library Software Driver.

### Comments

This unsolicited service event is sent to the driver/switch in order to inform it of the state of the overall application system. The driver/switch may act on this information in order to insure proper end-to-end system operation and performance. Frequent occurrence of the Overload Imminent cause value can be a symptoms of a poorly engineered application system which should reviewed for proper loading.

---

**Introduction**

---

This chapter provides programming notes for the following clients

- TSAPI on Win32
- TSAPI on Unixware
- TSAPI on HP-UX

**TSAPI on Win32 Programming Overview**

---

Read this section for information on developing TSAPI applications for Windows NT or Windows 95. You do not need to be familiar with the CSTA call model or API before reading further, but you should read Chapter 4, ACS Control Services.

**Development Platforms**

---

The TSAPI header files and import libraries in this SDK are compatible with the Microsoft Visual C++ Development System. 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, etc. The Win32 TSAPI library assumes the default 8-byte structure packing and an enum size of 4 bytes.

**Linking to the TSAPI Library**

---

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

## **Using Application Control Services**

---

This section discusses how to use application control services (ACS) to retrieve events on Win32 platforms. If you are porting code that uses Telephony Services, you should read this section to get an overview of the differences between Win32 and other platforms.

### **Event Notification**

---

`acsEventNotify()` enables asynchronous notification of incoming events via Windows messages.

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

### **Receiving Events**

---

This section discusses 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.

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 Telephony Services 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 TSSPY32.EXE program may be used to obtain a trace of messages flowing between applications and the Telephony Server.

## **TSAPI on UnixWare**

---

Read this section for information on developing TSAPI applications on UnixWare. You do not need to be familiar with the CSTA call model or API before reading further, but you should read Chapter 4, ACS Control Services.

### **Development Platforms**

---

Telephony Services applications must be built with an environment that supports the Executable and Linking Format (ELF) and dynamic linking. The TSAPI header files in this SDK are compatible with the C Optimized Compilation System provided with the UnixWare Software Development Kit. 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, etc.

TSAPI works with both UnixWare 1.x and UnixWare 2.x, and is thread-safe for 2.x applications.

### **Linking to the TSAPI Library**

---

The TSAPI for UnixWare 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 -o myprog myprog.c -lcsta
```

Note that `libcsta.so` depends upon the Networking Support Library, `libnsl.so`.

### **Using Application Control Services**

---

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

#### **Event Notification**

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

Unlike other Telephony Services platforms, UnixWare does not directly promote an event-driven programming model, but rather a file-oriented one. To work most effectively in the UnixWare environment, the TSAPI event stream should appear as a file. The `acsGetFile()` function returns the STREAMS 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. Alternatively, an application may register to receive the SIGPOLL signal using the `I_SETSIG` `ioctl()` call. Refer to Programming with UNIX System Calls - STREAMS Polling and Multiplexing in the UnixWare SDK documentation.



**NOTE:**

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 discusses event reception using `acsGetEventPoll()` and `acsGetEventBlock()` on UnixWare.

### **Blocking Versus Polling**

`acsGetEventBlock()` 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. `acsGetEventPoll()` 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()` system call, or installing a SIGPOLL handler.

### **Receiving Events From Any Stream**

An application may specify a NULL stream handle when calling `acsGetEventPoll()` or `acsGetEventBlock()` to request that the Telephony Services 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 Telephony Server, 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 pseudo-code illustrates the use of the `acsGetFile()` function to set up an asynchronous event handler.

```
int EventIsPending = 0;

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

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

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

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

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

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

    /* enable SIGPOLL on normal "read" events */
    ioctl (acs_fd, I_SETSIG, S_RDNORM);

    /* proceed with application processing */
    while (notDone)
    {
        if (EventIsPending > 0)
        {
            /* retrieve a TSAPI event */
            acsGetEventPoll (acsHandle, ...etc...);
            EventIsPending = 0;
            /* re-enable handler */
            signal (SIGPOLL, handleEvent);
        }
        /* perform other background processing... */
    }
}
```

## **TSAPI on HP-UX**

---

Read this section for information on developing TSAPI applications on HP-UX. You do not need to be familiar with the CSTA call model or API before reading further, but you should read Chapter 4, ACS Control Services.

### **Development Platforms**

---

The TSAPI header files in this SDK are compatible with the HP-UX 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, etc.

### **Linking to the TSAPI Library**

---

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

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

### **Using Application Control Services**

---

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

### **Event Notification**

The `acsEventNotify()` and `acsSetESR()` functions are not provided on the HP-UX platform.

Unlike other Telephony Services platforms, HP-UX does not directly promote an event-driven programming model, but rather a file-oriented one. To work most effectively in the HP-UX 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.

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 discusses event reception using `acsGetEventPoll()` and `acsGetEventBlock()` on HP-UX.

### **Blocking Versus Polling\*\***

`acsGetEventBlock()` 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. `acsGetEventPoll()` 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 Telephony Services 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 Telephony Server, 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.

## Using High Memory on Windows Clients

---

Windows client application developers need to be aware of interactions between the TSAPI DLL and Windows memory allocation.

The CSTA Telephony Server client library, `csta.dll`, is required to occupy fixed memory, because it interfaces to NetWare drivers. As a result, when `csta.dll` loads into memory, Windows memory allocation attempts to place it in the lowest possible memory location by moving all movable memory to higher locations. If the client loads multiple applications with this same behavior, then memory below one megabyte may become exhausted. A symptom is that when the user attempts to start an application Windows displays "insufficient memory to start this application."

Developers can code their programs to force Windows to allocate low memory before the TSAPI DLL loads, then free up the memory after the DLL loads. The C code below illustrates how to do this. Note that if any other application has already loaded `csta.dll` the following code has no effect, because `csta.dll` is fixed in memory.

```
for ( i=0; i<20; ++i )
{
    // grab as much base memory as possible
    if ( (GlobDret=GlobalDosAlloc(50000L)) != NULL)
        {
            GselectorDos2[i] = LOWORD(GlobDret);
        }
    else if ( (GlobDret=GlobalDosAlloc(10000L)) != NULL)
        {
            GselectorDos2[i] = LOWORD(GlobDret);
        }
    else if ( (GlobDret=GlobalDosAlloc(5000L)) != NULL)
        {
            GselectorDos2[i] = LOWORD(GlobDret);
        }
    else if ( (GlobDret=GlobalDosAlloc(1024L)) != NULL)
        {
            GselectorDos2[i] = LOWORD(GlobDret);
        }
    else break;
}

for (i = 19; i >= 0; --i)
{
    // free some of the grabbed base memory
    if ( GselectorDos2[i] )
        {
            GlobalDosFree(GselectorDos2[i]);
            GselectorDos2[i] = 0;
            break;
        }
}
```

```
        }
    }

    // load the .dll or start the application that will load
    it.

    for (i=0; i<20; ++i)
    {
        // free all the captured base memory
        if ( GselectorDos2[i] )
        {
            GlobalDosFree(GselectorDos2[i]);
        }
        else
        {
            break;
        }
    }
}
```

---

### Introduction

---

This chapter describes the data types used by the functions and messages defined for the TSAPI. The data type are divided into two categories: CSTA Data Type which are associated with telephony functions and Interface Data Types which are associated with the API itself and the Telephony Services client/server interface.

### Device Identifiers

---

```
typedef char    Nulltype;

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;
```

## Basic Call Control Confirmation Events

---

### CSTAAlternateCallConfEvent structures

---

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

### CSTAAnswerCallConfEvent structures

---

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

typedef enum Feature_t {
    FT_CAMPON = 0,
    FT_CALLBACK = 1,
    FT_INTRUDE = 2
} Feature_t;
```

### CSTACallCompletionConfEvent structures

---

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

### CSTAClearCallConfEvent structures

---

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

### CSTAClearConnectionConfEvent structures

---

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

### **CSTAConferenceCallConfEvent structures**

```
typedef struct Connection_t {
    ConnectionID_t party;
    DeviceID_t     staticDevice; /* NULL for not present */
} Connection_t;

typedef struct ConnectionList {
    int         count;
    Connection_t *connection;
} ConnectionList;

typedef struct CSTAConferenceCallConfEvent_t {
    ConnectionID_t activeCall;
    ConnectionList connList;
} CSTAConferenceCallConfEvent_t;
```

### **CSTAConsultationCallConfEvent structures**

```
typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;
```

### **CSTADeflectCallConfEvent structures**

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

### **CSTAGroupPickupCallConfEvent structures**

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

### **CSTAHoldCallConfEvent structures**

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

### **CSTAMakeCallConfEvent structures**

```
typedef struct CSTAMakeCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAMakeCallConfEvent_t;
```

### **CSTAMakePredictiveCallConfEvent structures**

```
typedef enum AllocationState_t {
    AS_CALL_DELIVERED = 0,
    AS_CALL_ESTABLISHED = 1
} AllocationState_t;

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

### **CSTAPickupCallConfEvent structures**

```
typedef struct CSTAPickupCallConfEvent_t {
    Nulltype null;
} CSTAPickupCallConfEvent_t;
CSTARReconnectCallConfEvent structures
typedef struct CSTARReconnectCallConfEvent_t {
} CSTARReconnectCallConfEvent_t;
```

### **CSTARRetrieveCallConfEvent structures**

```
typedef struct CSTARRetrieveCallConfEvent_t {
    Nulltype null;
} CSTARRetrieveCallConfEvent_t;
CSTATransferCallConfEvent structures
typedef struct CSTATransferCallConfEvent_t {
    ConnectionID_t resultingCall;
    ConnectionList connList;
} CSTATransferCallConfEvent_t;
```

## CSTAUniversalFailureEvent structures

```
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 = 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
} CSTAUniversalFailure_t;

typedef struct CSTAUniversalFailureConfEvent_t {
    CSTAUniversalFailure_t error;
} CSTAUniversalFailureConfEvent_t;
```

## Telephony Supplementary Confirmation Events

---

### CSTASetMsgWaitingConfEvent structures

---

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

### CSTASetDndConfEvent structures

---

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

### CSTASetFwdConfEvent structures

---

```
typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0,
    FWD_BUSY = 1,
    FWD_NO_ANS = 2,
    FWD_BUSY_INT = 3,
    FWD_BUSY_EXT = 4,
    FWD_NO_ANS_INT = 5,
    FWD_NO_ANS_EXT = 6
} ForwardingType_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t forwardingType;
    Boolean          forwardingOn;
    DeviceID_t      forwardDN; /* NULL for not present */
} ForwardingInfo_t;

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

### **CSTASetAgentStateConfEvent structures**

---

```
typedef enum AgentMode_t {
    AM_LOG_IN = 0,
    AM_LOG_OUT = 1,
    AM_NOT_READY = 2,
    AM_READY = 3,
    AM_WORK_NOT_READY = 4,
    AM_WORK_READY = 5
} AgentMode_t;

typedef char          AgentID_t[32];

typedef DeviceID_t   AgentGroup_t;

typedef char          AgentPassword_t[32];

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

### **CSTAQueryMwiConfEvent structures**

---

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

### **CSTAQueryDndConfEvent structures**

---

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

### CSTAQueryFwdConfEvent structures

```
typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0,
    FWD_BUSY = 1,
    FWD_NO_ANS = 2,
    FWD_BUSY_INT = 3,
    FWD_BUSY_EXT = 4,
    FWD_NO_ANS_INT = 5,
    FWD_NO_ANS_EXT = 6
} ForwardingType_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t forwardingType;
    Boolean          forwardingOn;
    DeviceID_t      forwardDN;
} ForwardingInfo_t;

typedef struct ListForwardParameters_t {
    short          count;
    ForwardingInfo_t param[7];
} ListForwardParameters_t;

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

### CSTAQueryAgentStateConfEvent structures

```
typedef enum AgentState_t {
    AG_NOT_READY = 0,
    AG_NULL = 1,
    AG_READY = 2,
    AG_WORK_NOT_READY = 3,
    AG_WORK_READY = 4
} AgentState_t;

typedef struct CSTAQueryAgentStateConfEvent_t {
    AgentState_t agentState;
} CSTAQueryAgentStateConfEvent_t;
CSTAQueryLastNumberConfEvent structures
typedef struct CSTAQueryLastNumberConfEvent_t {
    DeviceID_t lastNumber;
} CSTAQueryLastNumberConfEvent_t;
```

**CSTAQueryDeviceInfoConfEvent structures**

---

```
typedef enum DeviceType_t {
    DT_STATION = 0,
    DT_LINE = 1,
    DT_BUTTON = 2,
    DT_ACD = 3,
    DT_TRUNK = 4,
    DT_OPERATOR = 5,
    DT_STATION_GROUP = 16,
    DT_LINE_GROUP = 17,
    DT_BUTTON_GROUP = 18,
    DT_ACD_GROUP = 19,
    DT_TRUNK_GROUP = 20,
    DT_OPERATOR_GROUP = 21,
    DT_OTHER = 255
} DeviceType_t;

typedef unsigned char DeviceClass_t;
#define DC_VOICE 0x80
#define DC_DATA 0x40
#define DC_IMAGE 0x20
#define DC_OTHER 0x10

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

## Status Reporting Confirmation Events

### cstaMonitorDevice structures

```
typedef long          CSTAMonitorCrossRefID_t;

typedef CSTAObject_t  CSTAMonitorObject_t;

typedef unsigned short CSTACallFilter_t;
    CF_CALL_CLEARED 0x8000
    CF_CONFERENCED 0x4000
    CF_CONNECTION_CLEARED 0x2000
    CF_DELIVERED 0x1000
    CF_DIVERTED 0x0800
    CF_ESTABLISHED 0x0400
    CF_FAILED 0x0200
    CF_HELD 0x0100
    CF_NETWORK_REACHED 0x0080
    CF_ORIGINATED 0x0040
    CF_QUEUED 0x0020
    CF_RETRIEVED 0x0010
    CF_SERVICE_INITIATED 0x0008
    CF_TRANSFERRED 0x0004

typedef unsigned char  CSTAFeatureFilter_t;
    FF_CALL_INFORMATION 0x80
    FF_DO_NOT_DISTURB 0x40
    FF_FORWARDING 0x20
    FF_MESSAGE_WAITING 0x10

typedef unsigned char  CSTAAgentFilter_t;
    AF_LOGGED_ON 0x80
    AF_LOGGED_OFF 0x40
    AF_NOT_READY 0x20
    AF_READY 0x10
    AF_WORK_NOT_READY 0x08
    AF_WORK_READY 0x04

typedef unsigned char  CSTAMaintenanceFilter_t;
    MF_BACK_IN_SERVICE 0x80
    MF_OUT_OF_SERVICE 0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t call;
    CSTAFeatureFilter_t feature;
    CSTAAgentFilter_t agent;
    CSTAMaintenanceFilter_t maintenance;
    Boolean          private;
} CSTAMonitorFilter_t;

typedef enum CSTAMonitorType_t {
    MT_CALL = 0,
    MT_DEVICE = 1
} CSTAMonitorType_t;
```

### **cstaMonitorCall structures**

See cstaMonitorDevice structures.

### **cstaMonitorCallsViaDevice structures**

See cstaMonitorDevice structures.

### **CSTAMonitorConfEvent structures**

```
typedef struct CSTAMonitorStartConfEvent_t {
    CSTAMonitorCrossRefID_t monitorCrossRefID;
    CSTAMonitorFilter_t monitorFilter;
} CSTAMonitorStartConfEvent_t;
```

### **CSTACHangeMonitorFilterConfEvent structures**

```
typedef struct CSTACHangeMonitorFilterConfEvent_t {
    CSTAMonitorFilter_t filterList;
} CSTACHangeMonitorFilterConfEvent_t;
```

### **CSTAMonitorStopConfEvent structures**

```
typedef struct CSTAMonitorStopConfEvent_t {
    Nulltype null;
} CSTAMonitorStopConfEvent_t;
CSTAMonitorStopEvent structures
typedef struct CSTAMonitorStopEvent_t {
    InvokeID_t invokeID;
} CSTAMonitorStopEvent_t
```

## Call Event Reports

---

### Call Event Report data structures

---

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

typedef enum CSTAEventCause_t {
    ACTIVE_MONITOR = 1,
    ALTERNATE = 2,
    BUSY = 3,
    CALL_BACK = 4,
    CALL_CANCELLED = 5,
    CALL_FORWARD_ALWAYS = 6,
    CALL_FORWARD_BUSY = 7,
    CALL_FORWARD_NO_ANSWER = 8,
    CALL_FORWARD = 9,
    CALL_NOT_ANSWERED = 10,
    CALL_PICKUP = 11,
    CAMP_ON = 12,
    DEST_NOT_OBTAINABLE = 13,
    DO_NOT_DISTURB = 14,
    INCOMPATIBLE_DESTINATION = 15,
    INVALID_ACCOUNT_CODE = 16,
    KEY_CONFERENCE = 17,
    LOCKOUT = 18,
    MAINTENANCE = 19,
    NETWORK_CONGESTION = 20,
    NETWORK_NOT_OBTAINABLE = 21,
    NEW_CALL = 22,
    NO_AVAILABLE_AGENTS = 23,
    OVERRIDE = 24,
    PARK = 25,
    OVERFLOW = 26,
    RECALL = 27,
    REDIRECTED = 28,
    REORDER_TONE = 29,
    RESOURCES_NOT_AVAILABLE = 30,
    SILENT_MONITOR = 31,
    TRANSFER = 32,
    TRUNKS_BUSY = 33,
    VOICE_UNIT_INITIATOR = 34
} CSTAEventCause_t;
```

## CSTACallClearedEvent structures

```
typedef struct CSTACallClearedEvent_t {
    ConnectionID_t   clearedCall;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTACallClearedEvent_t;
```

## CSTAConferencedEvent structures

```
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
} DeviceIDType_t;

typedef enum DeviceIDStatus_t {
    PROVIDED = 0,
    NOT_KNOWN = 1,
    NOT_REQUIRED = 2
} DeviceIDStatus_t;

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

typedef ExtendedDeviceID_t SubjectDeviceID_t;

typedef struct CSTAConferencedEvent_t {
    ConnectionID_t primaryOldCall;
    ConnectionID_t secondaryOldCall;
    SubjectDeviceID_t confController;
    SubjectDeviceID_t addedParty;
    ConnectionList conferenceConnections;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAConferencedEvent_t;
```

### CSTAConnectionClearedEvent structures

```
typedef struct CSTAConnectionClearedEvent_t {
    ConnectionID_t    droppedConnection;
    SubjectDeviceID_t releasingDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAConnectionClearedEvent_t;
CSTADeliveredEvent structures
typedef ExtendedDeviceID_t CallingDeviceID_t;

typedef ExtendedDeviceID_t CalledDeviceID_t;

typedef ExtendedDeviceID_t RedirectionDeviceID_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;
```

### CSTADivertedEvent structures

```
typedef struct CSTADivertedEvent_t {
    ConnectionID_t    connection;
    SubjectDeviceID_t divertingDevice;
    CalledDeviceID_t  newDestination;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTADivertedEvent_t;
```

### CSTAEstablishedEvent structures

```
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;
```

### **CSTAFailedEvent structures**

---

```
typedef struct CSTAFailedEvent_t {
    ConnectionID_t failedConnection;
    SubjectDeviceID_t failingDevice;
    CalledDeviceID_t calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAFailedEvent_t;
```

### **CSTAHeldEvent structures**

---

```
typedef struct CSTAHeldEvent_t {
    ConnectionID_t heldConnection;
    SubjectDeviceID_t holdingDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAHeldEvent_t;
```

### **CSTANetworkReachedEvent structures**

---

```
typedef struct CSTANetworkReachedEvent_t {
    ConnectionID_t connection;
    SubjectDeviceID_t trunkUsed;
    CalledDeviceID_t calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTANetworkReachedEvent_t;
```

### **CSTAOriginatedEvent structures**

---

```
typedef struct CSTAOriginatedEvent_t {
    ConnectionID_t originatedConnection;
    SubjectDeviceID_t callingDevice;
    CalledDeviceID_t calledDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAOriginatedEvent_t;
```

### **CSTAQueuedEvent structures**

---

```
typedef struct CSTAQueuedEvent_t {
    ConnectionID_t queuedConnection;
    SubjectDeviceID_t queue;
    CallingDeviceID_t callingDevice;
    CalledDeviceID_t calledDevice;
    RedirectionDeviceID_t lastRedirectionDevice;
    int numberQueued;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAQueuedEvent_t;
```

### **CSTARetrievedEvent structures**

---

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

### **CSTAServiceInitiatedEvent structures**

---

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

### **CSTATransferredEvent structures**

---

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

## **Feature Event Reports**

---

### **CSTACallInformationEvent structures**

---

```
typedef char          AccountInfo_t[32];

typedef char          AuthCode_t[32];

typedef struct CSTACallInformationEvent_t {
    ConnectionID_t   connection;
    SubjectDeviceID_t device;
    AccountInfo_t    accountInfo;
    AuthCode_t       authorisationCode;
} CSTACallInformationEvent_t;
```

### **CSTADoNotDisturbEvent structures**

---

```
typedef struct CSTADoNotDisturbEvent_t {
    SubjectDeviceID_t device;
    Boolean           doNotDisturbOn;
} CSTADoNotDisturbEvent_t;
```

### CSTAForwardingEvent structures

```
typedef struct CSTAForwardingEvent_t {
    SubjectDeviceID_t device;
    ForwardingInfo_t forwardingInformation;
} CSTAForwardingEvent_t;

typedef struct CSTAMessageWaitingEvent_t {
    CalledDeviceID_t deviceForMessage;
    SubjectDeviceID_t invokingDevice;
    Boolean          messageWaitingOn;
} CSTAMessageWaitingEvent_t;
```

## Agent Status Report Events

---

### CSTALoggedOnEvent structures

---

```
typedef struct CSTALoggedOnEvent_t {
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
    AgentGroup_t       agentGroup;
    AgentPassword_t    password;
} CSTALoggedOnEvent_t;
```

### CSTALoggedOffEvent structures

---

```
typedef struct CSTALoggedOffEvent_t {
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
    AgentGroup_t       agentGroup;
} CSTALoggedOffEvent_t;
```

### CSTANotReadyEvent structures

---

```
typedef struct CSTANotReadyEvent_t {
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
} CSTANotReadyEvent_t;
```

### CSTAReadyEvent structures

---

```
typedef struct CSTAReadyEvent_t {
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
} CSTAReadyEvent_t;
```

### CSTAWorkNotReadyEvent structures

---

```
typedef struct CSTAWorkNotReadyEvent_t {
    SubjectDeviceID_t agentDevice;
    AgentID_t          agentID;
} CSTAWorkNotReadyEvent_t;
```

### CSTAWorkReadyEvent structures

---

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

## Snapshot Services

---

### CSTASnapshotDeviceConfEvent structures

---

```
typedef struct CSTASnapshotDeviceData_t {
    int          count;
    struct CSTASnapshotDeviceResponseInfo_t *info;
} CSTASnapshotDeviceData_t;

typedef struct CSTASnapshotDeviceConfEvent_t {
    CSTASnapshotDeviceData_t snapshotData;
} CSTASnapshotDeviceConfEvent_t;
```

### CSTASnapshotCallConfEvent structures

---

```
typedef struct CSTASnapshotCallData_t {
    int          count;
    struct CSTASnapshotCallResponseInfo_t *info;
} CSTASnapshotCallData_t;

typedef struct CSTASnapshotCallConfEvent_t {
    CSTASnapshotCallData_t snapshotData;
} CSTASnapshotCallConfEvent_t;
```

## CSTASnapshotDeviceConfEvent structures

```
typedef enum CSTASimpleCallState_t {
    CALL_NULL = 0,
    CALL_PENDING = 1,
    CALL_ORIGINATED = 3,
    CALL_DELIVERED = 35,
    CALL_DELIVERED_HELD = 36,
    CALL_RECEIVED = 50,
    CALL_ESTABLISHED = 51,
    CALL_ESTABLISHED_HELD = 52,
    CALL_RECEIVED_ON_HOLD = 66,
    CALL_ESTABLISHED_ON_HOLD = 67,
    CALL_QUEUED = 83,
    CALL_QUEUED_HELD = 84,
    CALL_FAILED = 99,
    CALL_FAILED_HELD = 100
} CSTASimpleCallState_t;

typedef struct CSTACallState_t {
    int          count;
    LocalConnectionState_t *state;
} CSTACallState_t;

/* Used to take a CSTACallState_t which contains only two
 * LocalConnectionState_t and match them to the set of
 * CSTASimpleCallState_t above.
 */
#define SIMPLE_CALL_STATE(ccs)(ccs.stat[0] + (ccs.state[1] << 4))

typedef struct CSTASnapshotDeviceResponseInfo_t {
    ConnectionID_t callIdentifier;
    CSTACallState_t callState;
} CSTASnapshotDeviceResponseInfo_t;

typedef struct CSTASnapshotCallResponseInfo_t {
    SubjectDeviceID_t deviceOnCall;
    ConnectionID_t callIdentifier;
    LocalConnectionState_t localConnectionState;
} CSTASnapshotResponseInfoEvent_t;
```

## Computing Function Services

---

### **cstaRouteRegisterReq structures**

---

```
typedef struct CSTARouteRegisterReq_t {
    DeviceID_troutingDevice;
} CSTARouteRegisterReq_t;
```

### **cstaRouteRegisterReqConfEvent structures**

---

```
typedef long RegisterReqID_t;

typedef struct {
    RegisterReqID_tregisterReqID;
} CSTARouteRegisterReqConfEvent_t;
```

### **cstaRouteRegisterCancel structures**

---

```
typedef struct CSTARouteRegisterCancel_t {
    RegisterRegID_troutingRegID;
} CSTARouteRegisterCancel_t;
```

### **cstaRouteRegisterCancelConfEvent structures**

---

```
typedef struct {
    RegisterRegID_troutingRegID;
} CSTARouteRegisterCancelConfEvent_t;
```

## cstaRouteRequestEvent structures

```
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 {
    int          length;
    unsigned char *value;
} SetUpValues_t;

typedef struct CSTARouteRequestEvent_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
    DeviceID_t      currentRoute;
    DeviceID_t      callingDevice;
    ConnectionID_t  routedCall;
    SelectValue_t   routeSelAlgorithm;
    Boolean          priority;
    SetUpValues_t   setupInformation;
} CSTARouteRequestEvent_t;
```

## cstaRouteSelect structures

```
typedef int          RetryValue_t;
#define              noListAvailable -1
#define              noCountAvailable -2

typedef struct CSTARouteSelect_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
    DeviceID_t      routeSelected;
    RetryValue_t    remainRetry;
    SetUpValues_t   setupInformation;
    Boolean          routeUsedReq;
} CSTARouteSelect_t;
```

### **CSTAReRouteRequestEvent structures**

---

```
typedef struct CSTAReRouteEvent_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
} CSTAReRouteEvent_t;
```

### **cstaRouteUsedEvent structures**

```
typedef struct CSTARouteUsedEvent_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
    DeviceID_t routeUsed;
    DeviceID_t callingDevice;
    Boolean domain;
} CSTARouteUsedEvent_t;
```

### **cstaRouteEndEvent structures**

---

```
typedef struct CSTARouteEndEvent_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
    CSTAUniversalFailure_t errorValue;
} CSTARouteEndEvent_t;
```

### **cstaRouteEnd structures**

---

```
typedef struct CSTARouteEnd_t {
    RegisterReqID_t registerReqID;
    RoutingCrossRefID_t routingCrossRefID;
    CSTAUniversalFailure_t errorValue;
} CSTARouteEnd_t;
```

## Escape Services

---

### cstaEscapeService structures

---

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

### CSTAEscapeServiceConfEvent structures

---

```
typedef struct CSTAEscapeServiceConfEvent_t {
    Nulltype    null;
} CSTAEscapeServiceConfEvent_t;
PrivateEvent structures
typedef struct CSTAPrivateEvent_t {
    int         length;
    unsigned char *data;
} CSTAPrivateEvent_t;
```

### PrivateStatusEvent structures

---

```
typedef struct CSTAPrivateStatusEvent_t {
    Nulltypenull;
}
```

### cstaPrivateStatusEvent structures

---

```
typedef struct CSTAEscapeServiceEventConf_t
{
    UniversalFailure_terror;
};
```

### CSTAEscapeServiceEvent structures

---

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

### cstaEscapeServiceConf structures

---

```
typedef struct cstaEscapeServiceConf_t {
    CSTAUniversalFailure_terror;
};
```

### cstaSendPrivateEvent structures

---

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

## Maintenance Services

---

### CSTABackInServiceEvent structures

---

```
typedef struct CSTABackInServiceEvent_t {
    DeviceID_t      device;
    CSTAEventCause_t cause;
} CSTABackInServiceEvent_t;
```

### CSTAOutOfServiceEvent structures

---

```
typedef struct CSTAOutOfServiceEvent_t {
    DeviceID_t      device;
    CSTAEventCause_t cause;
} CSTAOutOfServiceEvent_t;
```

### cstaSysStatReq structures

---

```
typedef struct CSTASysStatReq_t (
    Nulltype      null;
) CSTASysStatReq_t
```

### CSTASysStatReqConfEvent structures

---

```
typedef enum SystemStatus_t {
    SS_INITIALIZING = 0,
    SS_ENABLED = 1,
    SS_NORMAL = 2,
    SS_MESSAGES_LOST = 3,
    SS_DISABLED = 4,
    SS_OVERLOAD_IMMINENT = 5,
    SS_OVERLOAD_REACHED = 6,
    SS_OVERLOAD_RELIEVED = 7
} SystemStatus_t;

typedef struct CSTASysStatReqConfEvent_t (
    SystemStatus_t systemStatus;
) CSTASysStatReqConfEvent_t
```

### cstaSysStatStart structures

```
typedef unsigned char  SystemStatusFilter_t;
#define SS_Initializing 0x800
#define SS_Enabled     0x400
#define SS_Normal      0x200
#define SS_MessageLost 0x100
#define SS_Disabled    0x080
#define SS_OverloadImminent0x040
#define SS_OverloadReached0x020
#define SS_OverloadRelieved0x010

typedef struct CSTASysStatStart_t (
SystemStatusFilter_t statusFilter;
) CSTASysStatStart_t;
```

### CSTASysStatStartConfEvent structures

```
typedef struct CSTASysStatStartConfEvent_t (
SystemStatusFilter_t systemFilter;
) CSTASysStatStartConfEvent_t
```

### cstaSysStatStop structures

```
typedef struct CSTASysStatStop_t (
Nulltype    null;
) CSTASysStatStop_t;
```

### CSTASysStatStopConfEvent structures

```
typedef struct CSTASysStatStopConfEvent_t (
Nulltype    null;
) CSTASysStatStopConfEvent_t
```

### cstaChangeSysStatFilter structures

```
typedef struct CSTAChangeSysStatFilter_t (
SystemStatusFilter_t statusFilter;
) CSTAChangeSysStatFilter_t;
```

### CSTAChangeSysStatFilterConfEvent structures

```
typedef struct CSTAChangeSysStatFilerConfEvent_t
{
SystemStatusFilter_t statusFilterSelected;
SystemStatusFilter_t statusFilterActive;
} CSTAChangeSysStatFilterConfEvent_t;
CSTASysStatEvent structures
typedef struct CSTASysStatEvent_t {
{
SystemStatus_t systemStatus;
};
};
```

### **CSTASysStatReqEvent structures**

---

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

### **cstaSysStatReqConf structures**

---

```
typedef struct CSTASysStatReqConf_t {  
    SystemStatus_tsystemStatus;  
} CSTASysStatReqConf_t;
```

### **cstaSysStatEventSend structures**

---

```
typedef struct CSTASysStatEventSend_t {  
    SystemStatus_tsystemStatus;  
} CSTASysStatEventSend_t;
```

## CSTA Control Services

---

### cstaGetAPICaps structures

---

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

### CSTAGetAPICapsConfEvent structures

---

```
typedef struct CSTAGetAPICapsConfEvent_t {
    int        alternateCall;
    int        answerCall;
    int        callCompletion;
    int        clearCall;
    int        clearConnection;
    int        conferenceCall;
    int        consultationCall;
    int        deflectCall;
    int        pickupCall;
    int        groupPickupCall;
    int        holdCall;
    int        makeCall;
    int        makePredictiveCall;
    int        queryMwi;
    int        queryDnd;
    int        queryFwd;
    int        queryAgentState;
    int        queryLastNumber;
    int        queryDeviceInfo;
    int        reconnectCall;
    int        retrieveCall;
    int        setMwi;
    int        setDnd;
    int        setFwd;
    int        setAgentState;
    int        transferCall;
    int        eventReport;
    int        callClearedEvent;
    int        conferencedEvent;
    int        connectionClearedEvent;
    int        deliveredEvent;
    int        divertedEvent;
    int        establishedEvent;
    int        failedEvent;
    int        heldEvent;
    int        networkReachedEvent;
    int        originatedEvent;
    int        queuedEvent;
    int        retrievedEvent;
    int        serviceInitiatedEvent;
    int        transferredEvent;
    int        callInformationEvent;
    int        doNotDisturbEvent;
    int        forwardingEvent;
    int        messageWaitingEvent;
```

Issue 1 — December 2002

```

int         loggedOnEvent;
int         loggedOffEvent;
int         notReadyEvent;
int         readyEvent;
int         workNotReadyEvent;
int         workReadyEvent;
int         backInServiceEvent;
int         outOfServiceEvent;
int         privateEvent;
int         routeRequestEvent;
int         reRoute;
int         routeSelect;
int         routeUsedEvent;
int         routeEndEvent;
int         monitorDevice;
int         monitorCall;
int         monitorCallsViaDevice;
int         changeMonitorFilter;
int         monitorStop;
int         monitorEnded;
int         snapshotDeviceReq;
int         snapshotCallReq;
int         escapeService;
int         privateStatusEvent;
int         escapeServiceEvent;
int         escapeServiceConf;
int         sendPrivateEvent;
int         sysStatReq;
int         sysStatStart;
int         sysStatStop;
int         changeSysStatFilter;
int         sysStatReqEvent;
int         sysStatReqConf;
int         sysStatEvent;
} CSTAGetAPICapsConfEvent_t;

```

### **cstaGetDeviceList structures**

---

```

typedef enum CSTALevel_t {
    CSTA_LEVEL1 = 1,
    CSTA_LEVEL2 = 2,
    CSTA_LEVEL3 = 3,
    CSTA_LEVEL4 = 4,
    CSTA_LEVEL5 = 5,
    CSTA_LEVEL6 = 6
} CSTALevel_t;

```

### **CSTAGetDeviceListConfEvent structures**

---

```
typedef struct CSTAGetDeviceListConfEvent_t
{
Level_t      acsLevelReq;
int          totalDevices;
DeviceID_t* deviceIDs;
} CSTAGetDeviceListConfEvent_t;
```

### **CSTA Event Structures**

---

```
CSTA event types
#define CSTAREQUEST 3
#define CSTAUNSOLICITED4
#define CSTACONFIRMATION5
#define CSTAEVENTREPORT6
```

### **CSTA Request Event structure**

---

```
typedef struct
{
InvokeID_t invokeID;
union
{
CSTARouteRequestEvent_t routeRequest;
CSTAReRouteRequest_t reRouteRequest;
CSTAEscapeSvcReqEvent_t escapeSvcRequest;
CSTASysStatReqEvent_t sysStatRequest;
} u;
} CSTARequestEvent;
```

### **CSTA Event Report structure**

---

```
typedef struct
{
union
{
CSTARouteRegisterAbortEvent_t registerAbort;
CSTARouteUsedEvent_t routeUsed;
CSTARouteEndEvent_t routeEnd;
CSTAPrivateEvent_t privateEvent;
CSTASysStatEvent_t sysStat;
CSTASysStatEndedEvent_t sysStatEnded;
}u;
} CSTAEventReport;
```

**CSTA Unsolicited Event structure**

---

```
typedef struct
{
    CSTAMonitorCrossRefID_tmonitorCrossRefId;
    union
    {
        CSTACallClearedEvent_tcallCleared;
        CSTAConferencedEvent_tconferenced;
        CSTAConnectionClearedEvent_tconnectionCleared;
        CSTADeliveredEvent_tdelivered;
        CSTADivertedEvent_tdiverted;
        CSTAEstablishedEvent_testablished;
        CSTAFailedEvent_tfailed;
        CSTAHeldEvent_theld;
        CSTANetworkReachedEvent_tnetworkReached;
        CSTAOriginatedEvent_toriginated;
        CSTAQueuedEvent_tqueued;
        CSTARetrievedEvent_tretrieved;
        CSTAServiceInitiatedEvent_tserviceInitiated;
        CSTATransferredEvent_ttransferred;
        CSTACallInformationEvent_tcallInformation;
        CSTADoNotDisturbEvent_tdoNotDisturb;
        CSTAForwardingEvent_tforwarding;
        CSTAMessageWaitingEvent_tmessageWaiting;
        CSTALoggedOnEvent_tloggedOn;
        CSTALoggedOffEvent_tloggedOff;
        CSTANotReadyEvent_tnotReady;
        CSTAReadyEvent_tready;
        CSTAWorkNotReadyEvent_tworkNotReady;
        CSTAWorkReadyEvent_tworkReady;
        CSTABackInServiceEvent_tbackInService;
        CSTAOutOfServiceEvent_toutOfService;
        CSTAPrivateStatusEvent_tprivateStatus;
        CSTAMonitorEndedEvent_t monitorEnded;
    } u;
} CSTAUnsolicitedEvent;
```

## CSTA Confirmation Event structure

```

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;
  } u;
} CSTAConfirmationEvent;

```

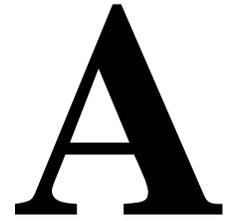
### CSTA Event\_t structure

```
typedef struct
{
  ACSEventHeader_t eventHeader;
  union
  {
    ACSUnsolicitedEvent_t acsUnsolicited;
    ACSConfirmationEvent_t acsConfirmation;
    CSTARequestEvent_t cstaRequest;
    CSTAUnsolicitedEvent_t cstaUnsolicited;
    CSTAConfirmationEvent_t cstaConfirmation;
    CSTAEventReport_t cstaEventReport;
  } event;
  char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```



---

## References



- 
- |     |  |
|-----|--|
| [1] | ECMA - Standard ECMA-179, Services for Computer-Supported Telecommunications Applications (CSTA) - June 1992 |
| [2] | ECMA - Standard ECMA-180, Protocol for Computer-Supported Telecommunications Applications (CSTA) - June 1992 |



---

## Numerics

1, 8-1

---

## A

ACD, 3-1, 3-2, 3-3  
 Agent, 3-1  
 ACS, 4-1  
   Unsolicited Events, 4-43  
 ACS Data Types, 4-53  
   Common, 4-54  
   Event, 4-56  
 ACS stream  
   Aborting, 4-2, 4-3  
   Access, 4-2  
   Checking establishment of, 4-2  
   Closing, 4-2, 4-3  
   CSTA services available on, 4-1  
   Freeing associated resources, 4-3, 4-4  
   Multiple, 4-2  
   Opening, 4-1, 4-2  
   Per advertised service, 4-2  
   Receiving events on, 4-3  
   Releasing, 4-3  
   Sending requests and responses over, 4-4  
 acsAbortStream(), 4-22  
 acsAbortStream(), 4-3, 4-4  
 acsCloseStream(), 4-17  
 acsCloseStream(), 4-3, 4-4  
 ACSCloseStreamConfEvent, 4-3, 4-19  
 acsEnumServerNames(), 4-38  
 acsEventNotify()  
   Windows 3.1, 4-32, 4-35  
 acsFlushEventQueue(), 4-37  
 acsGetEventBlock(), 4-5, 4-23  
 acsGetEventPoll(), 4-5, 4-25  
 acsGetFile() (UnixWare), 4-27  
 acsHandle, 4-2, 4-3, 4-4, 4-5  
   Freeing, 4-3  
 acsOpenStream(), 4-2, 4-9, 4-41, 4-42, 4-43  
 ACSOpenStreamConfEvent, 4-2, 4-4, 4-15  
 acsQueryAuthInfo(), 4-39, 4-40, 4-41, 4-42  
 acsSetESR(), 4-6  
   Windows, 4-28, 4-30  
 ACSUniversalFailureConfEvent, 4-20  
 ACSUniversalFailureEvent, 4-43  
   Driver errors, 4-52  
   Possible values, 4-45  
   Security database errors, 4-50  
   Tserver operation errors, 4-47  
 Administration, 1-2, 2-4, 4-2  
 advertised service, 8-2  
 Advertised services

Getting list of available, 4-1  
 Agent Feature Event Reports  
   See Feature Event (Agent), 6-63  
 Agent feature event reports, 6-63  
 API Control Services  
   See ACS, 4-1  
 Application Call Routing, 8-1, 8-2  
 Application domain, 3-1  
 Application Programming Interface Control Services  
   See ACS, 4-1  
 Applications, 1-1, 1-2, 2-2, 2-5, 3-1, 3-3, 3-4, 3-5, 3-6, 3-7,  
   3-8, 3-10, 3-11, 3-12, 4-1  
 Architecture, 1-1, 1-2, 2-2, 3-4  
 Attendant, 3-7  
 Automatic Call Distribution  
   See ACD, 3-1

---

## B

Basic Call Control  
   Services, 5-1  
 Basic Call Control Services, 5-1

---

## C

Call, 3-2, 3-3, 3-6, 3-10, 3-11  
   Active, 3-1  
   Alerting, 3-1  
   Basic, 3-1  
   Complex, 3-2  
   Held, 3-3  
   Identifier, 3-2, 3-10, 3-11, 3-15  
   Management, 3-15  
   Uniqueness, 3-10  
   State, 3-10, 3-13  
     Common, 3-10  
     Compound, 3-13, 3-14  
     Simple, 3-14  
 Call control service, 1-2  
 Call monitoring service, 1-2  
 Call origination, 5-47  
 Call routing service  
   See Routing, 1-2  
 Call Snapshot Services, 7-3  
 Call status event reports  
   See Event reports, 3-12  
 Call-type monitor, 6-2  
 Cause code definitions, 6-76  
 Client, 3-4, 3-5  
   Library, 2-4  
   Supported, 1-2  
 Client/server model, 3-4  
 Communications relationship  
   See Call, 3-2

- Computer Telephony Integration
  - See CTI, 1-1
- Computer-Supported Telecommunications Applications
  - See CSTA, 1-2
- Computing
  - Domain, 3-2, 3-3, 3-6
  - Function, 3-1, 3-2, 3-3, 3-4, 3-6, 3-8
  - Sub-domain, 3-1, 3-2, 3-3
- Computing Function Services, 8-1
- Connection, 3-2, 3-3, 3-6, 3-11, 3-12
  - Attributes, 3-11
  - Identifier, 3-2, 3-10, 3-11, 3-15
    - In event report, 3-15
    - Invalid, 3-15
    - Reuse, 3-15
    - Uniqueness, 3-11, 3-15
    - Update, 3-15
  - State, 3-2, 3-3, 3-10, 3-11, 3-12, 3-13, 3-14
    - Alerting, 3-12
    - Connected, 3-12
    - Failed, 3-12
    - Held, 3-12
    - Initiated, 3-12
    - Local, 3-14
    - Null, 3-12
    - Queued, 3-12
- CSTA, 1-2, 2-2, 3-1, 3-3, 3-4, 3-12
  - Confirmation Events, 4-57
  - Control Services, 4-1, 4-57
  - Event Data Types, 4-68
  - Services, 4-2
    - Available on ACS Stream, 5-2
    - Available on ACS stream, 4-1
- CSTA Universal Failure
  - Conference Member Limit Exceeded., 5-9
  - CSTA Driver Interface Errors, 5-11
  - Duplicate Invocation, 5-6
  - External Trunk Limit Exceeded, 5-9
  - Generic Operation, 5-6
  - Generic Operation Rejection, 5-6
  - Generic Performance Management Error, 5-9
  - Generic State Incompatibility, 5-7
  - Generic Subscribed Resource Availability Error, 5-9
  - Generic System Resource Availability Error, 5-8
  - Incorrect Object State, 5-7
  - Invalid Allocation State, 5-7
  - Invalid Called Device, 5-6
  - Invalid Calling Device, 5-6
  - Invalid Cross Ref ID, 5-7
  - Invalid CSTA Call Identifier, 5-6
  - Invalid CSTA Connection Identifier, 5-7
  - Invalid CSTA Connection Identifier For Active Call, 5-7
  - Invalid CSTA Device Identifier, 5-7
  - Invalid Destination, 5-7
  - Invalid Feature, 5-7
  - Invalid Forwarding Destination, 5-6
  - Invalid Object Type, 5-7
  - Network Busy, 5-8
  - Network Out Of Service, 5-8
  - No Active Call, 5-7
  - No Call To Answer, 5-8
  - No Call To Clear, 5-8
  - No Call To Complete, 5-8
  - No Connection To Clear, 5-8
  - No Held Call, 5-8
  - Object Monitor Limit Exceeded, 5-9
  - Object not Known, 5-6
  - Operation errors, 5-6
  - Overall Monitor Limit Exceeded., 5-8
  - PAC Violated, 5-10
  - Performance Limit Exceeded, 5-9
  - Performance management errors, 5-9
  - Private Data errors, 5-12
  - Privilege Violation on Called Device, 5-6
  - Privilege Violation on Calling Device, 5-6
  - Privilege Violation on Specified Device, 5-6
  - Request Incompatible with Object, 5-6
  - Resource Busy, 5-8
  - Resource Out Of Service, 5-8
  - Seal Violated, 5-11
  - Security errors, 5-9
  - Security Violation, 5-7
  - Sequence Number Violated, 5-9
  - Service Busy, 5-8
  - State incompatibility errors, 5-7
  - Subscribed resource availability errors, 5-9
  - System resource availability errors, 5-8
  - Time Stamp Error, 5-9
  - TSAPI errors, 5-12
  - Unrecognized Operation, 5-6
  - Unspecified errors, 5-4
  - Unspecified Security Error, 5-9
  - Value Out Of Range, 5-6
  - cstaAlternateCall(), 5-12
  - CSTAAAlternateCallConfEvent, 5-14
  - cstaAnswerCall(), 5-16
  - CSTAAnswerCallConfEvent, 5-17
  - CSTABackInServiceEvent, 9-17
  - cstaCallClearedEvent, 6-21
  - cstaCallCompletion(), 5-19
  - CSTACallCompletionConfEvent, 5-20
  - CSTACallInfoEvent, 6-55
  - cstaChangeMonitorFilter(), 6-15
  - CSTAChangeMonitorFilterConfEvent, 6-18
  - cstaChangeSysStatFilter(), 9-33
  - CSTAChangeSysStatFilterConfEvent, 9-35
  - cstaClearCall(), 5-22
  - CSTAClearCallConfEvent, 5-23
  - cstaClearConnection(), 5-26
  - CSTAClearConnectionConfEvent, 5-28
  - cstaConferenceCall(), 5-29
  - CSTAConferenceCallConfEvent, 5-31
  - cstaConferencedEvent, 6-24
  - CSTAConnectionClearedEvent, 6-27
  - cstaConsultationCall(), 5-33
  - CSTAConsultationCallConfEvent, 5-35

- cstaDeflectCall(), 5-37
- CSTADeflectCallConfEvent, 5-38
- CSTADeliveredEvent, 6-29
- CSTADivertedEvent, 6-32
- CSTADoNotDisturbEvent, 6-57
- cstaEscapeService(), 9-5
- cstaEscapeServiceConf(), 9-12
- CSTAEscapeServiceConfEvent, 9-7
- CSTAEscapeServiceReq, 9-11
- CSTAEstablishedEvent, 6-34
- CSTAEventCause\_t, 6-75
- CSTAFailedEvent, 6-36
- CSTAForwardingEvent, 6-58
- cstaGetAPICaps, 5-2
- cstaGetAPICaps(), 4-57
- CSTAGetAPICapsConfEvent, 4-59
- cstaGetDeviceList(), 4-61
- CSTAGetDeviceListConfEvent, 4-63
- cstaGroupPickupCall(), 5-40
- CSTAGroupPickupCallConfEvent, 5-42
- CSTAHeldEvent, 6-38
- cstaHoldCall(), 5-43
- CSTAHoldCallConfEvent, 5-45
- CSTALoggedOffEvent, 6-65
- CSTALoggedOnEvent, 6-63
- cstaMakeCall(), 5-47
- CSTAMakeCallConfEvent, 5-50
- cstaMakePredictiveCall(), 5-51
- CSTAMakePredictiveCallConfEvent, 5-54
- CSTAMessageWaitingEvent, 6-61
- cstaMonitorCall(), 6-6
- cstaMonitorCallsViaDevice(), 6-8
- CSTAMonitorConfEvent, 6-10
- cstaMonitorDevice(), 6-3
- CSTAMonitorEndedEvent, 6-19
- CSTAMonitorFilter\_t, 6-74
- cstaMonitorStop(), 6-12
- CSTAMonitorStopConfEvent, 6-14
- CSTANetworkReachedEvent, 6-41
- CSTANotReadyEvent, 6-66
- CSTAOriginatedEvent, 6-43
- CSTAOutOfServiceEvent, 9-18
- cstaPickupCall(), 5-55
- CSTAPickupCallConfEvent, 5-57
- CSTAPrivateEvent, 9-8
- CSTAPrivateStatusEvent, 9-9
- cstaQueryAgentState(), 5-93
- CSTAQueryAgentStateConfEvent, 5-95
- cstaQueryCallMonitor(), 4-66
- CSTAQueryCallMonitorConfEvent, 4-67
- cstaQueryDeviceInfo(), 5-100
- CSTAQueryDeviceInfoConfEvent, 5-102
- CSTAQueryDndConfEvent, 5-87
- cstaQueryDoNotDisturb(), 5-86
- cstaQueryFwd (), 5-89
- CSTAQueryFwdConfEvent, 5-90
- cstaQueryLastNumber(), 5-97
- CSTAQueryLastNumberConfEvent, 5-98
- cstaQueryMsgWaitingInd(), 5-83
- CSTAQueryMwiConfEvent, 5-85
- CSTAQueuedEvent, 6-45
- CSTARReadyEvent, 6-68
- cstaReconnectCall(), 5-58
- CSTARReconnectCallConfEvent, 5-61
- CSTAReRouteRequestEvent, 8-18
- cstaRetrieveCall(), 5-63
- CSTARRetrieveCallConfEvent, 5-64
- CSTARetrievedEvent, 6-48
- cstaRouteEnd(), 8-29
- CSTARRouteEndEvent, 8-27
- cstaRouteEndInv(), 8-32
- CSTARRouteRegisterAbortEvent, 8-11
- cstaRouteRegisterCancel(), 8-8
- CSTARRouteRegisterCancelConfEvent, 8-10
- cstaRouteRegisterReq(), 8-4
- CSTARRouteRegisterReqConfEvent, 8-6
- CSTARRouteRequestEvent, 8-15
- cstaRouteSelect(), 8-20
- cstaRouteSelectInv(), 8-23
- CSTARRouteUsedEvent, 8-25
- cstaSendPrivateEvent(), 9-14
- CSTAServiceInitiatedEvent, 6-50
- cstaSetAgentState(), 5-80
- CSTASetAgentStateConfEvent, 5-82
- CSTASetDndConfEvent, 5-75
- cstaSetDoNotDisturb (), 5-73
- cstaSetForwarding(), 5-76
- CSTASetFwdConfEvent, 5-78
- cstaSetMsgWaitingInd(), 5-70
- CSTASetMwiConfEvent, 5-72
- CSTASnapshotCallConfEvent, 7-5
- cstaSnapshotCallReq(), 7-4
- CSTASnapshotDeviceConfEvent, 7-9
- cstaSnapshotDeviceReq(), 7-7
- cstaSysStatEndedEvent, 9-39
- CSTASysStatEvent, 9-37
- cstaSysStatEventSend(), 9-44
- cstaSysStatReq(), 9-20
- cstaSysStatReqConf(), 9-43
- CSTASysStatReqConfEvent, 9-22
- CSTASysStatReqEvent, 9-40
- cstaSysStatStart(), 9-26
- CSTASysStatStartConfEvent, 9-28
- cstaSysStatStop(), 9-30
- CSTASysStatStopConfEvent, 9-32
- cstaTransferCall(), 5-66
- CSTATransferCallConfEvent, 5-68
- CSTATransferredEvent, 6-52
- CSTAUniversalFailureConfEvent, 5-2
- CSTAWorkNotReadyEvent, 6-69
- CSTAWorkReadyEvent, 6-71
- CTI, 1-1
  - Link, 2-3, 2-4, 3-2, 3-3, 3-5, 4-2
  - Link hardware, 2-3, 2-4
  - CTI link, 8-2

**D**

## Data Types

ACS, 4-53

## Default routing server

See Routing (Default Routing Server), 8-14

default routing server, 8-2, 8-3, 8-5

Device, 3-2, 3-3, 3-7, 3-10, 3-11, 3-12, 3-13

Attribute, 3-7

Class, 3-8

Data, 3-8

Image, 3-8

Other, 3-8

Voice, 3-8

Identifier, 3-2, 3-8, 3-10, 3-11, 3-15

Dynamic, 3-8, 3-10, 3-15

Invalid, 3-15

Management, 3-15

Reuse, 3-15

Static, 3-8

Static Short Form, 3-8

Uniqueness, 3-9, 3-15

Update, 3-15

Monitoring service, 1-2

## Query

For controllable devices, 4-1

State, 3-10

State change, 3-2

Type, 3-7

ACD, 3-7

ACD group, 3-7

Button, 3-7

Button group, 3-7

Line, 3-7

Line group, 3-7

Operator, 3-7

Operator group, 3-7

Station, 3-7

Station group, 3-8

Trunk, 3-8

Trunk group, 3-8

Device Snapshot Service, 7-7

Device-type monitor, 6-2

Directory number, 3-2

Distribution, 3-4

Domain, 3-2, 3-6

Driver ACS Handle Rejection, 4-53

Driver ACSHandle Termination, 4-53

Driver errors, 4-52

ACS Handle Rejection, 4-53

ACSHandle Termination, 4-53

Duplicate ACSHandle, 4-52

Generic Rejection, 4-53

Invalid ACS Request, 4-52

Invalid Class Rejection, 4-53

Link Unavailable, 4-53

OAM In Use, 4-53

Resource Limitation, 4-53

**E**

ECMA, 1-2

Address, 2-2

## Error

## Driver

See Driver Errors, 4-52

## Tserver

See TServer Errors, 4-49

Tserver Bad Device Record, 4-52

Tserver Bad Password Or Login, 4-50

Tserver Bad SDB Level, 4-50

Tserver Bad Server ID, 4-50

Tserver Bad Stream Type, 4-50

Tserver Device Not On List, 4-52

Tserver Device Not Supported, 4-52

Tserver Exception List, 4-52

Tserver Insufficient Permission, 4-52

Tserver No Away Permission, 4-52

Tserver No Away Worktop, 4-52

Tserver No Device Record, 4-51

Tserver No Home Permission, 4-52

Tserver No SDB, 4-50

Tserver No SDB Check Needed, 4-50

Tserver No User Record, 4-50

Tserver SDB Check Needed, 4-50

Tserver Users Restricted Home, 4-52

## Escape Service

Application as Client, 9-5

Driver/Switch as the Client, 9-11

European Computer Manufacturers Association

See ECMA, 1-2

## Event

Cause Relationships, 6-78

Data Types (Unsolicited), 6-74

Service Routine (ESR), 4-6

Also see acsSetESR, 4-1

Initializing, 4-1

Unsolicited, 5-70

Event report, 3-3, 3-5, 3-12

Events, 3-2, 3-3, 4-4, 6-21

Asynchronous, 6-1

Blocking for, 4-1, 4-5

Chronological order, 4-5

Confirmation, 5-1

From all streams, 4-5

Polling for, 4-1, 4-5

Preventing queue overflow, 4-6

Unsolicited, 5-2, 6-1, 6-3

---

## F

Feature  
  Event  
    Agent, 6-63  
    Unsolicited, 6-55  
Feature event, 6-55  
Feature state, 6-55

---

## G

Generic State Incompatibility, 5-8

---

## H

Handle  
  Register Request ID, 8-14  
  Routing Cross-Reference ID, 8-14  
Hold call request, 5-45

---

## I

Identifier  
  See TSAPI Programming Handle, 3-3  
InvokeID  
  Correlating responses, 5-1  
Integration, 1-1, 1-2  
Interconnection Service Boundary, 3-3  
InvokeID  
  Application generated, 4-4, 5-1  
  Correlating responses, 4-4  
  In confirmation event, 4-4  
  In service request, 4-4  
  Library generated, 4-4, 5-1  
  Type, 4-4  
ISDN, 3-4, 3-12

---

## L

Logical  
  Association, 1-1  
  Device, 3-7  
  Link, 4-2

---

---

## M

Maintenance Services, 9-3, 9-16

---

## N

NetWare NOS, 2-1

---

## O

Object  
  See TSAPI programming object, 3-2  
Outstanding Requests Limit Exceeded, 5-9

---

## P

Party, 3-3  
PBX driver, 8-2  
PBX Drivers, 4-2  
Permissions  
  See Administration, 1-2  
Programming handle  
  See TSAPI programming handle, 3-2

---

## Q

Query  
  Call/Call Monitoring, 4-1  
Query service, 1-2

---

## R

Register Request ID, 8-14  
Route  
  Invalid, 8-3  
  Request, 8-3  
  Re-route, 8-3, 8-14  
  successful route, 8-3  
  Used, 8-14  
routeRegisterReqID  
  Duration, 8-14  
  Uniqueness, 8-14  
Routing  
  Actual destination, 8-25  
  Cancel, 8-8, 8-11

---

- Cross-reference identifier, 8-2
- Default routing server, 8-14
- Device, 8-2, 8-14
- Events, 8-14
- Functions, 8-14
- Switch default route, 8-3
- Terminating, 8-3
- Routing Cross-Reference ID, 8-14
- Routing device, 8-2
- routing device, 8-5
- Routing Procedure, 8-2
- Routing Registration, 8-2
- Routing Registration Functions and Events, 8-4
- Routing service, 1-2
- routingCrossRefID
  - Duration, 8-14
  - Uniqueness, 8-14

## S

- Security
  - See Administration, 1-2
- Server, 3-4, 3-5
  - Library, 2-4
  - Supported, 1-2, 2-2
- Service, 3-3, 3-5
  - Boundary, 3-3
  - Description, 3-5
- Services
  - Library, 2-4
- State
  - See Call State, 3-3
  - See Connection State, 3-3
  - See Device State, 3-3
- Status Reporting
  - Confirmation Events, 6-1
  - Functions, 6-1
- Switch
  - Driver, 2-3, 2-4
    - Interface, 2-3, 2-4
  - Independent, 1-1, 1-2, 2-3, 2-4, 3-4, 3-11
  - Specific, 1-1, 1-2, 2-3, 2-4, 3-5, 3-11, 4-2, 8-14, 8-19, 8-22, 8-24
- Switching
  - Domain, 3-2, 3-3, 3-6
  - Function, 3-1, 3-2, 3-3, 3-4, 3-5, 3-6, 3-7, 3-8, 3-10, 3-15
  - Sub-domain, 3-1, 3-3, 3-5, 3-8, 3-10, 3-11, 3-12, 3-15
  - Model, 3-6
- Switching Function Services
  - Basic Call Control Services, 5-1
- SwitchingFunctionServices
  - TelephonySupplementaryServices, 5-1
- System status
  - Application as the client, 9-20
  - Cause codes, 9-3
  - Driver/switch as the client, 9-40

## T

- Telephony Supplementary Services, 5-1, 5-70
- Trunk group, 3-2
- TSAPI
  - Call States
    - See Call State, 3-13
  - Programming handle
    - See Call Identifier, 3-2
    - See Connection Identifier, 3-2
    - See Device Identifier, 3-2
  - Programming object, 3-4, 3-5, 3-6
    - See Call, 3-2
    - See Connection, 3-2
    - See Device, 3-2
- Tserver Errors
  - Bad Connection, 4-48
  - Bad Device Record, 4-52
  - Bad Driver ID, 4-47
  - Bad Driver Protocol, 4-49
  - Bad Password Encryption, 4-49
  - Bad Password Or Login, 4-50
  - Bad PDU, 4-48
  - Bad Protocol, 4-49
  - Bad Protocol Format, 4-49
  - Bad SDB Level, 4-50
  - Bad Server ID, 4-50
  - Bad Stream Type, 4-50
  - Bad Transport Type, 4-49
  - Dead Driver, 4-48
  - Decode Failed, 4-48
  - Device Not On List, 4-52
  - Device Not Supported, 4-52
  - ECB Max Exceeded, 4-48
  - ECB Overdue, 4-49
  - Encode Failed, 4-48
  - Exception List, 4-52
  - Free Buffer Failed, 4-48
  - Insufficient Permission, 4-52
  - Invalid Message, 4-49
  - Message High Water Mark, 4-48
  - No Away Permission, 4-52
  - No Away Worktop, 4-52
  - No Device Record, 4-51
  - No ECBS, 4-49
  - No Home Permission, 4-52
  - No Memory, 4-48
  - No Resource Tag, 4-49
  - No SDB, 4-50
  - No SDB Check Needed, 4-50
  - No Thread, 4-47
  - No User Record, 4-50
  - No Version, 4-48
  - Old Tslib, 4-49
  - PDU Version Mismatch, 4-49
  - Receive From Driver, 4-48

Registration Failed, 4-48  
SDB Check Needed, 4-50  
Send To Driver, 4-48  
Stream Failed, 4-47  
Trace, 4-48  
Users Restricted Home, 4-52

---

## U

Unsolicited Events, 4-43  
User, 3-3

