DMS-100 NetWare Telephony Services

Programmer's Guide

NORTEL

A World of Networks

Contents

About this guide

This chapter defines the purpose, scope, and audience of the *DMS-100 NetWare™ Telephony Services™Programmer's Guide.* A list of related reading material for application developers, system engineers, and system administrators is also provided.

Purpose

The purpose of this guide is to provide detailed information about NetWare Telephony Services (NTS) and the application programming interface (TSAPI) for the DMS-100 Switch**.**

Intended audience

This guide is written for the application developer responsible for programming customized Computer-Supported Telecommunications Applications (CSTA) applications for the DMS-100 Switch in an NTS environment. To use this document, application developers, system engineers, and system administrators should have a basic understanding of telephony, European Computer Manufacturers Association (ECMA) and CSTA services, protocol definitions, and DMS-100 features and operations. Refer to the following "Related reading" section for information on documents on these topics.

Related reading

This guide provides DMS-100-specific information for NTS and is written as a companion to Novell's *Telephony Services Application Programming Interface (TSAPI).* For this reason, specific TSAPI chapters are listed below according to audience needs. Also listed are documents containing DMS-100-specific information and documents containing Computer Telephony Integration (CTI) standards information.

RELATED READING FOR A DMS-100 APPLICATION DEVELOPER

The *Telephony Services Application Programming Interface* is published by Novell Inc. This document details the interface specification for the NetWare Telephony Server (Tserver) architecture on the NTS platform. It also describes the CSTA-based API for call control, call/device monitoring and query, call routing, and device/system maintenance capabilities. This document is required reading for developers designing CSTA applications, and for Switch customers and Switch vendors supporting CSTA applications for DMS-100.

Below is a list of relevant TSAPI chapter titles and their content:

- Chapter 1 "Abstract" provides a general overview of server-based technology and its benefits.
- Chapter 2 "Introduction" describes TSAPI's purpose and its architecture. Hardware and software components for Telephony Services are listed and defined.
- Chapter 3 "Function Call Model" gives the ECMA CSTA standard concepts for TSAPI application programming.
- Chapter 4 "Control Services" describes Application Programming Interface (API) Control Services (ACS) associated with the call control capabilities (CSTA-based services) supported by NTS.
- Chapter 5 "Switch Function Services" describes telephony services that operate on calls and activate switch related features that are associated with the user desktop telephone or any other device defined by the switching domain. Switch Function Services are divided into Basic Call Control Services and Telephony Supplementary Services.
- Chapter 6 "Status Reporting Services" describes the status reporting services that are available from the API. Function calls and events associated with unsolicited event messages are also addressed.
- Chapter 7 "CSTA Data Types" describes the data types (CSTA Data Type and Interface Data Types) used by the functions and messages defined for the TSAPI.

Other useful documents include the following:

- *Telephony Server Application Programming Interface (TSAPI),* Release 2, Novell Inc.
- *Novell Software Developers Kit: NetWare Btrieve Programmer's Manual,* April 1994, Part No. 100-001341-002.
- *NLM Certification Testing Tick Sheets and Instructions,* Version 1.0, August 24, 1994, Novell Labs Software Services.
- *Novell's NLM Certification Testing Tick Sheets and Instructions,* Version1.0.
- *NetWare Telephony Services, ????? Driver Profiles,* August 18, 1994, Version 0.8, Oliver Tavakoli, Novell Advanced Services.
- *DMS-100 CompuCALL/SL-100 Meridian SCAI Interface Specification,* Release 05.01, July 1993, Nortel, NIS-Q218-1.

RELATED READING FOR A DMS-100 SYSTEMS ENGINEER

Systems engineers should be familiar with telephony and with the operation and architecture of telephony services for DMS-100. Refer to chapters 1, 2, and 3 of this guide for basic information about CSTA and telephony services for DMS-100.

RELATED READING FOR A DMS-100 SYSTEM ADMINISTRATOR

The *DMS-100 NetWare Telephony Services Network Manager's Guide* provides descriptions of the hardware and software requirements as well as the parameters for NTS.

ECMA DOCUMENTS

ECMA standards do not provide information on the platforms supporting the programming interfaces and telecommunication integrations described in these documents. Novell's *Telephony Services Application Programming Interface (TSAPI)*, Issue 1.9, provides the necessary platform and API specifications that are not supplied by the CSTA standard. Also, some services or parameters defined in the CSTA Standards are not meaningful to the DMS-100 Switch Communications System environment. However, some service requests described in the ECMA documents have corresponding CSTA service functions defined in *Telephony Services Application Programming Interface.* Each definition specifies the generic CSTA application program syntax and interface for NTS. In this context, the term "generic" means applicable to all switch vendors.

- *STANDARD ECMA-179 Services For Computer-Supported Telecommunications Applications (CSTA)*, European Computer Manufacturers Association (June 1992), defines services for CSTA for the OSI Layer 7 communication between a computing network and a telecommunications network. This standard, plus its companion, Standard ECMA-180, reflects agreements of ECMA member companies on the first phase of standards for CSTA.
- *STANDARD ECMA-180 Protocol For Computer-Supported Telecommunications Applications (CSTA)*, European Computer Manufacturers Association (June 1992), defines a protocol for CSTA for the OSI Layer 7 communication between a computing network and a telecommunications network.
- *Computer-Supported Telecommunications Applications ECMA TR/52,* European Computer Manufacturers Association defines the foundation for the OSI Layer 7 service Protocol Message Interface communication between computing applications and switching applications.

To obtain a copy of these documents, write to

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

Purpose of CSTA

The Computer-Supported Telecommunications Applications (CSTA) standard provides an architectural framework that enables switch and computer vendors/customers to enhance network capabilities using Computer Telephony Integration (CTI) technology. In this way, cooperating networks support new applications without the need for significant enhancement or redesign. The goal of CSTA is to enhance the individual capabilities of cooperating computing and switching networks. Under CSTA, cooperating networks, such as a general purpose computer and a voice-switch network, can incorporate the services provided by one another. Neither system could implement all the capabilities feasible under CTI technology. A computer application running on a PC can control call distribution by integrating voice and data transmissions on the switch network. In turn, the switch network runs an application that utilizes a database management system available on the general purpose computer.

CSTA Standard and API Specification

Telephony Services Application Programming Interface (TSAPI) is based on the European Computer Manufacturers Association (ECMA) CTI definition of CSTA. The CSTA standard is a technical agreement reached by an open, multi-vendor consortium of major European switch and computer vendors. These standards are as follows:

- *Standard ECMA-179* defines the CSTA services standard.
- *Standard ECMA-180* defines the application protocol data units for the CSTA services.

These documents form the basis of the CSTA implementation by all switch vendors and provide abstract descriptions of the design and implementation of telephony services. They provide CSTA application developers with the necessary information to enable all switch and computer vendors to support the same set of telephony services, regardless of platform type. Since CSTA Services and Protocol definitions are the basis for TSAPI, *Telephony Services Application Programming Interface (TSAPI)* provides the necessary platform and API specifications that are not supplied by the CSTA standard.

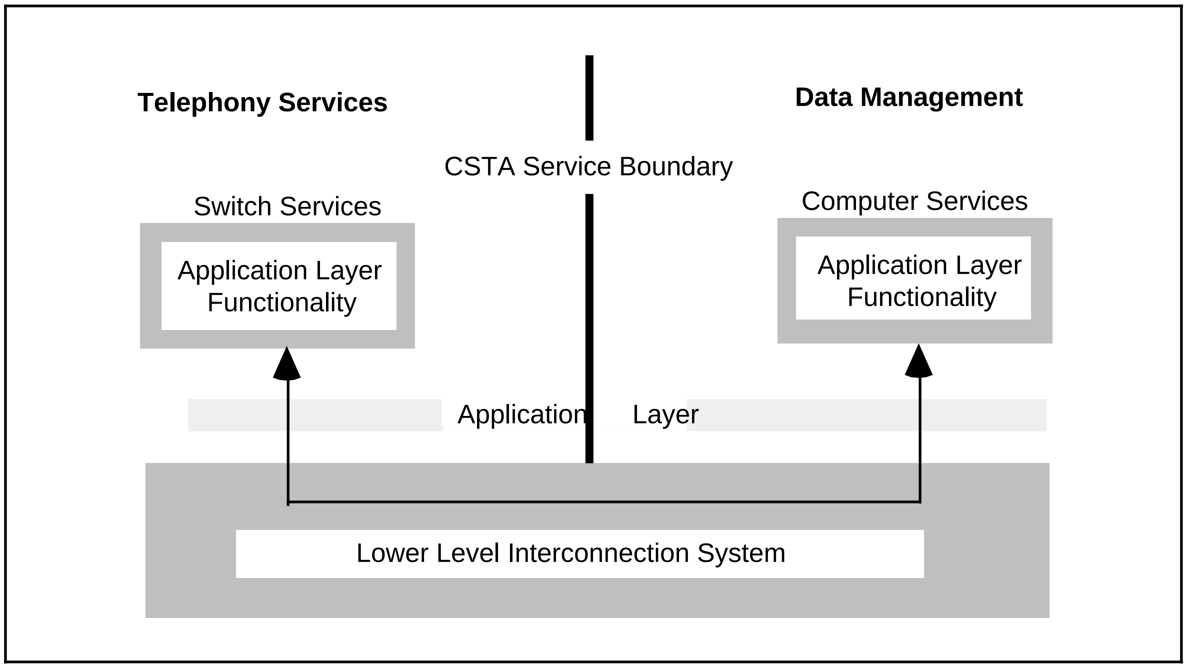
Theoretically, the CSTA services of each vendor's switch should work in a uniform manner. In practice, however, individual switches are typically implemented with different limitations or restrictions. In some cases, a vendor provides more services than are defined in the CSTA standard. As a result, CSTA provides a "private data" mechanism that allows for the support of vendor-specific services. Conversely, there are various optional protocol elements defined in the CSTA standard that may provide more services than are defined by a particular vendor's implementation. As a result, the use of optional protocol elements in the API may be vendor-dependent.

CSTA Communication Layers

A CSTA application is one that is supported by a computing component (providing data management) and a switching component (providing telephony services). The communications interface between the application layers of each component is defined by the CSTA protocol. See Figure 1-1.

The actual implementation of the CSTA protocol is defined by the CSTA standard. The CSTA standard defines an Open System Interconnection (OSI) application layer communication protocol that supports peer-to-peer communication between computer applications and switch applications. The lower level communication protocol ("Lower Layer Interconnection System") is implementation-dependent and is transparent to the services provided by the application layer.

Figure 1-1: CSTA Communication Layers



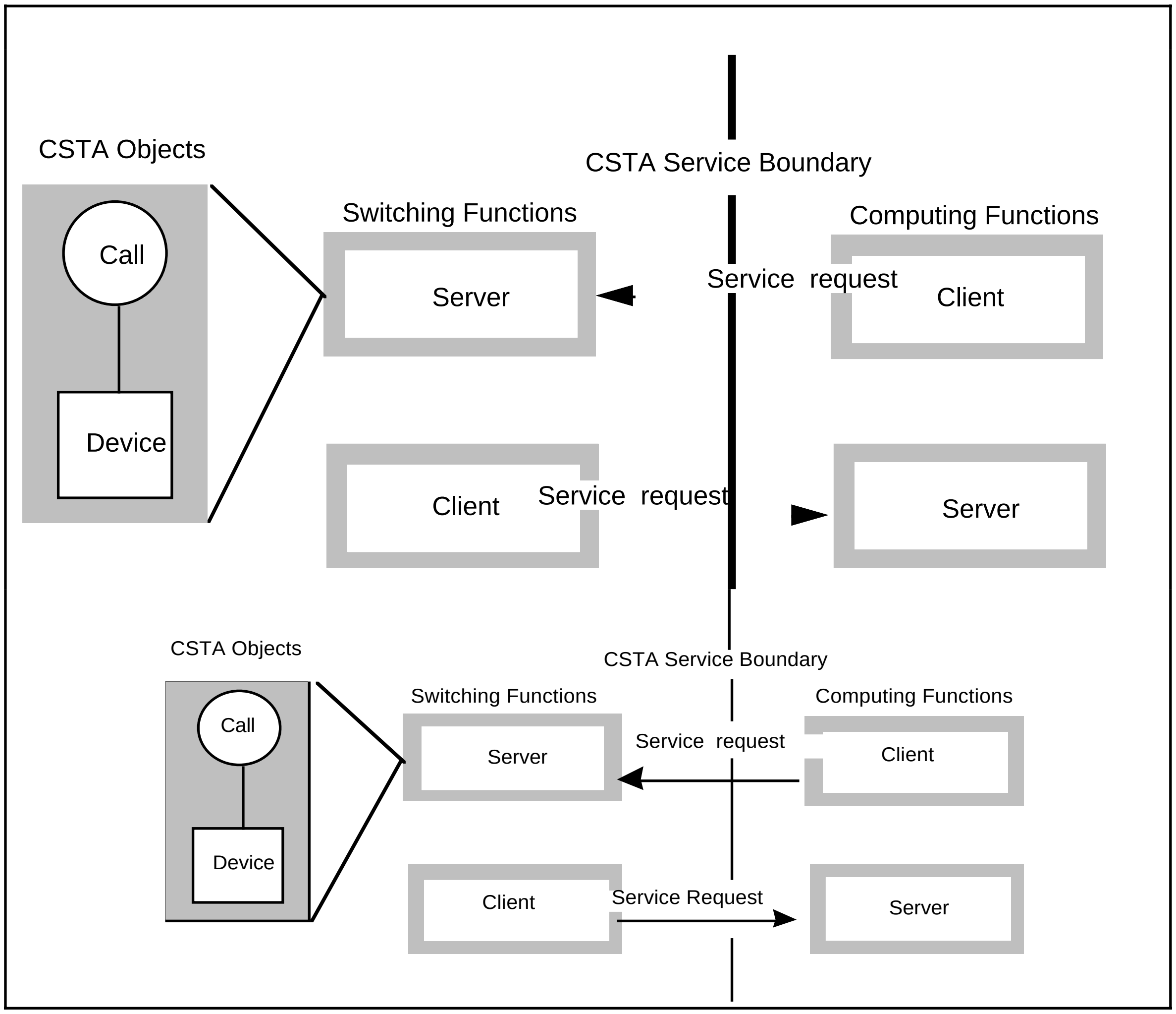
CSTA Client/Server Operational Model

The CSTA standard describes a client/server operational model as a relationship between two peer application layer processes where one process is able to perform a service for another. A "server" process performs a service for a "client" process. Specifically, a server observes and manipulates various telecommunication objects on behalf of a client.

The communication interface between switching applications and computing applications is modeled on a client/server relationship. This relationship, as defined by the CSTA architecture, allows for bi-directional services; that is, switching and computer applications can both assume the role of either client or server. The application that initiates a request for service assumes the role of client. The application that responds to the request and provides the service is the server.

When an application requests a service, its local communications component (client) invokes the service by communicating with its peer component (server). Each instance of a request that is responded to with a service creates a new client/server relationship.

Figure 1-2: CSTA Client/Server Operational Model



Client/Server Session and Operation Invocation

To obtain CSTA services from the Tserver, a communication channel must be established between the client and the server. This client/server connection establishes a session (ACS stream) between the TSAPI application at the client PC and the CSTA application on the server. ACS streams are identified by an *acsHandle* returned from the *acsOpenStream* function.

When a CSTA service is requested (invoked) by a client application, a request/reply relationship is established between the client and the CSTA server application. The server replies to the client’s request with either confirmation (result) or failure (error/rejection). A client’s request for service is also known as an "operation invocation".

Normally, after the client receives a positive or negative acknowledgment ("service response") from the server, the operation invocation is terminated. Clients can identify the acknowledgment of a particular request via the "invoke ID" that is returned with the acknowledgment.

For some services, the invoke ID terminates after a positive acknowledgment, while the service does not. In this case, a "cross reference ID" is included with the acknowledgment. The cross reference ID is a unique value that can be associated by event reports to the service request that initiated it. The cross reference terminates when the service is stopped. An example of such a service is the Monitor Service Group that provides event reports for a call or device.

Call Control and Call Events

CSTA Call Control Services allow a telephony application client to control a call or control connections on a switch. These services are used to set up and control any endpoint on a single call.

Although they can manipulate the state of the switch objects, Call Control Services do not provide CSTA Event Reports as objects change their state. To monitor switch object state changes (that is, receive CSTA Event Report Services from a switch), a client must request a CSTA Monitor Service for an object before the Call Control Services are requested for that object.

DMS-100 CSTA System Overview

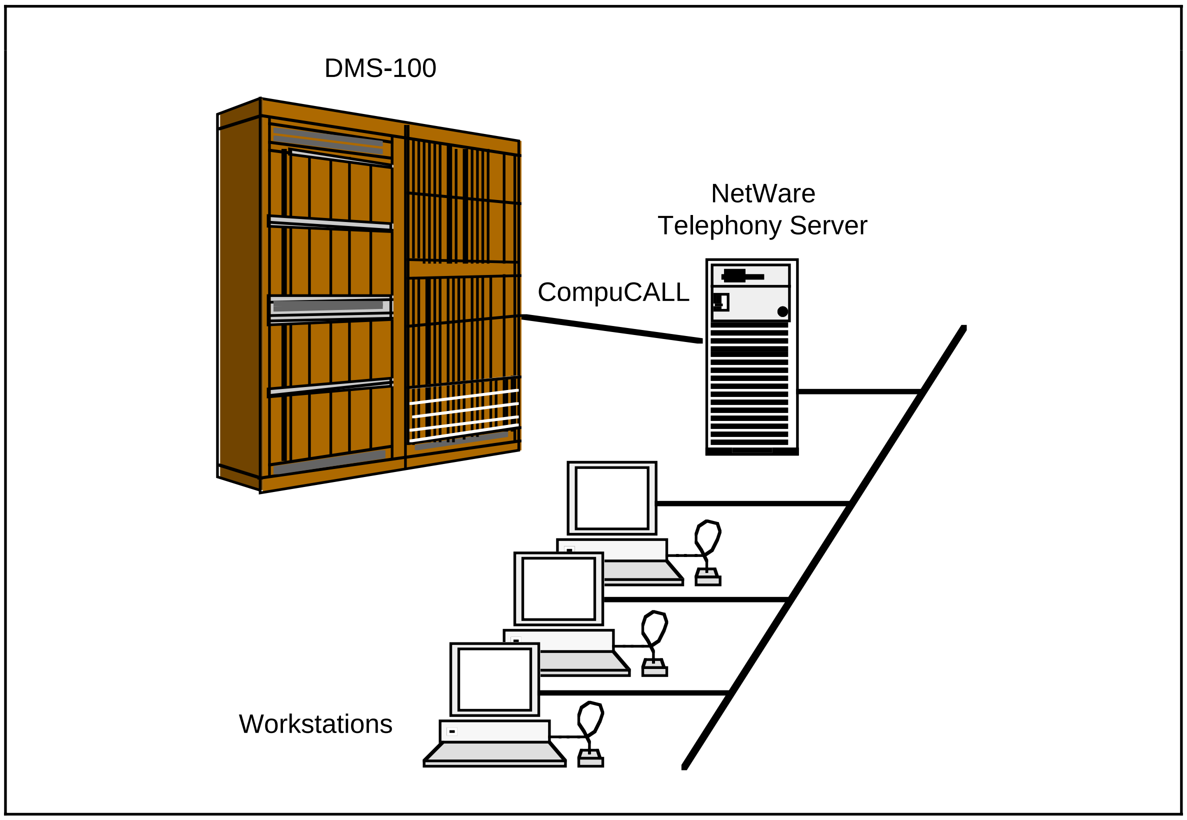
************* needs change **

In a DMS-100 environment, the DMS-100 Central Office Switch (or SL-100 PBX) is connected to a telephony server through a Multi-Protocol Controller (MPC) card. The MPC implements the X.25 protocol which is normally connected to the Tserver through a leased data line. The connection between the Tserver and the DMS-100 IPE Module or Application Module is referred to as CompuCALL Link.

Desktop PCs are networked to the Tserver via a local area network (LAN) that supports a NetWare-based communication platform. Voice terminals are directly connected to the

DMS-100 and may be either analog or digital. All telecommunications, whether internal or external, are supported by the DMS-100 Communications System.

Figure 1-3: DMS-100 NTS System Overview



The NetWare Telephony Server (NTSTserver)

The following are server components:

- X.25 CompuCALL Link Hardware Driver
- X.25 Transport NLM Switch Driver
- DMS-100 NTS Driver NLM
- Telephony Services NLM
- Telephony Services Library

Of these components, the X.25 Hardware Driver, CompuCALL Link Hardware Driver X.25 Transport NLM and DMS-100 NTS Driver NLM are switch dependent. All other components are switch independent.

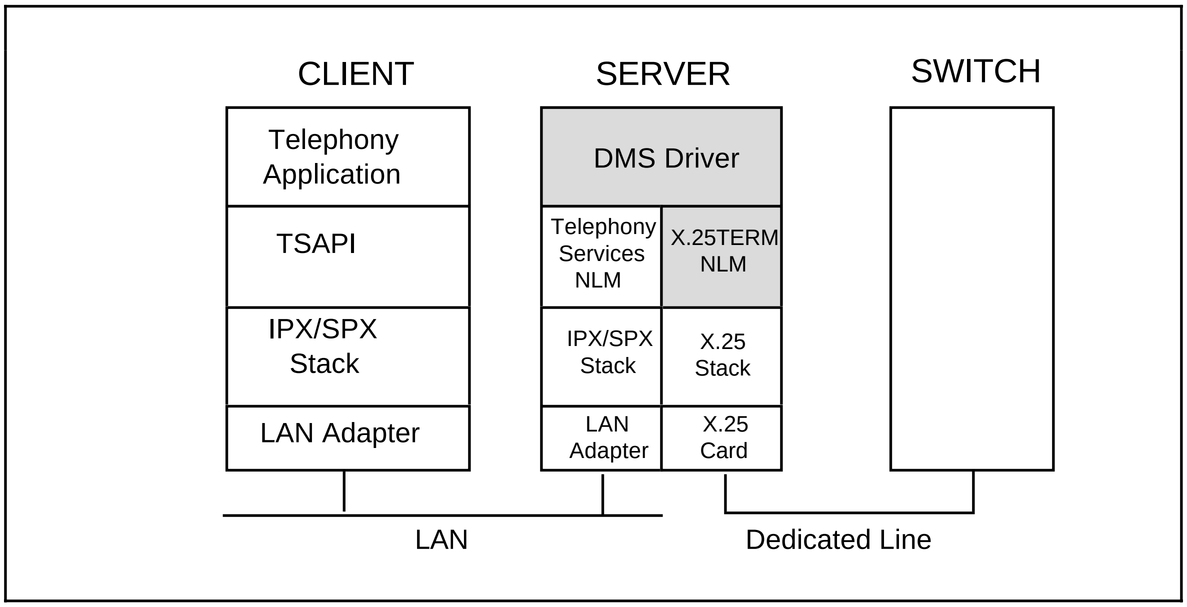
DMS-100 CSTA Software Overview

DMS-100 CSTA services are provided by the DMS-100 Switch software and the DMS-100 NTS Driver NetWare Loadable Module (NLM) running on a Tserver. The CompuCALL Link application processes the CSTA requests and responses between client applications and the call processing software on the DMS-100 Switch. The CompuCALL Link between the DMS-100 Switch and the Tserver supports an interface for the DMS-100.

The client telephony application is supported by the client Telephony Services API Library in a Windows environment on the desktop PC (or other operation systems/computers as defined by the TSAPI product). Through the TSAPI, the client application can use TSAPI to establish a CSTA session with the Telephony Services NLM (Tserver NLM) running on the Tserver.

The Telephony Services NLM controls the communication between users on the network and the DMS-100 NTS Driver NLM. Communication between the Telephony Services NLM and the DMS-100 NTS Driver NLM is supported by the Telephony Services Driver Interface (TSDI). The TSDI (or CSTA Services Driver Interface - CSDI) provides an open interface for the Telephony Services NLM to communicate with any switch vendor’s DMS-100 NTS Driver NLM for supporting the CSTA services for a specific switch.

Figure 1-4: Components and their configuration



DMS-100 CSTA Software Components Overview

CLIENT TSAPI LIBRARY

The Client TSAPI Library makes the CSTA API available to applications running on client workstations. Independent software can use the library to communicate with any device that understands the message format used by the Tserver. In the future, this may be true even for a non-telephony API such as a voice mail API that communicates with a voice mail driver and voice mail server (instead of a switching server). The TSAPI Library is able to support these applications in the same way that it supports the CSTA API that communicates to the switch through the Tserver and DMS-100 NTS Driver NLM.

The Client TSAPI Library is comprised of various communication layers. These layers are responsible for

- providing the API Control Services and the CSTA API
- encoding/decoding data to and from a machine-independent byte stream
- sending/receiving data from the network

TELEPHONY SERVICES NLM

The Tserver is a *switch-independent* NetWare NLM that runs on a NetWare server connected to a local area network (LAN). It processes the open and close requests of communication channels for all clients connected to the LAN, thereby managing the communications session (ACS) between a client application and a DMS-100 NTS Driver NLM.

The Tserver also processes service requests from clients on the LAN by authenticating the requests and passing them on to the appropriate DMS-100 NTS Driver NLM. It then returns the DMS-100 NTS Driver NLM responses to the appropriate client.

The DMS-100 NTS Driver NLM registers its services with the Tserver. The Tserver informs clients on the LAN of both its own services and those of the DMS-100 NTS Driver NLM. When a driver unregisters, the Tserver no longer informs clients on the LAN of the DMS-100 NTS Driver NLM's services.

The Tserver uses a security to authenticate the user privileges that exist on various devices for different telephony services such as call control, monitoring, query, and routing. Part of the security check involves verifying that the user requesting a connection to the Tserver has an administered login.

Administration and maintenance capabilities are also available from the Tserver. The Tserver provides a private API for Tserver administration and maintenance, and a public API for DMS-100 NTS Driver NLM administration and maintenance.

Only one Tserver can be loaded onto a NetWare server.

TELEPHONY SERVICES DRIVER INTERFACE (TSDI)

The interface between the Tserver and the DMS-100 NTS Driver NLM is referred to as the TSDI. The TSDI exchanges messages between the Tserver NLM and a Switch (or other) Driver NLM. This interface defines a set of function calls accessed by both the Tserver and DMS-100 NTS Driver NLMs. All of these functions are defined in the Tserver NLM, and the functions that are used by the DMS-100 NTS Driver NLM are exported by the Tserver NLM.

A DMS-100 NTS Driver NLM must register a connection with the Tserver before messages can be exchanged across the TSDI. The DMS-100 NTS Driver NLM is responsible for configuring the parameters for this Telephony Services Driver Interface at registration time. After a connection to the Tserver has been registered, the Tserver and the DMS-100 NTS Driver NLMs can exchange messages using a send/receive programming model.

All memory allocated for a specific TSDI interface is allocated by the Tserver NLM. Data structures maintained by the TSDI are allocated at DMS-100 NTS Driver NLM registration time, and the message buffers used to exchange messages across the TSDI are allocated and released in the Telephony Services NLM context when needed.

SWITCH

A Switch Driver NLM is a *switch-dependent* NetWare NLM that provides vendor-specific telephony services to client applications. This NLM is provided by the vendor that provides the Switch and the CSTA services for that switch. For the DMS-100 Switch, this NLM is referred to as the DMS-100 NTS Driver NLM.

A DMS-100 NTS Driver NLM can represent one or more Switch switches in the switching domain and communicates with the Tserver to provide switching services to applications in the computing domain. The DMS-100 NTS Driver NLM communicates with the Tserver via the *switch-independent* Telephony Services Driver. However, communication between the DMS-100 NTS Driver NLM and a switch is defined by an interface (possibly proprietary) that depends on the switch vendor’s implementation.

The main functions of the DMS-100 NTS Driver NLM are the following:

- Handle CSTA telephony requests from the Telephony Services NLM; translate those CSTA requests into the appropriate CompuCALL protocol requests; and send them to the DMS-100.
- Communicate with the Switch through the X.25 Transport NLM, a vendor-specific card and a X.25 hardware interface on the Tserver.
- Handle requests, responses, and events from the Switch, translate them into the corresponding CSTA messages; and send those CSTA messages to the Telephony Services NLM.
- Provide a private application programming interface for Switch-specific administration and maintenance.

Note that the device connected to the DMS-100 NTS Driver NLM that provides the services might not actually be a Switch. As stated previously, it could be a device providing voice mail service or some other telephony-related services. However, the term "DMS-100 NTS

Driver NLM" will be used generically since most telephony services are provided by Switch devices.

DMS-100 NTS DRIVER

The NetWare Telephony Services (NTS) product provides a DMS-100 NTS Driver NLM for the DMS-100 Switch. This driver communicates with the DMS-100 Switch over the CompuCALL Link using the X.25 CTI protocol stack software. A major function of the DMS-100 NTS Driver NLM is to translate protocols of the CSTA requests and responses between the DMS-100 Switch call processing and client applications

When the DMS-100 NTS Driver NLM is loaded, it initializes the X.25 CTI protocol stack with the DMS-100 Switch. It then registers its services with the Tserver After it establishes the communications to the Tserver, it begins to service requests from the switch and client applications.

For most CSTA services, the DMS-100 Switch and DMS-100 NTS Driver NLM act jointly as a server. However, for the Routing Service, the client/server relationship reverses. In this case, the client application becomes a routing server.

The DMS-100 NTS Driver NLM currently supports only one DMS-100 switch connection.

The DMS-100 NTS Driver NLM may be configured through the use of a text editor.

Supported Service Groups

DMS-100 NTS Driver NLM Release 1.0 supports the following service groups on a DMS-100 Switch:

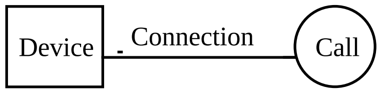
Call Control Service Group:	The services in this group enable a telephony client application to control a call or connection on the DMS-100 Switch. Typical uses of these services are placing calls from a device and controlling any connection on a single call as the call moves through the DMS-100. The following services are supported: **Conference Call Service** **Consultation Call Service** **Make Call Service** **Reconnect Call Service** **Transfer Call Service**
Set Feature Service Group:	These services allow a client application to set switch-controlled features or values on a DMS-100 Switch device. The following services are supported: **Set Agent State Service**
Monitor Service Group:	The services in this group allow a client application to request and cancel the reporting of events that cause a change in the states of a DMS-100 Switch object. The following services are supported: **Monitor Device Service** **Monitor Stop Service**
Event Report Service Group:	This service group provides a client with application reports of events that cause a change in the status of a call, a connection, or a device. The following services are supported: **Call Event Reports :** **Connection Cleared** **Conferenced** **Delivered** **Established** **Network Reached** **Queued** **Transferred** **Agent State Event Reports:** **Logged On** **Logged Off**

	Ready **Not Ready**
Computer Function Routing()? Services Group:	These services allow .... The following services are supported: **Route Request Service** **Route Select Service**

Not Ready

DMS-100 CSTA Objects

There are three types of CSTA objects: Device, Call, and Connection.



CSTA OBJECT: DEVICE

The term *Device* refers to both physical devices (stations, trunks, etc.) and logical devices (CDN or ACD) that are controlled via the switching application. Each device is characterized by a set of attributes. These attributes define the manner in which a CSTA application may observe and manipulate a device. The set of device attributes is Device Type, Device Class, and Device Identifier.

Device Type

The following table defines the most commonly used DMS-100 CSTA devices and their types:

Table 2—1: DMS-100 CSTA types

CSTA Type	Definition	DMS-100 Object
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).	A Meridian Digital Switch or RES Directory Number on the DMS-100.
ACD	A mechanism that distributes calls within a switch function.	DMS Meridian ACD.
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._	ACD group in the DMS-100. ACD agent position

Other device types that are CSTA-defined but are not commonly used by the DMS-100 CSTA include *Station, button, button group, line group, operator, operator group, station group, trunk; trunk group.*

Device Class

One class of devices can be observed and manipulated within DMS-100 CSTA. This is *voice.*

Device Identifier

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

Static Device Identifier

This identifier is stable over time and remains both constant and unique between calls. The static device identifier is known *a priori* by both the computing and switching functions.

DMS-100 phone and directory numbers internal extensions are *static* device identifiers. This includes numbers extensions that uniquely identify any DMS-100 devices such as lines, ACD, and ACD agent’s position ID. Valid phone numbers for endpoints external to the DMS-100 Switch are also static device identifiers.

Note that a connection on a call with a static device identifier does not mean that the device is on Switch or that the connection is not made through a trunk.

Dynamic Device Identifier

A trunk is a *dynamic* device identifier in the switch. This identifier is not like a static device identifier such as a station that identifies a static endpoint and can be stored in a database for use by an application. This is because a trunk addresses the *point* at which the call crosses a CSTA switching sub-domain boundary. In order to manipulate and view calls that cross a CSTA switching sub-domain, it may be desirable to know the trunk identifier. An off-Switch endpoint without a known static identifier has a trunk identifier. The trunk identifier of an external endpoint is preserved across the conference and transfer operation. A trunk identifier is meaningful to an application only for one call scenario and should not be used like a phone number or a station extension in a database operation. A call identifier and a trunk identifier can uniquely represent a connection identifier in a DMS-100 switch. A trunk identifier does not necessarily identify its trunk group. DMS-100 NTS does not support Trunk Identifiers.

Device Identifier Syntax

Device Identifier Syntax

```
typedef char        DeviceID_t[64];

typedef enum DeviceIDType_t {
            DEVICE_IDENTIFIER = 0,                                         // supported
            IMPLICIT_PUBLIC = 20,                                          // not supported
            EXPLICIT_PUBLIC_UNKNOWN = 30,                                  // not supported
            EXPLICIT_PUBLIC_INTERNATIONAL = 31,                   // not supported
            EXPLICIT_PUBLIC_NATIONAL = 32,                                 // not supported
            EXPLICIT_PUBLIC_NETWORK_SPECIFIC = 33,                         // not supported
            EXPLICIT_PUBLIC_SUBSCRIBER = 34,                               // not supported
            EXPLICIT_PUBLIC_ABBREVIATED = 35,                              // not supported
            IMPLICIT_PRIVATE = 40,                                // not supported
            EXPLICIT_PRIVATE_UNKNOWN = 50,                                 // not supported
            EXPLICIT_PRIVATE_LEVEL3_REGIONAL_NUMBER = 51,                  // not supported
            EXPLICIT_PRIVATE_LEVEL2_REGIONAL_NUMBER = 52,                  // not supported
            EXPLICIT_PRIVATE_LEVEL1_REGIONAL_NUMBER = 53,                  // not supported
            EXPLICIT_PRIVATE_PTN_SPECIFIC_NUMBER = 54,                     // not supported
            EXPLICIT_PRIVATE_LOCAL_NUMBER = 55,                            // not supported
            EXPLICIT_PRIVATE_ABBREVIATED = 56,                             // not supported
            OTHER_PLAN = 60,                                               // not supported
            TRUNK_IDENTIFIER=70,                                           // not supported
            TRUNK_GROUP_IDENTIFIER=71                                      // not supported
} DeviceIDType_t.i.DeviceIDType_t;;

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

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

typedef ExtendedDeviceID_t          CallingDeviceID_t.;

typedef ExtendedDeviceID_t          CalledDeviceID_t.

typedef ExtendedDeviceID_t          SubjectDeviceID_t;

typedef ExtendedDeviceID_t          RedirectionDeviceID_t;
```

DMS-100 Device Type and Status

Table 2—2: DMS-100 device types and status

DMS-100 Object	DeviceIDType_t	ConnectionID_Device_t	DeviceIDStatus_t
Line	DEVICE_IDENTIFIER	STATIC_ID	ID_PROVIDED
ACD	DEVICE_IDENTIFIER	STATIC_ID	ID_PROVIDED
ACD Group	DEVICE_IDENTIFIER	DYNAMIC_ID	ID_PROVIDED

CSTA OBJECT: CALL   CALL

Call behavior, including establishment and release, can be observed and manipulated by the CSTA API. There are two types of call attributes: *Call Identifier* and *Call State*.

Call Identifier (callIDD)

When a call is initiated, a unique Call Identifier (callID) is allocated by the DMS-100 switching function. Before it finally terminates, a call may assume many different states involving a variety of devices. Even though the call identifier may change (as with transfer and conference, for example), its status as a CSTA object remains the same.

A callID first becomes visible to an application when it is reported in an event report or confirmation event. The allocation of a callID is always reported. Each callID must be specified in a *connection identifierr* parameter that crosses the boundary of the switching and computing domains.

The DMS-100 driver uses the CompuCALL callID in it's reporting monitoring of all events with the exception of consultation legs calls where a negative callID is used to for the duration of the consult process.

Note that the TSAPI interface passes callID parameters within *connectionID* parameters.

Call Identifier Syntax

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

Call State

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

CSTA OBJECT: CONNECTION

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

Connection identifier

A *connectionID* is a combination of Call Identifier (callID) and Device Identifier (deviceID). The connectionID is unique within a DMS-100 Switch. This identifier is a DMS-100 NTS Driver NLM-assigned identifier. A CSTA application should cannot use a connectionID until it has received it from the DMS-100 NTS Driver NLM. This rule restricts a CSTA application from fabricating a connectionID.

A connectionID always contains a callID value. A DMS-100 NTS Driver NLM connectionID may contain a static device identifier. If the callID is the only value that is set, the deviceID is set to 0 (with DYNAMIC_ID). The callID of a connectionID assigned to an endpoint on a call may change when the call is transferred or conferenced, however, the deviceID of the connectionID assigned to an endpoint will not change when the call is transferred or conferenced.

For a call, there are as many Connection Identifiers as there are associated devices. For a device, there are as many Connection Identifiers as there are associated calls.

Connection Identifier Syntax

```
typedef char        DeviceID_t[34];

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

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

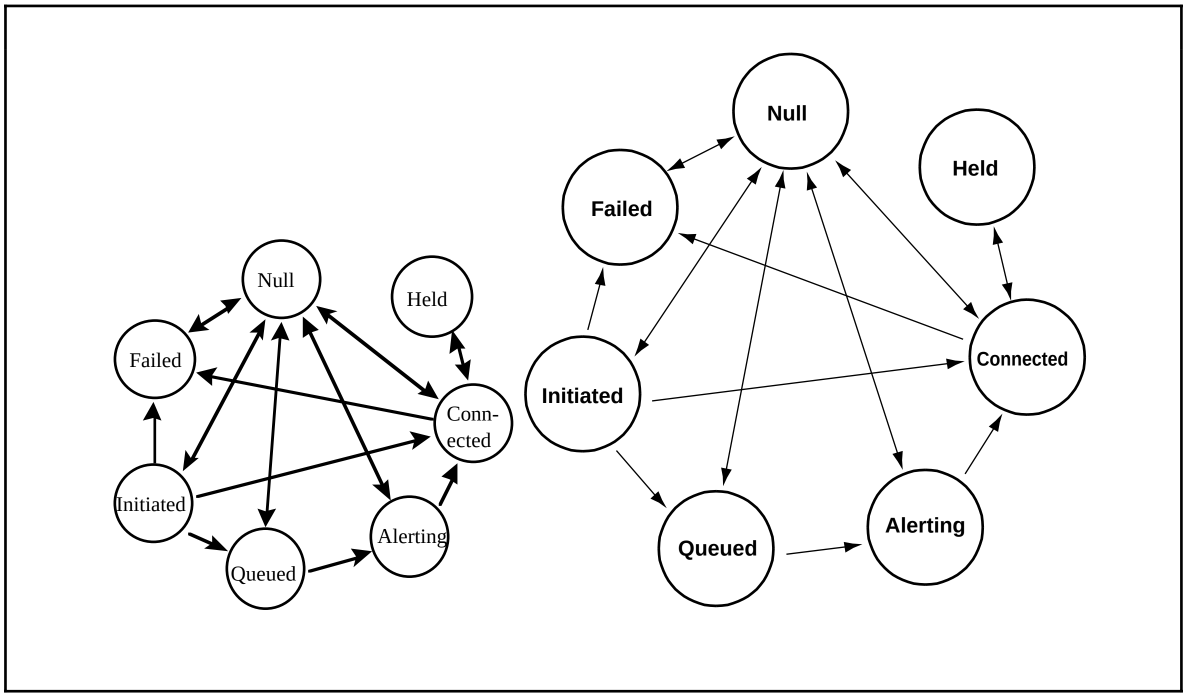
Connection State

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

The following figure illustrates a CSTA connection state model or switching model that shows typical connection state changes. It provides an abstract view of various call state transitions that can occur when a call is either initiated from, or delivered to, a device. Note that it does not include all the possible states that may result from interactions with DMS-100 Switch features. Also, some of the state changes presented may not be meaningful to the switch.

This switching model is a more detailed description than is required by a CSTA application. It does not represent a programming model for the *call statement report connection state relationship.* A detailed description of such a programming model is presented in the Chapter 7, "Call Scenarios and Events" section.

Figure 2-1: CSTA Communication Layers



CSTA Connection State Model

In the preceding illustration, connection states are represented by circles. Arrows are used to signify transitions between states. A transition from one connection state to another results in the generation of an event report. The various connection states are defined as follows:

Null	No relationship exists between the call and device; a device does not participate in a call.
Initiated	A device is requesting service. Usually, this results in the creation of a call. Often, this is when the station receives a dial tone and begins to dial.
Alerting	A device is alerting (ringing). A call is attempting to become connected with a device. The term "active" is also used to indicate an alerting (or connected) state.

Connected	A device is actively participating in a call, either logically or physically (that is, not Held). The term "active" is also used to indicate a connected (or alerting) state.
Held	A device inactively participates in a call. That is, the device participates logically but not physically.
Queued	Normal state progression has been stalled. Generally, either a device is trying to establish a connection with a call, or a call is trying to establish a connection with a device.
Failed	Normal state progression has been aborted. Generally, either a device is trying to establish a connection with a call, or a call is trying to establish a connection with a device. A Failed state can result from a failure to connect to the calling device (origin) or to the called device (destination). It can also be caused by a failure to create the call, or other factors as well.
Unknown	A device participates in a call, but its state is not known.

Connection State Syntax

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

DMS-100 NetWare Telephony Services System Capacity

The following table provides the system capacity information for the DMS-100 CSTA. These are maximum system capacities. A NetWare Telephony Server's capacity is limited by these numbers, as well as by the hardware configuration on the Telephony Server and by the switch configuration. The number of users that can access the Telephony Server is independent of these numbers.

Table 2—3: DMS-100 NTS system capacity

Parameter	DMS-100 NTS System Capacity	Comments
CompuCALL Links	1	
CSTA Service requests per CompuCALL Link	limited by the switch capacity	
Objects monitored by ***cstaMonitorDevice*** requests	Overall 10/session**	This is the maximum number of monitored DNs and ACD position IDs. See Note 1.
Objects monitored by ***cstaMonitorCallsViaDevice*** requests	Overall 10/session	This is the maximum number of monitored DNs and ACD position IDs. See Note 1.
Simultaneous ***cstaMonitorDevice*** monitor requests on one station device OR ***cstaMonitorCallsViaDevice*** monitor requests on one ACD device	Only 1 association between the DMS-100 NTS Driver NLM and the switch	Applications may issue multiple monitor requests on a device; however, only one request is allowed per session.*
CSTA Client Sessions	2000	
Simultaneous CSTA service requests	2000	See Note 2.
Number of devices that can be on a call	6	See Note 3.
Number of ***cstaMonitorDevice*** monitored objects that can be involved in a call OR **cstaMonitorCallsviaDevice**	10	
Number of monitored objects that can be involved in a call	10	
Number of CSTA monitor requests from applications that can be involved in a call	N/A	Each CSTA Event Report of a monitored object will be sent to every monitor request.

* Objects, calls, sessions, and monitors are limited by server memory.

* * Objects monitored per session are limited to 10.

Note 1 This is not the number of total monitor requests. An object monitored by multiple monitor requests is counted only once. All Call Control Service requests on a station device other than Clear Connection and Clear Call are included in this number. When a station device is monitored, the Call Control Service requests on the device are not counted as additional requests.

Note 2 This is an estimated number. This number includes all outstanding Clear Connection Service requests, and Set Feature Service requests.

Note 3 A call can have a maximum of six parties.

Format and Conventions

The following is the general format used to describe each DMS-100 CSTA service.

Direction: direction of the service request across the TSAPI interface
C $\rightarrow$ S client to server
C $\leftarrow$ S server to client

Function: function name and confirmation event as defined in **TSAPI**.

Service Parameters: A list of parameters for this service request. Common ACS parameters such as acsHandle, invokeID, and privateData are not shown.

Private Parameters: A list of parameters can be specified in private data for this service request.

Ack Parameters: A list of parameters in the confirmation event for the positive acknowledgment from the server. Common ACS parameters such as acsHandle, eventClass, eventType, and privateData are **not** shown.

Ack Private Parameters: list of parameters in the private data of the confirmation event for the positive acknowledgment from the server.

Nak Parameter: *universalFailure*

Functional Description: Detailed description of the telephony function that this CSTA Service provides in a DMS-100 CSTA environment.

Service Parameters:

Service Parameters:

parameter	Detailed information for each parameter in the service request. A ***noData*** indicator means that it requires no additional parameters other than the common ACS parameters. The mandatory/optional attribute of a parameter is defined as follows:
mandatory	This parameter is mandatory as defined in Standard ECMA-179. It must be present in the service request. If not , the service request will be denied with OBJECT_NOT_KNOWN.
mandatoryPartially	This parameter is mandatory as defined in Standard ECMA-179. However, DMS-100 CSTA can only support part of the parameter due to the DMS-100 Switch feature limitations. A default value is set for the portion not supported.
mandatoryNotSupported	This parameter is mandatory as defined in Standard ECMA-179. However, DMS-100 CSTA does not support this parameter due to the DMS-100 Switch feature limitations. "Not supported" means that, whether it is present or not, the value specified is ignored and a default value is assigned. If this is a parameter (for example, event report parameter) returned from the switch, a default value (for example, ID_NOT_REQUIRED) is set for this parameter.
optional	This parameter is optional as defined in Standard ECMA-179. It may or may not be present in the service request. If not, a default value is assigned.
optionalSupported	This parameter is optional as defined in Standard ECMA-179, but it is always supported.
optionalPartially	This parameter is optional as defined in Standard ECMA-179. However, DMS-100 CSTA can only support part of the parameter due to the DMS-100 Switch feature limitations. The part that is not supported is ignored if it is present.
optionalNotSupport	This parameter is optional as defined in Standard ECMA-179, but it is not supported by DMS-100 CSTA. "Not supported" means that, whether it is present or not, the value specified is ignored and a default value is assigned.

Private Service Parameters:

parameter	Detailed information for each private parameter in the service request. The mandatory/optional attribute of a parameter is defined as follows:
mandatory-nt	This parameter is mandatory for the specific service. It must be present in the private data of the request. If not, the service request is denied with OBJECT_NOT_KNOWN.
optional-nt	This parameter is optional for the specific service. It may or may not be present in the private data. If not, a default value is assigned.
optional-nt NotSupport	This parameter is optional for the specific service. This parameter is reserved for future use. It is ignored for the current implementation.

Ack Parameters:

parameter	Detailed information for each parameter in the service confirmation event. A **noData** indicator means that the DMS-100 NTS Driver NLM sends no additional parameters other than the confirmation event itself along with the common ACS parameters.

Ack Private Parameters:

parameter	Detailed information for each parameter in the private data of the service confirmation event.

Nak Parameter:

The following is a list of the most commonly used CSTA errors returned from the DMS-100 CSTA in the CSTAUniversalFailureConfEvent for a negative acknowledgment of this service.

NOTE: This list does not include those error codes that are returned by the Tserver NLM. Those error codes (for example, security violation detected by Tserver) returned by the Tserver NLM are documented in *Telephony Services Application Programming Interface (TSAPI),* 555-201-202.

NOTE: This list does *not* include all possible errors. An application program should be able to handle any CSTA error defined in the CSTAUniversalFailure_t. Failure to do so may cause the application program to fail.

The following common errors apply to every CSTA Service supported by DMS-100 CSTA. They will not be repeated for each service description.

universalFailure

GENERIC_OPERATION (1)

The CTI protocol has been violated or the service invoked is not consistent with a CTI application association. This error should be reported to Northern Telecom.

REQUEST_INCOMPATIBLE_WITH_OBJECT (2)

The service request does not correspond to a CTI application association. This error should be reported to Northern Telecom.

VALUE_OUT_OF_RANGE (3)

The DMS-100 Switch detects that a required parameter is missing in the request or a out-of-range value is used.

OBJECT_NOT_KNOWN (4)

The DMS-100 NTS Driver NLM detects that a required parameter is missing in the request. For example, the deviceID of a connectionID is not specified in a service request.

INVALID_FEATURE (15)

The DMS-100 NTS Driver NLM detects a CSTA Service request that is not supported by the DMS-100 Switch.

GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31)

The request cannot be executed due to a lack of available switch resources.

RESOURCE_OUT_OF_SERVICE (34)

An application can receive this error code when a single CSTA Service request is ending abnormally due to protocol error.

NETWORK_BUSY (35)

The Switch is not accepting the request at this time because of processor overload. The application may wish to retry the request but should not do so immediately.

OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44)

The given request cannot be processed due to the system resource limit on the device.

GENERIC_UNSPECIFIED_REJECTION (70)

This is a DMS-100 NTS Driver NLM internal error, but it cannot be any more specific. A system administrator may find more detailed information about this error in the DMS-100 NTS Driver NLM error logs. This error should be reported to Nortel.

GENERIC_OPERATION_REJECTION (71)

This is a DMS-100 NTS Driver NLM internal error but not a defined error. A system administrator should check the DMS-100 NTS Driver NLM error logs for more detailed information about this error. This error should be reported to Nortel.

UNRECOGNIZED_OPERATION_REJECTION (73)

The DMS-100 NTS Driver NLM detects that the service request from a client application is not defined in the API. This could be an unrecognized message received by the DMS-100 NTS Driver NLM from the TSDI interface. A CSTA request with a 0 or negative Invoke ID will receive this error.

RESOURCE_LIMITATION_REJECTION (75)

The DMS-100 NTS Driver NLM detects that it lacks internal resources such as the memory or data records to process a service request. A system administrator should check the DMS-100 NTS Driver NLM error logs for more detailed information about this error. This failure may reflect a temporary situation. An application should retry the request.

Detailed Information: Detailed information about switch operations, feature interactions, restrictions, and special rules.

Syntax: C-declarations of the TSAPI function and the confirmation event for this service.

Private Parameter Syntax: C-declarations of the private parameters with the function to set them up in the private data of the request, and C-declarations of the private parameters in the confirmation event for this service.

Example: Programming examples are given for some of the services and events.

Common ACS Parameter Syntax

```
typedef unsigned long       InvokeID_t;

typedef unsigned short      ACSHandle_t

typedef unsigned short      EventClass_t

typedef unsigned short      EventType_t;

// defines for ACS event classes

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

// defines for CSTA event classes

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

CSTAUniversalFailureConfEvent

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

Syntax

The following structure shows only the relevant portions of the unions for this message.

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

typedef struct
{
                ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        InvokeID_t      invokeID;
                        union
                        {
                        CSTAUniversalFailureConfEvent_t         universalFailure;
                        } u;
                } cstaConfirmation;
        } event;
} CSTAEvent_t.i.CSTAEvent_t;;

typedef struct CSTAUniversalFailureConfEvent_t {
   CSTAUniversalFailure_t error;
} CSTAUniversalFailureConfEvent_t;
```

The ***universalFailure*** error codes are listed below:

```
typedef enum CSTAUniversalFailure_t {
      GENERIC_UNSPECIFIED = 0,
      GENERIC_OPERATION = 1,
      REQUEST_INCOMPATIBLE_WITH_OBJECT = 2,
      VALUE_OUT_OF_RANGE = 3,
      OBJECT_NOT_KNOWN = 4,
      INVALID_CALLING_DEVICE = 5,
      INVALID_CALLED_DEVICE = 6,
      INVALID_FORWARDING_DESTINATION = 7,
      PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE = 8,
      PRIVILEGE_VIOLATION_ON_CALLED_DEVICE = 9,
      PRIVILEGE_VIOLATION_ON_CALLING_DEVICE = 10,
      INVALID_CSTA_CALL_IDENTIFIER = 11,
      INVALID_CSTA_DEVICE_IDENTIFIER = 12,
      INVALID_CSTA_CONNECTION_IDENTIFIER = 13,
      INVALID_DESTINATION = 14,
      INVALID_FEATURE = 15,
      INVALID_ALLOCATION_STATE = 16,
      INVALID_CROSS_REF_ID = 17,
      INVALID_OBJECT_TYPE = 18,
      SECURITY_VIOLATION = 19,
      GENERIC_STATE_INCOMPATIBILITY = 21,
      INVALID_OBJECT_STATE = 22,
      INVALID_CONNECTION_ID_FOR_ACTIVE_CALL = 23,
      NO_ACTIVE_CALL = 24,
      NO_HELD_CALL = 25,
      NO_CALL_TO_CLEAR = 26,
      NO_CONNECTION_TO_CLEAR = 27,
      NO_CALL_TO_ANSWER = 28,
      NO_CALL_TO_COMPLETE = 29,
      GENERIC_SYSTEM_RESOURCE_AVAILABILITY = 31,
      SERVICE_BUSY = 32,
      RESOURCE_BUSY = 33,
      RESOURCE_OUT_OF_SERVICE = 34,
      NETWORK_BUSY = 35,
      NETWORK_OUT_OF_SERVICE = 36,
      OVERALL_MONITOR_LIMIT_EXCEEDED = 37,
      CONFERENCE_MEMBER_LIMIT_EXCEEDED = 38,
      GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY = 41,
      OBJECT_MONITOR_LIMIT_EXCEEDED = 42,
      EXTERNAL_TRUNK_LIMIT_EXCEEDED = 43,
      OUTSTANDING_REQUEST_LIMIT_EXCEEDED = 44,
      GENERIC_PERFORMANCE_MANAGEMENT = 51,
      PERFORMANCE_LIMIT_EXCEEDED = 52,
      SEQUENCE_NUMBER_VIOLATED = 61,
      TIME_STAMP_VIOLATED = 62,
      PAC_VIOLATED = 63,
      SEAL_VIOLATED = 64,                          // The above errors are detected either by
                                                   // the switch or by the TServerDMS-100 NTS
Driver.
                                                   // NLM. The following rejections are
                                                   // generated by the DMS-100 NTS Driver.
```

```
    GENERIC_UNSPECIFIED_REJECTION=70,              // NLM, not by the switch.
    GENERIC_OPERATION_REJECTION =71,
    DUPLICATE_INVOCATION_REJECTION=72,
    UNRECOGNIZED_OPERATION_REJECTION=73,
    MISTYPED_ARGUMENT_REJECTION=74,
    RESOURCE_LIMITATION_REJECTION=75,
    ACS_HANDLE_TERMINATION_REJECTION=76,
    SERVICE_TERMINATION_REJECTION=77,
    REQUEST_TIMEOUT_REJECTION=78,
    REQUESTS_ON_DEVICE_EXCEEDED_REJECTION=79
} CSTAUniversalFailure_t;
```

ACSUniversalFailureConfEvent

Error values in this category indicate that the DMS-100 NTS Driver NLM detected an ACS-related error. This type includes one of the following specific error values:

DRIVER_DUPLICATE_ACSHANDLE (1000)

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

DRIVER_INVALID_ACS_REQUEST (1001)

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

DRIVER_ACS_HANDLE_REJECTION (1002)

The request is rejected because a CSTA request was issued with no prior acsOpenStream request, or the acsHandle given for an acsOpenStream request is 0 or negative.

DRIVER_INVALID_CLASS_REJECTION (1003)

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

DRIVER_GENERIC_REJECTION (1004)

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 (1005)

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

DRIVER_ACSHANDLE_TERMINATION (1006)

Due to problems with the CompuCALL link , the driver has found it necessary to terminate the session with the given acsHandle. The session is closed, and all outstanding requests are terminated.

DRIVER_LINK_UNAVAILABLE (1007)

The driver was unable to open the new session because no link was available to the switch. The link may have been placed in the BLOCKED state, it may have been taken offline, or some other link failure may have occurred. When the link is in this state, DMS-100 NTS Driver NLM remains loaded and advertised, and sends this error for every new acsOpenStream request until the link becomes available again. An already opened session remains open when the link is in this state. It receives no specific notification about the link status unless it attempts a CSTA request. In this state, a CSTA request receives a CSTA Universal Failure with error SERVICE_TERMINATION_REQUEST.

Terminology Usage

The terms party, connection, connectionID, and endpoint are used interchangeably in this document.

The terms client, client application, and application are used to refer to a TSAPI application running on a user’s desktop PC.

Call Control Service Group

Overview

The services in this group enable a telephony client application to control a call or connection on the DMS-100 switch. A typical use of these services is the placing of calls from a device and controlling any connection on a single call as the call moves through the DMS-100 switch.

This section provides descriptions of all Call Control Services supported in the NetWare Telephony Services product. The service descriptions are accompanied by illustrations to depict the sequence of events following execution of a specific service. The illustrations were created using the following conventions:

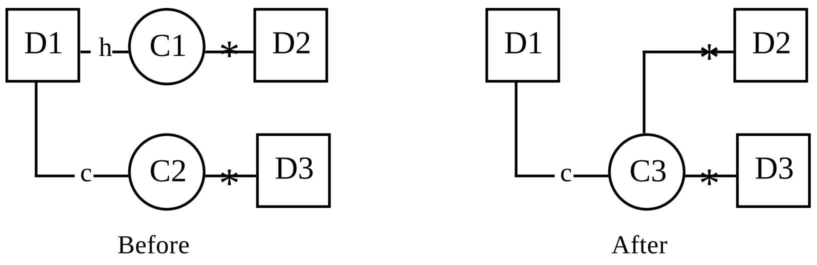
- Boxes represent deviceIDs.
- Circles represent calls and C1, C2, and C3 represent callIDs.
- Lines represent connections between a call and a device, and C1-D1, C1-D2, C2-D3, and so on, represent connectionIDs.
- Absence of a line is equivalent to a connection in the Null connection state.
- Labels in boxes and circles represent call and device instances.
- Labels on lines represent a connection state using the following key:

a = Alerting	c = Connected	f = Failed	h = Held
i = Initiated	q = Queued	a/h = Alerting or Held	* = Unspecified

- Grayed boxes represent devices in a call unaffected by the service or event report.
- White boxes and circles represent devices and calls affected by the service or event report.
- ***Parameters*** for the function call of the service are indicated in bold and italics.

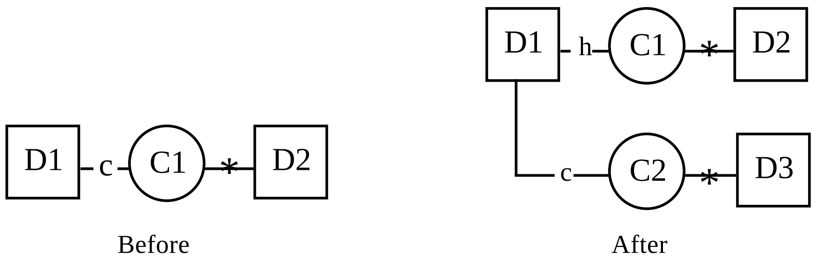
CONFERENCE CALL SERVICE

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



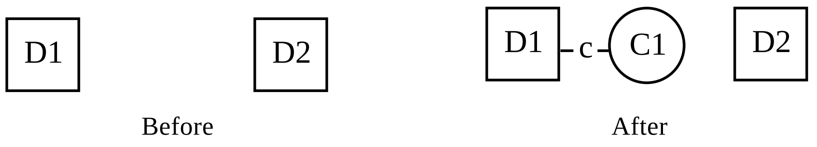
CONSULTATION CALL SERVICE

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



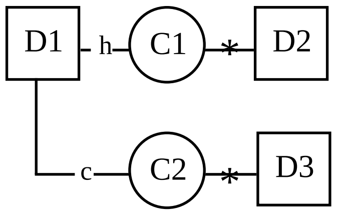
MAKE CALL SERVICE

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

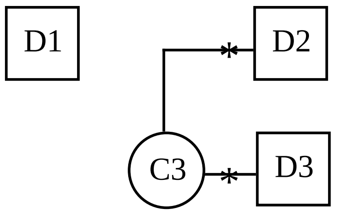


TRANSFER CALL SERVICEERVICE

This service provides the transfer of a ***heldCall*** (C1-D1) with an ***activeCall*** (C2-D1) at the same device (D1). The transfer service merges two calls (C1, C2) with connections (C3-D2, C3-D3) at a single common device (D1) into one call (C3). Also, both of the connections to the common device become Null, and their connectionIDs are released. When the transfer completes, the common device (D1) is released from the calls (C1, C2). A callID, ***newCall*** (C3), that specifies the resulting new call for the transferred call, is provided.



Before



After

Conference Call Service

NOTE: The Conference Call Service must always be preceded by a Consultation Call Service. If the Conference Call request is received prior to a Consultation Call request, the system returns the following error message: "GENERIC_STATE_INCOMPATIBILITY (21)". For more information on the subject, refer to the "Call Scenarios and Events" chapter in this guide.

Function: ***cstaConferenceCall(),*** **C*STAConferenceCallConfEvent***
Direction: C $\rightarrow$ S
Service Parameters: ***heldCall, activeCall***
Ack Parameters: ***newCall, connList***
Nak Parameter: ***universalFailure***
Functional Description:

This service provides the conference of an existing held call (***heldCall***) and another call. The two calls are merged into a single call, and the two connections at the conference controlling device are resolved into a single connection in the connected state. The pre-existing CSTA connectionID associated with the device creating the conference are released, and a new callID for the resulting conferenced call is provided. The DMS-100 NTS driver uses the CompuCALL DV-CONFERENCE-PARTY operation to perform this request. Please see reference ? for details of switch operation and feature interactions.

Service Parameters:

heldCall	[mandatory] Must be a valid connection identifier for the call which is on hold at the controlling device and is to be conferenced with the ***activeCall***. The deviceID in ***heldCall*** must contain the *station extension* of the controlling device.
activeCall	[mandatory] Must be a valid connection identifier for the call which is active or proceeding at the controlling device and is to be conferenced with the ***heldCall***. The deviceID in ***activeCall*** must contain the *station extension* of the controlling device.

Ack Parameters:

newCall	[mandatory - partially supported] A connection identifier specifies the resulting new call identifier for the calls which were conferenced at the conference controlling device. This connection identifier replaces the two previous call identifiers at that device.
connList	[optional - not supported] Specifies the devices on the resulting *newCall*. This includes a count of the number of devices in the new cal,l and a list up of to six connectionIDs and up to six deviceIDs which define each connection in the call. If a party is outside the CompuCALL environment, then its static device identifier is specified.

Nak Parameter:

universalFailure

- INVALID_CSTA_DEVICE_IDENTIFIER (12)

An invalid device identifier or extension is specified in ***heldCall*** or ***activeCall***.

- INVALID_CSTA_CONNECTION_IDENTIFIER (13)

The controlling deviceID in ***heldCall*** or ***activeCall*** has not been specified correctly.

- INVALID_OBJECT_STATE (22)

The connections specified in the request are not in the valid states for the operation to take place. For example, it does not have one call active and one call in the held state as required.

- INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23)

The callID or deviceID in ***activeCall*** or ***heldCall*** has not been specified correctly.

- CONFERENCE_MEMBER_LIMIT_EXCEEDED (38)

The request attempted to add a seventh party to an existing six-party conference call.

- NO_CALL_TO_COMPLETE

Cannot complete conference call.

- REQUEST_INCOMPATIBLE_WITH_OBJECT

Object is in an invalid state.

Detailed Information:

The phone sets associated with the service should be monitored.

Device identifier is limited to 33 charachters.

Syntax

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

// cstaConferenceCall() - Service Request

RetCode_t        cstaConferenceCall (
        ACSHandle_t             acsHandle,
        InvokeID_t              invokeID,
        ConnectionID_t          *heldCall,        // devIDType= STATIC_ID
        ConnectionID_t          *activeCall,      // devIDType= STATIC_ID
        PrivateData_t           *privateData);


// CSTAConferenceCallConfEvent - Service Response

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;       // CSTACONFIRMATION
        EventType_t     eventType;        // CSTA_CONFERENCE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
                ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        InvokeID_t          invokeID;
                        union
                        {
                                CSTAConferenceCallConfEvent_t   conferenceCall;
                        } u;
                } cstaConfirmation;
        } event;
} CSTAEvent_t;

typedef struct Connection_t {
    ConnectionID_t          party;
    DeviceID_t              staticDevice;        // NULL for not present
} Connection_t;

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

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

Consultation Call Service

Function: ***cstaConsultationCall(), CSTAConsultationCallConfEvent***
Direction: C → S
Service Parameters: ***activeCall, calledDevice***
Private Parameters: ***messageType***
ACK Parameters: ***newCall***
NAK Parameter: ***universalFailure***
Functional Description:

The Consultation Call Service places an existing active call (***activeCall***) at a device on hold and initiates a new call (***newCall***) from the same controlling device. This service provides the compound action of the Hold Call Service followed by Make Call Service. The DMS-100 NTS driver uses the CompuCALL DV-ADD-PARTY operation to perform this request. Please see reference ? for details of switch operation and feature interactions.

The Consultation Call Service request is acknowledged (Ack) by the switch if the switch is able to put the ***activeCall*** on hold and initiates a new call.

The request is negatively acknowledged if

- switch fails to put ***activeCall*** on hold (for example, call is in alerting state)
- switch fails to initiate a new call (for example, invalid parameter)

If the request is negatively acknowledged, the ***activeCall*** may return to active state, if it was in active or held state.

Service Parameters:

activeCall	[mandatory] A valid connection identifier indicates the *connection* to be placed on hold. This party must be in the active (talking) state. The device associated with the **activeCall** must be a station. The deviceID in ***activeCall*** must be the *ACD position number* of the controlling device.
calledDevice	[mandatory] Must be a valid number in the dialing plan for the device associated with the **activeCall** .

Ack Parameters:

newCall	[mandatory] A connection identifier indicates the connection between the controlling device and the new call. The ***newCall*** parameter contains the callID of the call and the station extension of the controlling device.

Nak Parameter:

universalFailure

INVALID_CSTA_DEVICE_IDENTIFIER (12)
An invalid device identifier or extension is specified in ***activeCall***.

INVALID_CSTA_CONNECTION_IDENTIFIER (13)
The connection identifier contained in the request is invalid or does not correspond to a station.

INVALID_CALLED_DEVICE
Called device has invalid device ID.

DRIVER_RESOURCE_LIMITATION
(no free slot to put new device info)
DMS-100 NTS Driver NLM has resource problem.

CONFERENCE_MEMBER_LIMIT_EXCEEDED
Total conference members exceeded the maximum number allowed.

INVALID_DESTINATION
Invalid destination device ID.

INVALID_OBJECT_STATE
The connections specified in the request are not in the valid states for the operation to take place.

REQUEST_INCOMPATIBLE_WITH_OBJECT
Object is in invalid state.

NO_ACTIVE_CALL (24)
The party to be put on hold is not currently active (for example, in alerting state), so it cannot be put on hold.

Detailed Information:

Phone sets associated with the service should be monitored. The phone set should be configured to allow the maximum number of conference members.

Device identifier is limited to 33 characters.

Private data is suggested for Transfer Call. Otherwise, the user receives a ring twice, as a result of the first call being torn down and the implementation of the second call.

Syntax

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

// cstaConsultationCall() - Service Request
```

```
RetCode_t         cstaConsultationCall (
            ACSHandle_t                 acsHandle,
            InvokeID_t                  invokeID,
            ConnectionID_t              *activeCall,        // devIDType= STATIC_ID
            DeviceID_t                  *calledDevice,
            NTPrivateData_t             *privateData);
```

// ***CSTAConsultationCallConfEvent*** **-** Service Response

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

typedef struct
{
            ACSEventHeader_t            eventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t          invokeID;
                                    union
                                    {
                                                CSTAConsultationCallConfEvent_t               consultationCall;
                                    } u;
                        } cstaConfirmation;
            } event;
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
            ConnectionID_t              newCall;
} CSTAConsultationCallConfEvent_t;

}
```

Make Call Service

Function: ***cstaMakeCall(), CSTAMakeCallConfEvent***
Direction: C → S
Service Parameters: ***callingDevice, calledDevice***
Private Parameters: ***noData***
Ack Parameters: ***newCall***
Nak Parameter: ***universalFailure***
Functional Description:

The Make Call Service originates a call between two devices. The service attempts to create a new call and establish a connection with the originating device (***callingDevice***). The Make Call Service also provides a connection identifier (***newCall***) that indicates the connection of the originating device in the ***CSTAMakeCallConfEvent***. The DMS-100 NTS driver uses the CompuCALL DV-MAKE-CALL operation to perform this request. Please see reference ? for details of switch operation and feature interactions.

The client application uses this service to set up a call on behalf of a station extension (calling party) to an on- or off-Switch endpoint (***calledDevice***). This service can be used by many types of applications such as Office Automation, Messaging, and Outbound Call Management (OCM) applications for Preview Dialing.

For the originator to place the call, the ***callingDevice*** (display or voice) must be an idle Ready ACD position and must not be in the talking (active).

If the originator is off-hook busy, the call cannot be placed and the request is denied (RESOURCE_BUSY). If the originator is unable to originate for other reasons (see ***universalFailure***), the switch denies the request.

Service Parameters:

callingDevice	[mandatory] Must be an idle ACD position.
calledDevice	[mandatory] Must be a valid number in the dialing plan for the device associated with the **activeCall**.

Ack Parameters:

newCall	[mandatory] A connection identifier indicates the connection between the originating device and the call. The ***newCall*** parameter contains the callID of the call and the ACD position of the ***callingDevice***.

Nak Parameter:

Nak Parameter:

A Make Call request is denied if the request fails before the call is attempted by the switch:

universalFailure

RESOURCE_LIMITATION_REJECTION

Driver does not have enough resource.

RESOURCE_BUSY

Origination phone is in busy state.

REQUEST_INCOMPATIBLE_WITH_OBJECT

The object is not in valid state.

INVALID_CALLING_DEVICE (5)

The ***callingDevice*** is out of service or not administered correctly in the switch.

INVALID_CSTA_DEVICE_IDENTIFIER (12)

An invalid device identifier or extension is specified in ***callingDevice.***

Detailed Information:

ACD Destination - When the destination is an agent position or an ACD group, ACD call delivery rules apply.

ACD Originator - Make Call Service cannot have an ACD group as the ***callingDevice***.

Call Destination - If the ***calledDevice*** is on-Switch station, the user at the station receives alerting. The user is alerted according to the call type (ACD or normal). Call delivery depends on the call type, station type, station administered options (manual/auto answer, call waiting, and so on), and station's talk state.

For example, for an ACD call, if the user is off-hook idle and in call forcing mode, the call is cut-through immediately. If the user is off-hook busy and has a multifunction-function set, the call alerts a free appearance. If the user is off-hook busy and has an analog set, and the user has "call waiting", the analog station user is given the "call waiting tone". If the user is off-hook busy on an analog station and does not have "call waiting", the calling endpoint hears busy. If the user is off-hook, alerting is started.

Call Forwarding All Calls - A Make Call Service to a station (***calledDevice***) with the Call Forwarding All Calls feature active redirects to the "forwarded to" station.

Class of Service (COS) - The Class of Service for the ***callingDevice*** is checked for the Make Call Service.

Data Calls - Data calls cannot be originated via the Make Call Service.

Display - If the ***callingDevice*** has a display set, the display shows the ***calledDevice***'s extension and name, if the ***calledDevice*** is on-switch, or the name of the trunk group, if the ***calledDevice*** is off-Switch. If the ***calledDevice*** is on-switch, normal display interactions apply for ***calledDevice*** with displays.

Single-Digit Dialing - Make Service request accepts single-digit dialing (for example, 0 for operator).

The ACD position associated with the service should be monitored.

Syntax

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

// cstaMakeCall() - Service Request

RetCode_t       cstaMakeCall (
        ACSHandle_t             acsHandle,
        InvokeID_t              invokeID,
        DeviceID_t              *callingDevice,
        DeviceID_t              *calledDevice,
        PrivateData_t           *privateData);


// CSTAMakeCallConfEvent - Service Response

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTACONFIRMATION
        EventType_t     eventType;      // CSTA_MAKE_CALL_CONF
} ACSEventHeader_t;

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

typedef struct
{
        ConnectionID_t          newCall;        // devIDType= STATIC_ID
} CSTAMakeCallConfEvent_t;
```

Transfer Call Service

NOTE: The Transfer Call Service must always be preceded by a Consultation Call Service. If the Transfer Call request is received prior to a Consultation Call request, the system returns the following error message: "GENERIC_STATE_INCOMPATIBILITY (21)". For more information on the subject, refer to the "Call Scenarios and Events" chapter in this guide.

Function: ***cstaTransferCall(), CSTATransferCallConfEvent***
Direction: C → S
Service Parameters: ***heldCall, activeCall***
Ack Parameters: ***newCall, connList***
Nak Parameter: ***universalFailure***
Functional Description:

This service provides the transfer of an existing held call (***heldCall***) and another active or proceeding call (alerting, queued, held, or connected) (***activeCall***) at a device, provided that ***heldCall*** and ***activeCall*** are not both in the alerting state at the controlling device. The Transfer Service releases the Consultation Call and initiates a new call to complete the transfer. Also, both of the connections to the common device become Null and their connectionIDs are released. A connectionID that specifies the resulting new connection for the transferred call is provided. The DMS-100 NTS driver uses the CompuCALL DV-TRANSFER-PARTY operation to perform this request. Please see reference ? for details of switch operation and feature interactions.

Service Parameters:

heldCall	[mandatory] Must be a valid connection identifier for the call which is on hold at the controlling device and is to be transferred to the ***activeCall***. The deviceID in ***heldCall*** must contain the *station extension* of the controlling device.
activeCall	[mandatory] Must be a valid connection identifier of an active or proceeding call at the controlling device to which the ***heldCall*** is to be transferred. The deviceID in ***activeCall*** must contain the *station extension* of the controlling device.

Ack Parameters:

newCal*l*	[mandatory - supported] A connection identifier that specifies the resulting new call identifier for the transferred call.
connList	[optional - not supported] Specifies the devices on the resulting newCall. This includes a count of the number of devices in the new call and a list of up to six connectionIDs and up to six deviceIDs which define each connection in the call. If a device is on-Switch, the extension is specified. It may contain alerting extensions. If a party is off-Switch, then its static device identifier is specified.

Nak Parameter:

universalFailure

INVALID_CSTA_DEVICE_IDENTIFIER (12)

An invalid device identifier or extension is specified in ***heldCall*** or ***activeCall***.

INVALID_CSTA_CONNECTION_IDENTIFIER (13)

The controlling deviceID in ***activeCall*** or ***heldCall*** has not been specified correctly.

INVALID_OBJECT_STATE (22)

The connections specified in the request are not in the valid states for the operation to take place. For example, it does not have one call active and one call in the held state as required.

INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23)

The callID in ***activeCall*** or ***heldCall*** has not been specified correctly.

NO_CALL_TO_COMPLETE (29)

There is no call to complete the transfer.

Detailed Information:

Phone sets associated with the service should be monitored.

Syntax

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

// cstaTransferCall() - Service Request

RetCode_t       cstaTransferCall (
        ACSHandle_t             acsHandle,
        InvokeID_t              invokeID,
        ConnectionID_t          *heldCall,      // devIDType = STATIC_ID
        ConnectionID_t          *activeCall,    // devIDType = STATIC_ID
        PrivateData_t           *privateData);


// CSTATransferCallConfEvent - Service Response

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTACONFIRMATION
        EventType_t     eventType;      // CSTA_TRANSFER_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        InvokeID_t      invokeID;
                        union
                        {
                                CSTATransferCallConfEvent_t     transferCall;
                        } u;
                } cstaConfirmation;
        } event;
} CSTAEvent_t;

typedef struct Connection_t {
    ConnectionID_t          party;
    DeviceID_t    staticDevice;         // NULL for not present
} Connection_t;

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

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

Set Feature Service Group

Overview

These services allow a client application to set switch-controlled features or values on a DMS-100 device.

The following CSTA Services are supported in the NetWare Telephony Services product:

Set Agent State Service

Set Agent State Service

Function: ***cstaSetAgentState(),*** **C*STASetAgentStateConfEvent***
Direction: C → S
Service Parameters: ***device, agentMode, agentID, agentGroup, agentPassword***
Ack Parameters: ***noData***
Nak Parameter: ***universalFailure***
Functional Description:

This service allows a client to log an ACD Agent into or out of a DMS-100 ACD group, and to specify a change of work mode for a DMS-100 ACD Agent.

Service Parameters:

device	[mandatory] Specifies the agent position ID. This must be a valid on-Switch station position ID for an ACD Agent.
agentMode	[mandatory] Specifies to log in or log out an Agent for an ACD, or a change of work mode for an Agent logged into an ACD as follows: • AM_LOG_IN - Log in the Agent so that they can contribute to the activities of the ACD. This does not imply that the Agent is ready to accept calls. The initial mode for the ACD Agent is AM_NOT_READY. • AM_LOG_OUT - Log an Agent out of a specific ACD. The Agent is unable to accept additional calls for the ACD. • AM_NOT_READY - Change the work mode for an Agent logged into an ACD to "Not Ready", indicating that the Agent is occupied with some task other than serving a call. • AM_READY - Change the work mode for Agent logged on to an ACD to "Ready". The Agent in the Ready state is ready to accept calls or is currently busy with an ACD call.
agentID	[optional] Specifies the Agent login identifier for the ACD Agent. This parameter is mandatory when the ***agentMode*** parameter is AM_LOG_IN; otherwise, it is ignored.
agentGroup	[mandatoryNot Supported] This Service Parameter is ignored. Specifies the ACD Agent Group to login, logout, or change the agent work mode to "*Not Ready*", "*Ready*", *or "Work Not Ready*". The ACD group defined by switch datafill is used.
agentPassword	[optional] Specifies a password that allows an ACD Agent to log into an ACD Group. Also used to specifiy the "Walk-away" code when setting the agent work mode to "*Not Ready".*

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure

GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41)

The user has not subscribed for the feature.

INVALID_CSTA_DEVICE_IDENTIFIER (12)

AN invalid device identifier has been specified in ***device***.

Detailed Information:

A request to login an ACD Agent (***agentMode*** is AM_LOG_IN) sets the initial Agent work state to "Not Ready".

Syntax

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

// cstaSetAgentState() - Service Request to change ACD Agent        work mode

RetCode_t       cstaSetAgentState(
        ACSHandle_t             acsHandle,
        InvokeID_t              invokeID,
        DeviceID_t              *device,
        AgentMode_t             agentMode,
        AgentID_t               *agentID,
        AgentGroup_t            *agentGroup,
        AgentPassword_t         *agentPassword,
        PrivateData_t           *privateData);

typedef char                    DeviceID_t[64];

typedef enum AgentMode_t {
        AM_LOG_IN                       = 0,
        AM_LOG_OUT                      = 1,
        AM_NOT_READY                    = 2,
        AM_READY                        = 3,
        AM_WORK_NOT_READY  = 4,
        AM_WORK_READY                   = 5
} AgentMode_t;

typedef char                    AgentID_t[32];

typedef DeviceID_t              AgentGroup_t;

// CSTASetAgentStateConfEvent

// ACS Event Header
typedef struct
{
        ACSHandle_t             acsHandle;
        EventClass_t            eventClass;     // CSTACONFIRMATION
        EventType_t             eventType;      // CSTA_SET_AGENT_STATE_CONF
} ACSEventHeader_t;

// CSTA Set Agent State Conf Event
typedef struct CSTASetAgentStateConfEvent_t {
        Nulltype                null;
} CSTASetAgentStateConfEvent_t;
```

```
// CSTA Event
typedef struct
{
            ACSEventHeader_t          eventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t          invokeID;
                                    union
                                    {
                                                CSTASetAgentStateConfEvent_t     setAgentState;
                                    } u;
                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Example

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

// cstaSetAgentState() - Request to login ACD Agent, initial work mode "Auto-In-Work Mode"

ACSHandle_t             acsHandle;                              // An open ACS Stream handle
InvokeID_t              invokeID        = 1;                    // Application generated Invoke ID
DeviceID_t              *device         = "12345";              // Device associated with ACD Agent
AgentMode_t             agentMode       = AM_LOG_IN;            // Service Request to login ACD Agent
AgentID_t               *agentID        = "01";                 // Agent login identifier
AgentGroup_t            *agentGroup     = "11111";              // ACD to log Agent into
AgentPassword_t         *agentPassword  = NULL;                 // No password, ie. not EAS
RetCode_t               retcode;                                // Return Code from service request

            PrivateData = Null;
            retcode =           cstaSetAgentState( acsHandle, invokeID, device, agentMode, agentID,
                                            agentGroup, agentPassword, privateData );
            if ( retcode == 0 ) {
                                // SUCCESSFUL Request, Now wait for confirmation
            }

// CSTASetAgentStateConfEvent - Agent logged in, initial work mode "Auto-In-Work Mode"

CSTAEvent_t       cstaEvent;

            retcode = acsGetEventBlock( acsHandle, (void *) &cstaEvent, sizeof(CSTAEvent_t),
                                            NULL, NULL );                // No Private Data Returned

            if ( retcode == ACSPOSITIVE_ACK ) {
                        if (( cstaEvent.eventHeader.eventClass == CSTACONFIRMATION ) &&
                                    (cstaEvent.eventHeader.eventType == CSTA_SET_AGENT_STATE_CONF)) {
                                    if (cstaEvent.event.cstaConfirmation.invokeID == 1) {
                                                // Invoke ID matches, Set Agent State is confirmed
                                    }
                        }
            }
```

Monitor Service Group

Overview

All monitor services for the DMS-100 NetWare Telephony Services have the same confirmation event. The monitor device service and the monitor calls via device service provide call event reports on all calls at a different station set. The Monitor Stop Service is used to cancel a monitor service of any type.

MONITOR DEVICE SERVICE - *CSTAMONITORDEVICE()*

This service provides call event reports for all devices on all calls at a station device. Event reports are provided for calls that occurred prior to the monitor request and arrive at the device after the monitor request is acknowledged. No further events of a call are reported if that call is dropped, forwarded, or transferred from the device, and the device ceases to participate in the call.

The service also provides agent event reports passed by the filter for a monitored ACD device.

The service does not provide maintenance event reports.

MONITOR STOP SERVICE - *CSTAMONITORSTOP()*

An application uses this service to cancel a previously requested Monitor Service.

Monitor Device Service

Function: ***cstaMonitorDevice(), CSTAMonitorConfEvent***
Direction: C S
Service Parameters: ***deviceID, monitorFilter***
Private Parameters: ***attPrivateFilter***
Ack Parameters: ***monitorCrossRefID, monitorFilter***
Nak Parameter: ***universalFailure***
Functional Description:

This service provides call event reports passed by the call filter for all devices on all calls at a device. Event reports are provided for calls that occurred previous to the monitor request and arrive at the device after the monitor request is acknowledged. No further events of a call are reported, if that call is dropped, forwarded, transferred, or conferenced, and the device ceases to participate in the call.

The Call Cleared Event is never provided for this service. There are no subsequent event reports for a call after a Connection Cleared Report has been received for this service. Reporting of the subsequent call event reports after a Transferred Event Report is dependent on whether the call is merged-in or merged-out from the monitored device.

The event reports are provided for endpoints connected to the DMS-100 switch and Associated with the CompuCALL environment.

This service supports Call Event Reports and Feature Event Reports for line devices and supports Agent Event Reports for ACD devices.

Maintenance Event Reports and Private Filter are not supported.

Service Parameters:

deviceID	[mandatory] Must be a valid Switch line, ACD Group Primary DN, or ACD agent position which is to be monitored.
monitorFilter	[optional - not supported] Specifies the filters to be used with deviceID. Call Filter/Event Reports and Feature Filter/Event Reports are supported for station device.

Ack Parameters:

monitorCrossRefID	[mandatory] Contains the handle chosen by the DMS-100 NTS Driver NLM. This handle is a unique value within an ***acsOpenStream*** session for the duration of the monitor and is used by the application to correlate subsequent event reports to the monitor request that initiated them. It also allows the correlation of the Monitor Stop to the original Monitor Service request.
monitorFilter	[optional - not supported] Specifies the event reports that are to be filtered out on the object being monitored by the application. This may not be the ***monitorFilter*** specified in the service request, because filters for events that are not supported by the DMS-100 switch and filters for events that do not apply to the monitored device are always turned on in ***monitorFilter.***.

Nak Parameter:

universalFailure

INVALID_CSTA_DEVICE_IDENTIFIER (12)

An invalid device identifier or extension is specified in ***deviceID***.

OBJECT_MONITOR_LIMIT_EXCEEDED (42)

The request cannot be executed because the system limit would be exceeded for the maximum number of monitors.

Detailed Information:

used handled by DMS-100 NTS driver to determine where event reports are to be sent. ASSOC.CFG file controlls DN-ASSOCIATE messages passed to switch. Ref DN-Associate in CompuCALL spec.
ACD Group - An ACD group can be monitored by this service.

Analog ports - Analog ports equipped with modems can be monitored by the ***cstaMonitorDevice*** Service.

Feature Access - Monitoring a station does not prohibit users from access to any enabled switch features. A monitored station can access any enabled switch feature.

Multiple Requests - Multiple requests per device are supported.

Also see Event Report "Detailed Information" section.

Syntax

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

// cstaMonitorDevice() - Service Request

RetCode_t       cstaMonitorDevice (
        ACSHandle_t                     acsHandle,
        InvokeID_t                      invokeID,
        DeviceID_t                      *deviceID,
        CSTAMonitorFilter_t             *monitorFilter,     // not supported
        PrivateData_t                   *privateData),

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTACONFIRMATION
        EventType_t     eventType;      // CSTA_MONITOR_CONF
} ACSEventHeader_t;

// CSTAMonitorConf - Event

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        InvokeID_t       invokeID;
                        union
                        {
                                CSTAMonitorConfEvent_t          monitorStart;
                        } u;
                } cstaConfirmation;
        } event;
} CSTAEvent_t;

typedef struct CSTAMonitorConfEvent_t {
    CSTAMonitorCrossRefID_t             monitorCrossRefID;
    CSTAMonitorFilter_t                 monitorFilter;
} CSTAMonitorConfEvent_t;

typedef long        CSTAMonitorCrossRefID_t;
```

Event Report Service Group

NOTE: Unsolicited events will only be received for monitored devices.

CSTAEventCause and *LocalConnectionState*

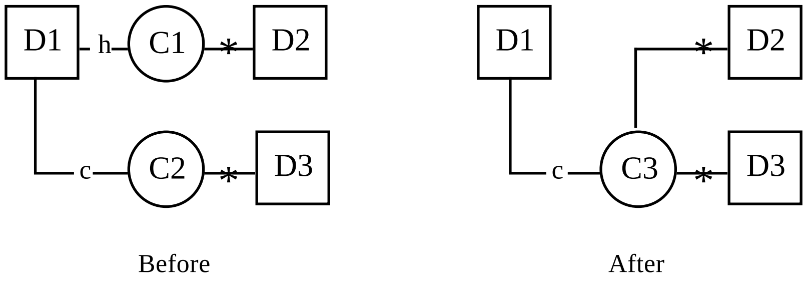
Following are the definitions of the enumerated types *CSTAEventCause* and *LocalConnectionState*. These data structures are used extensively by the Event Report Service Group members described in this chapter.

```
typedef enum CSTAEventCause_t {
      EC_NONE                          = -1,     // no cause value is specified
      EC_ACTIVE_MONITOR                = 1,
      EC_ALTERNATE                     = 2,
      EC_BUSY                          = 3,
      EC_CALL_BACK                     = 4,
      EC_CALL_CANCELLED                = 5,
      EC_CALL_FORWARD_ALWAYS           = 6,
      EC_CALL_FORWARD_BUSY             = 7,
      EC_CALL_FORWARD_NO_ANSWER        = 8,
      EC_CALL_FORWARD                  = 9,
      EC_CALL_NOT_ANSWERED             = 10,
      EC_CALL_PICKUP                   = 11,
      EC_CAMP_ON                       = 12,
      EC_DEST_NOT_OBTAINABLE           = 13,
      EC_DO_NOT_DISTURB                = 14,
      EC_INCOMPATIBLE_DESTINATION      = 15,
      EC_INVALID_ACCOUNT_CODE          = 16,
      EC_KEY_CONFERENCE                = 17,
      EC_LOCKOUT                       = 18,
      EC_MAINTENANCE                   = 19,
      EC_NETWORK_CONGESTION            = 20,
      EC_NETWORK_NOT_OBTAINABLE        = 21,
      EC_NEW_CALL                      = 22,
      EC_NO_AVAILABLE_AGENTS           = 23,
      EC_OVERRIDE                      = 24,
      EC_PARK                          = 25,
      EC_OVERFLOW                      = 26,
      EC_RECALL                        = 27,
      EC_REDIRECTED                    = 28,
      EC_REORDER_TONE                  = 29,
      EC_RESOURCES_NOT_AVAILABLE       = 30,
EC_SILENT_MONITOR                  = 31,
        EC_TRANSFER                             = 32,
        EC_TRUNKS_BUSY                          = 33,
        EC_VOICE_UNIT_INITIATOR                 = 34
} CSTAEventCause_t;
typedef enum LocalConnectionState_t {
        CS_UNKNOWN                              = -2,
        CS_NONE                                 = -1,
        CS_NULL                                 = 0,
        CS_INITIATE                             = 1,
        CS_ALERTING                             = 2,
        CS_CONNECT                              = 3,
        CS_HOLD                                 = 4,
        CS_QUEUED                               = 5,
        CS_FAIL                                 = 6
} LocalConnectionState_t;
```

Conferenced Event

Function: C*STAConferencedEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, primaryOldCall, secondaryOldCall, confController, addedParty, conferenceConnections, localConnectionInfo, cause*
Functional Description:

The Conference Event Report indicates that two calls are conferenced (merged) into one, and no parties are removed from the resulting call in the process. The event may include up to six parties on the resulting call.



The Conferenced Event Report is generated when an application processor successfully completes a ***cstaConferenceCall*** request.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
primaryOldCall	[mandatory] Specifies the callID of the call that was conferenced. This is usually the held call before the conference. This call ended as a result of the conference.
secondaryOldCall	[mandatory] Specifies the callID of the call that was conferenced. This is usually the active call before the conference. This call is retained by the switch after the conference.
confController	[mandatory] Specifies the device which is controlling the conference. This is the device which sets up the conference.
addedParty	[mandatory] Specifies the new conferenced-in device.

conferenceConnections	[optional - supported] Specifies a count of the number of devices and a list of up to six connectionIDs and up to six deviceIDs which resulted from the conference.
localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - supported] Specifies the reason for this event.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

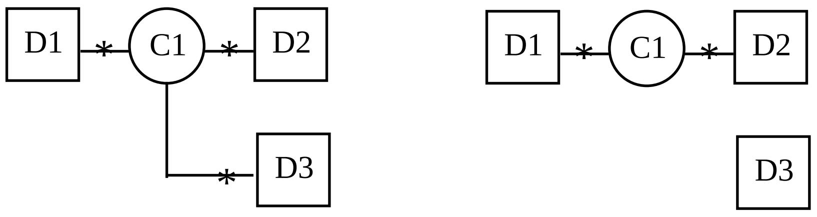
Syntax

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

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_CONFERENCED
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t                 monitorCrossRefId;
                        union
                        {
                                CSTAConferencedEvent_t          conferenced;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;

typedef struct CSTAConferencedEvent_t {
        ConnectionID_t          primaryOldCall;
        ConnectionID_t          secondaryOldCall;
        SubjectDeviceID_t       confController;
        SubjectDeviceID_t       addedParty;
        ConnectionList_t                conferenceConnections;
        LocalConnectionState_t  localConnectionInfo;
        CSTAEventCause_t        cause;
} CSTAConferencedEvent_t;

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

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

Connection Cleared Event

Function: C*STAConnectionClearedEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, droppedConnection, releasingDevice, localConnectionInfo, cause*
Functional Description:

The Connection Cleared Event Report indicates that a device in a call disconnects or is dropped. It does not indicate that a transferring device has left a call in the act of transferring that call. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-CALL-RELEASED-U message. with ReleaseReason codes ...



Before After

A Connection Cleared Event Report is *not* generated when a party drops as a result of a transfer operation.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
droppedConnection	[mandatory] Specifies the connection which has been dropped from the call.
releasingDevice	[mandatory] Specifies the dropped device.
localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - supported] Specifies a cause when the call is not terminated normally. No cause is specified for normal call termination.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

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

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

```
typedef struct
{
          ACSEventHeader_t          eventHeader;
          union
          {
                    struct
                    {
                              CSTAMonitorCrossRefID_t              monitorCrossRefId;
                              union
                              {
                                        CSTAConnectionClearedEvent_t     connectionCleared;
                              } u;
                    } cstaUnsolicited;
          } event;
} CSTAEvent_t;

typedef struct CSTAConnectionClearedEvent_t {
          ConnectionID_t                droppedConnection;
          SubjectDeviceID_t             releasingDevice;
          SubjectDeviceID_t             localConnectionInfo;
          CSTAEventCause_t              cause;
} CSTAConnectionClearedEvent_t;
```

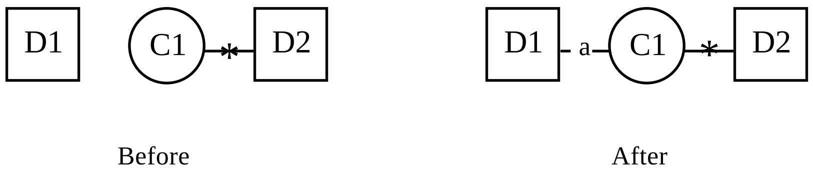
Delivered Event

Function: ***CSTADeliveredEvent***
Direction: C ← S
Service Parameters: ***monitorCrossRefID, connection, alertingDevice, callingDevice, calledDevice, lastRedirectionDevice, localConnectionInfo, cause***
Private Parameters: ***NT User Data-t***
Functional Description:

The DMS-100 switch reports two types of Delivered Event Reports. The *type* of the Delivered Event is ***not*** specified in the report. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-CALL-OFFERED-U message.

Call Delivered to a Line or ACD Position Device

A Delivered Event Report of this type indicates that "alerting" (tone, ring, and so on) is applied to a device or when the switch detects that "alerting" has been applied to a device. Delivered Events can be reported on devices which are ACD agent positions or Meridian Digital Switch lines.



Consecutive Delivered Event Reports are possible. Multiple Delivered Event Reports for multiple devices are also possible.

The switch generates the Delivered Event Report when the "Alerting" (tone, ring, and so on) is applied to a device or when the switch detects that "alerting" has been applied to a device.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
connection	[mandatory] Specifies the endpoint which is alerting.
alertingDevice	[mandatory] Specifies the device which is alerting. • If the call is delivered to an ACD, the monitored object is specified.
callingDevice	[mandatory] Specifies the calling device. The following rules apply: • For internal calls - the 10 digit North American Numbering Plan number for the extension placing the call, or, in the case of calls placed using the cstaMakeCall function, the INCALLS DN of the originating position..

	• For incoming calls over SS7 or PRI facilities - "calling number" from the ISDN SETUP is specified, if the "calling number" does not exist but the CompuCALL "chargenumber" does, then it is used. If neither is available (NULL). • For consultation calls...
calledDevice	[mandatory] Specifies the originally called device. The following rules apply: • For incoming ACD calls, The ACDDN paramter of the CompuCALL DV-CALL-OFFERED-U message. (The Primary or Supplementary DN used to access the ACD group.)
lastRedirectionDevice	[optional - not supported] Specifies the Primary ACDDN of the last ACD group where the DV-CALL_REDIRECT operation was used to redirect the call.
localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - supported] Specifies the reason for this event.

Private Parameters:

udataType	specifies a user-to-user ASCII data type.
udataLength	specifies the length of the user data.
udata	specifies user data.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

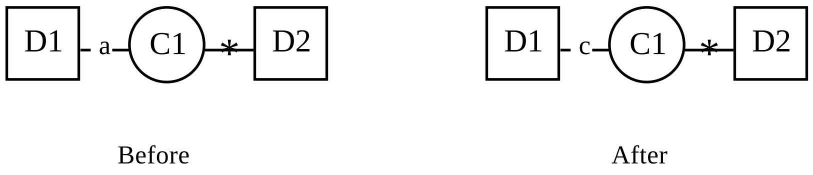
```
#include <csta.h>
#include <acs.h>

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_DELIVERED
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t         monitorCrossRefId;
                        union
                        {
                                CSTADeliveredEvent_t    delivered;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;

typedef struct CSTADeliveredEvent_t
```

```
{
            ConnectionID_t              connection;
            SubjectDeviceID_t           alertingDevice;
            CallingDeviceID_t           callingDevice;
            CalledDeviceID_t            calledDevice;
            RedirectionDevice_t         lastRedirectionDevice;      // not supported
            LocalConnectionState_t      localConnectionInfo;
            CSTAEventCause_t            cause;
} CSTADeliveredEvent_t;
```

Established Event

Function: C*STAEstablishedEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, establishedConnection, answeringDevice, callingDevice, calledDevice, lastRedirectionDevice, localConnectionInfo, cause*
Functional Description:

The Established Event Report indicates that switch detects that a device answers or connects to a call.



The Established Event Report is sent when a call is delivered to a monitored ACD position, MDC line or ACD group.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
establishedConnection	[mandatory] Specifies the endpoint which joined the call.
answeringDevice	[mandatory] Specifies the device which joined the call.
callingDevice	[mandatory] Specifies the calling device. The following rules apply: • For internal calls originated at an on-Switch station - the station's extension is specified. • For outgoing calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified, if the "calling number" does not exist (NULL).

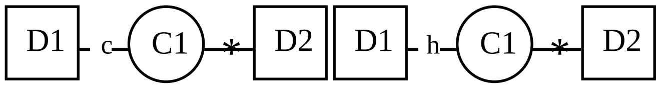
	• For incoming calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified, if the "calling number" does not exist (NULL). • For incoming calls over non-PRI facilities - the calling party number is generally not available. The assigned trunk identifier is provided instead. • The trunk group number is specified only when the calling party number is not available. • For calls originated at a bridged call appearance - the principal's extension is specified.
calledDevice	[mandatory- supported] Specifies the originally called device. The following rules apply: • For outgoing calls over PRI facilities - "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (NULL), the deviceID is ID_NOT_KNOWN. • For incoming calls over PRI facilities - "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (NULL), the deviceID is ID_NOT_KNOWN. • For incoming calls over non-PRI facilities - the principal extension is specified. If the switch is administered to modify the DNIS digits, then the modified DNIS string is specified. • If the called device is on-Switch and the call did not come over a PRI facility, the extension of the party dialed is specified.
lastRedirectionDevice	[optional - not supported] Specifies the previously alerted device in the case where the call was redirected or diverted to the ***answeringDevice***.
localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - supported] Specifies the reason for this event.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

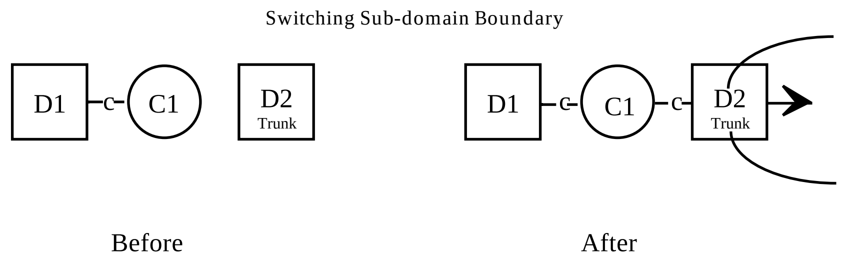
```
#include <csta.h>
#include <acs.h>

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_ESTABLISHED
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t             monitorCrossRefId;
                        union
                        {
                                CSTAEstablishedEvent_t  established;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;

typedef struct CSTAEstablishedEvent_t
{
        ConnectionID_t              establishedConnection;
        SubjectDeviceID_t           answeringDevice;
        CallingDeviceID_t           callingDevice;
        CalledDeviceID_t            calledDevice;
        RedirectionDevice_t         lastRedirectionDevice;      // not supported
        LocalConnectionState_t      localConnectionInfo;
        CSTAEventCause_t            cause;
} CSTAEstablishedEvent_t;
```



Network Reached Event

Function: C*STANetworkReachedEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, connection, trunkUsed, calledDevice, localConnectionInfo, cause*
Functional Description:

This event indicates that no further Event Report messages reporting the progress of a call should be expected as the call has left the portion of the switching domain which can report these events. As DMS-100 CompuCALL does not report progress on outgoing calls initiated by either the DV-MAKE-CALL or DV-ADD-PARTY operations, the DMS-100 NTS driver generates a Network Reached Event message to notify applications that no further messages may be received on the call.if a subsequent DV-CALL-QUEUED, DV-CALL-OFFERED-U or DV-CALL-ANSWERED message is not received within a programmable time period after the call was placed.



This event report implies that there is a reduced level of event reporting and possibly no additional device feedback, except disconnect/drop, provided for this party in the call.

A switch may receive multiple PROGress messages for any given call; each will generate a Network Reached Event Report.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
connection	[mandatory] Specifies the endpoint for the outbound connection to another network.
trunkUsed	[mandatory] Specifies the trunk identifier that was used to establish the connection.
calledDevice	[mandatory - supported] Specifies the destination device of the call.

localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - supported] Specifies the reason for this event.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

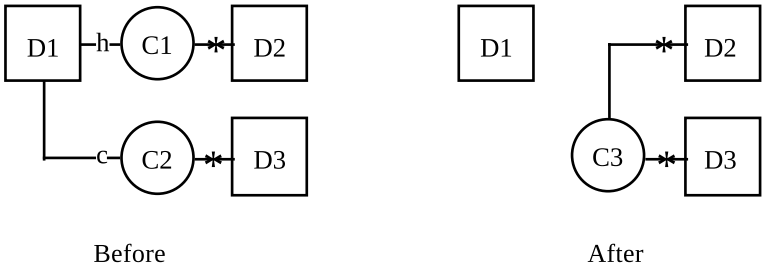
Syntax

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

typedef struct
{
            ACSHandle_t      acsHandle;
            EventClass_t     eventClass;        // CSTAUNSOLICITED
            EventType_t      eventType;         // CSTA_NETWORK_REACHED
} ACSEventHeader_t;

typedef struct
{
            ACSEventHeader_t            eventHeader;
            union
            {
                        struct
                        {
                                    CSTAMonitorCrossRefID_t             monitorCrossRefId;
                                    union
                                    {
                                                CSTANetworkReachedEvent_t       networkReached;
                                    } u;
                        } cstaUnsolicited;
            } event;
} CSTAEvent_t;

typedef struct CSTANetworkReachedEvent_t
{
            ConnectionID_t              connection;
            SubjectDeviceID_t           trunkUsed;
            CalledDeviceID_t            calledDevice;
            LocalConnectionState_t      localConnectionInfo;
            CSTAEventCause_t            cause;
} CSTANetworkReachedEvent_t;
```

Transferred Event

Function: C*STATransferredEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, primaryOldCall, secondaryOldCall, transferringDevice, transferredDevice, transferredConnections, localConnectionInfo, cause*
Functional Description:

The Transferred Call Event Report indicates that an existing call was transferred to another device and the device requesting the transfer has been dropped from the call. The ***transferringDevice*** does not appear in any future feedback for the call. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-CALL-RELEASED-U message. with a ReleaseReason code of Transferred.



The Transferred Event Report is generated when an adjunct successfully completes a ***cstaTransferCall*** request.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
primaryOldCall	[mandatory] Specifies the callID of the call that was transferred. This is usually the held call before the transfer. This call ended as a result of the transfer.
secondaryOldCall	[mandatory] Specifies the callID of the call that was transferred. This is usually the active call before the transfer. This call is retained by the switch after the transfer.
transferringDevice	[mandatory] Specifies the device which is controlling the transfer. This is the device which did the transfer.
transferredDevice	[mandatory] Specifies the new transferred-to device.

transferredConnections	[optional - supported] Specifies a count of the number of devices and a list of up to six connectionIDs and up to six deviceIDs which resulted from the transfer.
localConnectionInfo	[optional - supported] Specifies the connection state of the monitored device for this call.
cause	[optional - not supported] Specifies the reason for this event.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

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

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_TRANSFERRED
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t         monitorCrossRefId;
                        union
                        {
                                CSTATransferredEvent_t  transferred;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;

typedef struct CSTATransferredEvent_t
{
        ConnectionID_t          primaryOldCall;
        ConnectionID_t          secondaryOldCall;
        SubjectDeviceID_t       transferringDevice;
        SubjectDeviceID_t       transferredDevice;
        ConnectionList_t                transferredConnections;
        LocalConnectionState_t  localConnectionInfo;
        CSTAEventCause_t        cause;                  // not supported
        } CSTATransferredEvent_t;

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

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

Logged On Event

Function: C*STALoggedOnEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, agentDevice, agentID, agentGroup, password*
Functional Description:

This event report informs the application that an agent has logged into an ACD. An application needs to request a ***cstaMonitorDevice*** on the ACD position in order to receive this event. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-LOGGED-ON-U message.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
agentDevice	[mandatory] Indicates the position ID of the agent that is logging on.
agentID	[optional - supported] Indicates the agent identifier.
agentGroup	[optional - supported] Indicates the ACD group that is being logged on.
password	[optional - not supported] Indicates the agent password for logging in.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

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

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_LOGGED_ON
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t         monitorCrossRefId;
                        union
                        {
                                CSTAServiceInitiatedEvent_t     loggedOn;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;
```

```
typedef struct CSTALoggedOnEvent_t
{
        SubjectDeviceID_t       agentDevice;
        AgentID_t               agentID;
        AgentGroup_t            agentGroup;
        AgentPassword_t         password;
} CSTALoggedOnEvent_t;
```

Logged Off Event

Function: C*STALoggedOffEvent*
Direction: C ← S
Service Parameters: *monitorCrossRefID, agentDevice, agentID, agentGroup*
Functional Description:

This event report informs the application that an agent has logged out of an ACD. An application needs to request a ***cstaMonitorDevice*** on the ACD position in order to receive this event. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-LOGGED-ON-U message.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
agentDevice	[mandatory] Indicates the positonID of the agent that is logging on.
agentID	[optional] Indicates the agent identifier.
agentGroup	[optional - supported] Indicates the ACD that is being logged on.

Detailed Information:

Also see the Event Report "Detailed Information" section at the end of this chapter.

Syntax

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

typedef struct
{
        ACSHandle_t     acsHandle;
        EventClass_t    eventClass;     // CSTAUNSOLICITED
        EventType_t     eventType;      // CSTA_LOGGED_OFF
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t         monitorCrossRefId;
                        union
                        {
                                CSTAServiceInitiatedEvent_t     loggedOff;
                        } u;
                } cstaUnsolicited;
        } event;
} CSTAEvent_t;

typedef struct CSTALoggedOffEvent_t
{
        SubjectDeviceID_t       agentDevice;
        AgentID_t                    agentID;
        AgentGroup_t                 agentGroup;
} CSTALoggedOffEvent_t;
```

NotReady Event

Function: *CSTANotReadyEvent*
Direction: C ← S
Service *Parameter: monitorCrossRefID, agentDevice, agentID*
Functional Description:

The Not Ready event indicates that an agent is busy with tasks other than servicing a call at the device. In most cases, this implies that the agent is not ready to receive a call or that the agent is taking a break. An application needs to request a ***cstaMonitorDevice*** on the ACD position in order to receive this event. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-NOT-READY-U message.

Service Parameter:

monitorCrossRefID	(mandatory) Contains the handle to the CSTA association for which this event is associated.
agentDevice	(mandatory) Specifies the device from which the agent is logged on to the system.
agentID	(mandatory) Specifies the agent login ID.

Syntax

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

typedef struct
{
        ACSHandle_      acsHandle;
        EventClass_t    eventClass;     //CSTAUNSOLICITED
        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 CSTANotReadyEvent_t {
    SubjectDeviceID_t   agentDevice;
    AgentID_t           agentID;
} CSTANotReadyEvent_t;
```

Ready Event

Function: ***CSTAReadyEvent***
Direction: C ← S
Service parameter: ***monitorCrossRefID, agentDevice, agentID***
Functional Description:

The ready event indicates that an agent is ready to receive calls at the device. An application needs to request a ***cstaMonitorDevice*** on the ACD position in order to receive this event. The DMS-100 NTS driver generates this event upon receiving the CompuCALL DV-READY-U message.

Service Parameter:

monitorCrossRefID	(mandatory) Contains the handle to the CSTA association for which this event is associated.
agentDevice	(mandatory) Specifies the device which is ready to receive calls from the ACD.
agentID	(mandatory) Specifies the identifier of the agent who is ready to receive calls.

Syntax

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

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

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        CSTAMonitorCrossRefID_t         monitorCrossRefId;
                        union
                        {
                                CSTAReadyEvent_t        ready;
                        } u;
                }cstaUnsolicited;
        } event;
}CSTAEvent_t;

typedef struct CSTAReadyEvent_t {
    SubjectDeviceID_t   agentDevice;
    AgentID_t           agentID;
} CSTAReadyEvent_t;
```

Event Report Detailed Information

EVENT REPORTS FOR MONITORED OBJECTS

The following table shows the type of monitored objects over which various event reports can be sent to a monitor request on the objects.

Event Report	Station	ACD
Conferenced	Yes	No
Connection Cleared	Yes	Yes
Delivered	Yes	No
Established	Yes	Yes
Network Reached	Yes	?
Queued	Yes	No
Transferred	Yes	Yes

CONFERENCE

The Conferenced Event Report is only sent if the conference is initiated by NTS.

RLS BUTTON OPERATION

When the RLS button is pushed by one party in a two-party call, the Connection Cleared Event Report is sent to both parties if they have requested Connection Cleared Event service.

HOLD

Manually holding a call (either by using the Hold, Conference, Transfer buttons, or switch-hook flash) does not cause a Held Event Report to be sent to all monitor requests for this call, including the held device.

TRANSFER

Manual transfer from a station monitored by a ***cstaMonitorDevice*** request is allowed subject to the feature's restrictions. Transferred Event Reports are sent using available information.

Computing Function Service Groups

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, callerentered digits (provided as private data), or information in an application database to route incoming calls.

Application Call Routing

Application call routing requires that the switch be configured to direct calls to a special type of device know 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 cstaRouteRegisterReq( ) 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.

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.

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.

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.

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 applicationprovided destination when switch features such as call forwarding redirect the call.

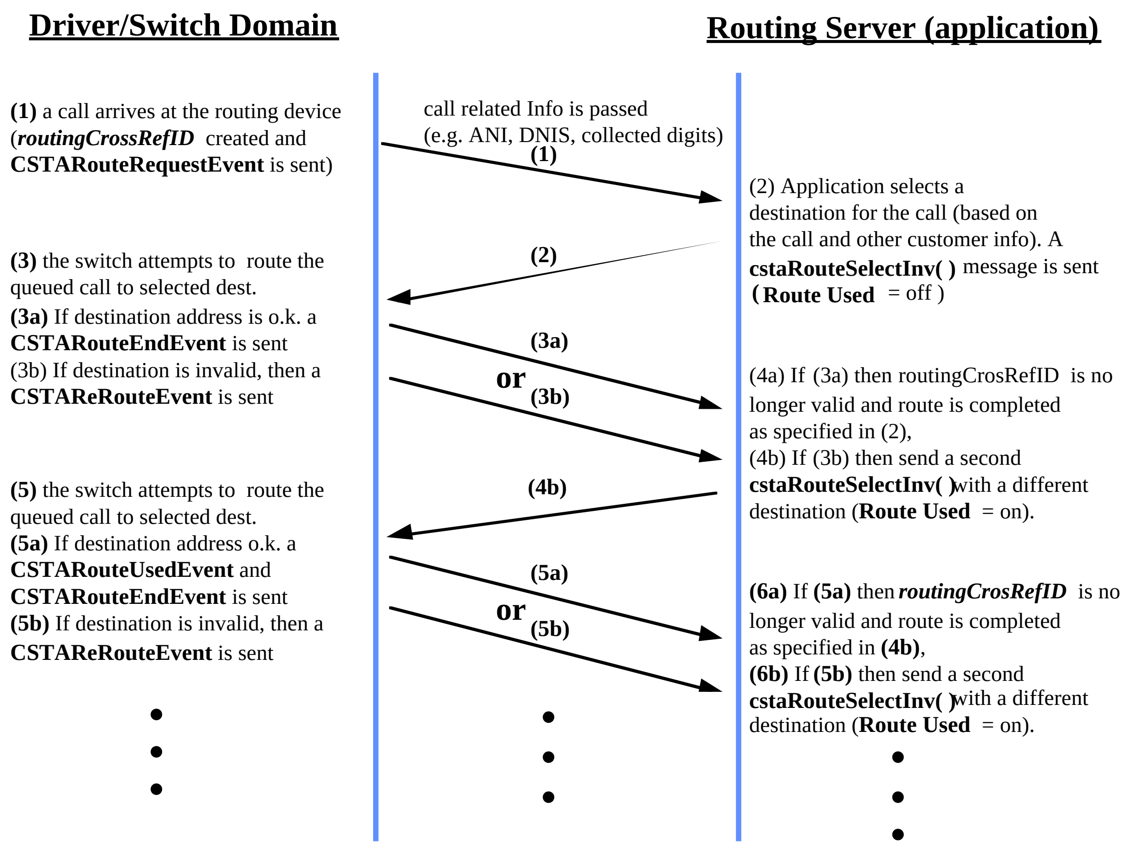
The switch receives the route select message (cstaRouteSelectInv) and attempts to route the call to the application-provided destination. If the destination is valid, the switch routes the call to that destination and sends the application a route end event (cstaRouteEndInv). This terminates the routing 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.

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.

The Figure below illustrates the Routing Procedure.


Driver/Switch Domain
Routing Server (application)
(1) a call arrives at the routing device (***routingCrossRefID*** created and **CSTARouteRequestEvent** is sent)
call related Info is passed (e.g. ANI, DNIS, collected digits)
(1)
(2) Application selects a destination for the call (based on the call and other customer info). A **cstaRouteSelectInv( )** message is sent (**Route Used** = off)
(2)
(3) the switch attempts to route the queued call to selected dest.
(3a) If destination address is o.k. a **CSTARouteEndEvent** is sent
(3b) If destination is invalid, then a **CSTAReRouteEvent** is sent
(3a)
or
(3b)
(4a) If (3a) then routingCrosRefID is no longer valid and route is completed as specified in (2),
(4b) If (3b) then send a second **cstaRouteSelectInv( )** with a different destination (**Route Used** = on).
(4b)
(5) the switch attempts to route the queued call to selected dest.
(5a) If destination address o.k. a **CSTARouteUsedEvent** and **CSTARouteEndEvent** is sent
(5b) If destination is invalid, then a **CSTAReRouteEvent** is sent
(5a)
or
(5b)
(6a) If **(5a)** then ***routingCrosRefID*** is no longer valid and route is completed as specified in **(4b)**,
(6b) If **(5b)** then send a second **cstaRouteSelectInv( )** with a different destination (**Route Used** = on).

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

CSTACSTAROUTEREGISTERREQ( )

An application uses CSTAcstaRouteRegisterReq( ) 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 <CSTAcsta.h>

RetCode_t CSTAcstaRouteRegisterReq (
            ACSHandle_t       acsHandle,
            InvokeID_t        invokeID,
            DeviceID_t        *routingDevice,
            rivateData_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 **CSTAcstaRouteRegisterReq**( ) 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 **CSTAcstaRouteRegisterReq**( ). 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

CSTAcstaRouteRegisterReq( ) 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 **CSTAcstaRouteRegisterReqConfEvent** message to ensure to ensure that the Telephony Server and switch have acknowledged the **CSTAcstaRouteRegisterReq**( ).

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 CSTAcstaRouteRegisterReq( ). 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, **CSTAcstaRouteRequestEvents** 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** 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;
                }  CSTAcstaConfirmation;
        } 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 **CSTAcstaRouteRegisterReq**( ) request.
eventClass	This is a tag with the value **CSTAcstaConfirmation**, 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 **CSTAcstaRouteRegisterReqConfEvent**.
invokeID	This parameter specifies the service request instance for the **CSTAcstaRouteRegisterReq**( ). 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 **CSTAcstaRouteRegisterReqConfEvent** does not deliver private data to the application.

Comments

This event provides the application with a positive confirmation to a request for routing registration.

CSTACSTAROUTEREGISTERCANCEL( )

Applications (routing servers) use CSTAcstaRouteRegisterCancel( ) 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 <CSTAcsta.h>

RetCode_t CSTAcstaRouteRegisterCancel (
            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 **CSTAcstaRouteRegisterCancel**( ) request will be made.
invokeID	This is an application provided handle that the application uses to match a specific instance of a **CSTAcstaRouteRegisterCancel**( ) 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 **CSTAcstaRouteRegisterReqConfEvent**.
privateData	This is an optional pointer to CSTA private data.

Return Values

CSTAcstaRouteRegisterCancel( ) 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 **CSTAcstaRouteRegisterCancelConfEvent** message to ensure that the Telephony Server and switch have acknowledged and processed the **CSTAcstaRouteRegisterCancel**( ) 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 **CSTAcstaRouteRegisterCancel**( ) request for a routing registration. Once tan 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** for a complete description of the event structure.

```
typedef struct
{
        ACSHandle_tacsHandle;
        EventClass_t    eventClass;
        EventType_teventType;
}  ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t eventHeader;
        union
        {
                struct
                {
                        InvokeID_t invokeID;
                        union
                        {
                                CSTARouteRegisterCancelConfEvent_t routeCancel;
                        } u;
                } CSTAcstaConfirmation;
        } 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 CSTAcstaRouteRegisterCancel( ) 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 **CSTAcstaRouteRegisterCancel**( ). 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 **CSTAcstaRouteRegisterCancel**( ) 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** 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;
                } CSTAcstaEventReport;
        } 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. The figure above 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 **CSTAcstaRouteSelectInv**( ). 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 **CSTAReRouteRequestEvent** to request an additional destination. See figure above 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 **CSTAcstaRouteRegisterReq**( ) 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.

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_t     acsHandle;
        EventClass_t eventClass;
        EventType_t     eventType;
} ACSEventHeader_t;

typedef struct {
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        InvokeID_t        invokeID;
                        union
                        {
                                CSTARouteRequestEvent_t         routeRequest;
                        } u;
                } CSTAcstaRequest;
        } event;
} CSTAEvent_t;

typedef struct {
        RouteRegisterReqID_t routeRegisterReqID;
        RoutingCrossRefID_t routingCrossRefID;
        CalledDeviceID_t            currentRoute;
        CallingDeviceID_t           callingDevice;
        ConnectionID_t  routedCall;
        SelectValue_t   routedSelAlgorithm;
        Boolean                     priority;
        SetUpValues_t   setupInformation;
} CSTARouteRequestExtEvent_t;

typedef enum SelectValue_t {
        SV_NORMAL = 0,
        SV_LEAST_COST = 1,
        SV_EMERGENCY = 2,
        SV_ACD = 3,
        SV_USER_DEFINED = 4
} SelectValue_t;

typedef struct SetUpValues_t {
```

```
                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**, 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 **CSTAcstaRouteSelectInv**( ) or **CSTAcstaRouteSelect**( ) 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 an 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 **CSTAcstaRouteSelectInv**( ) or **CSTAcstaRouteSelect**( ) 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;
                } CSTAcstaRequestEvent;
        } 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 is 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 **CSTAcstaRouteSelect**( ) 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 **CSTAcstaRouteEnd**( ) 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.

CSTACSTAROUTESELECTINV( )

The routing server application uses **CSTAcstaRouteSelectInv** 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.

Syntax

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

RetCode_t CSTAcstaRouteSelectInv (
        ACSHandle_t         acsHandle,
        InvokeID_t          invokeID,
        RouteRegisterReqID_t     routeRegisterReqID,
        RoutingCrossRefID_t      routingCrossRefID,
        DeviceID_t          *routeSelected,
        RetryValue_t        remainRetry,
        SetUpValues_t       *setupInformation,
        Boolean             routeUsedReq,
        PrivateData_t       *privateData);
```

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 **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

CSTAcstaRouteSelectInv( ) 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 **CSTAcstaRouteSelectInv** only in response to a **CSTARouteRequestEvent** or **CSTAReRouteRequestEvent**. The **CSTAcstaRouteSelectInv** 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 **CSTAcstaRouteSelect**( ) or **CSTAcstaRouteSelectInv**( ). To receive a **CSTARouteUsed**, the application must set the **CSTAcstaRouteSelect**( ) or **CSTAcstaRouteSelectInv**( ) parameter ***routeUsedReq*** to TRUE.

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;
                        } u;
                } CSTAcstaEventReport;
        } event;
} CSTAEvent_t;

Telephony Services API    8-34
typedef struct
{
        RouteRegisterReqID_t    routeRegisterReqID;
        RoutingCrossRefID_t     routingCrossRefID;
        CalledDeviceID_t                routeUsed;
        CallingDeviceID_t       callingDevice;
        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 **CSTAEVENTREPORT**, which identifies this message as an CSTA unsolicited event.
eventType	This is a tag with the value **CSTA_ROUTE_USED**, 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 **CSTAcstaRouteSelect**( ) or **CSTAcstaRouteSelectInv**( ). 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.

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_t     acsHandle;
        EventClass_t    eventClass;
        EventType_t     eventType;
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t        eventHeader;
        union
        {
                struct
                {
                        union
                        {
                                CSTARouteEndEvent_t  routeEnd,
                        } u;
                } CSTAcstaEventReport;
        } 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.

CSTACSTAROUTEENDINV( )

The routing server (application) uses CSTAcstaRouteEnd( ) to cancel an active routing dialog for a call. The service request includes a cause value giving the reason for the routing dialog termination.

Syntax

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

RetCode_t CSTAcstaRouteEnd (
        ACSHandle_t     acsHandle,
        InvokeID_t      invokeID;
        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

CSTAcstaRouteEndInv( ) 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 CSTAcstaRouteEndInv( ) 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 **CSTAcstaRouteEnd**( ) to inform the switch that it will not route the call in question. **CSTAcstaRouteEnd**( ) will terminate the CSTA routing dialog (**routingCrossRefID**) for the call. **CSTAcstaRouteEnd**( ) 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.