Chapter

4 *Control Services*

TSAPI provides two kinds of control services: API[1] Control ServicesXE "Application Programming Interface Control Services:See ACS" \t " "§XE "API Control Services:See ACS" \t " "§, or ACSXE "ACS"§, and CSTA Control ServicesXE "CSTA:Control Services"§. ApplicationsXE "Applications"§ use ACS to manage their interactions with NetWare Telephony Services. While most applications will use ACS to access CSTA services, applications that administer PBX drivers use ACS to interface to the PBX Driver. ACS functions manage the interface, while CSTA functions (chapters 5 through 9) provide the CSTA services. Applications use ACS to:

- Open an ACS streamXE "ACS stream:Opening"§ for CSTA services
- Open an ACS stream to do PBX Driver administration
- Close an ACS stream
- Block or poll for eventsXE "Events:Blocking for"§XE "Events:Polling for"§
- Initialize an operating system event notification facility. On a Windows, OS/2, Macintosh, or NetWare client, this initializes an Event Service RoutineXE "Event:Service

1 An API is an Application Programming Interface.

Routine (ESR):Also see acsSetESR" \t " "§ (ESR)XE "Event:Service Routine (ESR):Initializing"§

- Get a list of available advertised servicesXE "Advertised services:Getting list of available"§ (PBX Driver Services and PBX Driver administration services)

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

- Query for the CSTA ServicesXE "CSTA:Services:Available on ACS stream"§XE "ACS stream:CSTA services available on"§ available on an open ACS Stream
- Query for a list of DevicesXE "Device:Query:For controllable devices"§ that CSTA Services can monitor, control or route for on an open ACS Stream
- Query to determine if CSTA Call/Call MonitoringXE "Query:Call/Call Monitoring"§ is available on an open ACS Stream.

Opening, Closing and Aborting an ACS Stream

To obtain Telephony Services an application must open an ACS streamXE "ACS stream:Opening"§XE "ACS stream:Closing"§XE "ACS stream:Aborting"§ (or session). This stream establishes a logical linkXE "Logical:Link"§ between the application and call processing software on the switch. The application requests CSTA services (such as making a call) over the stream. Within a Telephony Server, the Telephony Server NLM and the PBX Driver NLM cooperate to provide ACS Streams. The Telephony Server NLM also does security checkingXE "Administration"§ to ensure that an application

receives CSTA services only for permitted Devices. Each application must open an ACS Stream before it requests any services.

An application should only open one stream per advertised serviceXE "ACS stream:Per advertised service"§. An application may open multiple ACS streams to multiple advertised servicesXE "ACS stream:Multiple"§. As PBX drivers initialize, they register the services that they offer (administrative as well as CSTA) with a Telephony Services NLM. The system then advertises these services to applications. An application opens an ACS Stream to use an advertised service. Each stream carries messages for the application to one advertised service. Since the PBX DriversXE "PBX Drivers"§ are switch specificXE "Switch:Specific"§, some drivers may provide services on a single CTI linkXE "CTI:Link"§, while others provide services on multiple CTI links. An application cannot correlate advertised telephony services with underlying physical CTI links.

Opening an ACS StreamXE "ACS stream:Opening"§

1. **The application calls** acsOpenStream( ).**XE "acsOpenStream( )"§**

   acsOpenStream( ) is a request to establish an ACS Stream with a Telephony Server. The **acsOpenStream( )** function returns an ***acsHandle****XE "acsHandle"§* to the application. The application will use this ***acsHandle*** to accessXE "ACS stream:Access"§ the ACS Stream (make requests and receive events).

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

The application monitors the *acsHandle*XE "acsHandle"§ (returned from the **acsOpenStream**( )XE "acsOpenStream( )"§ request) for the corresponding **ACSOpenStreamConfEvent**XE "**ACSOpenStreamConfEvent**"§. The application should not request services on the ACS Stream until it receives this corresponding **ACSOpenStreamConfEvent**.

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

The application should always check the **ACSOpenStreamConfEvent**XE "**ACSOpenStreamConfEvent**"§ to ensure that the ACS StreamXE "ACS stream:Checking establishment of"§ has been successfully established before making any CSTA Service requests.

An application is responsible for releasingXE "ACS stream:Releasing"§ its ACS Stream(s). To release the system resources associated with an ACS Stream the application may either closeXE "ACS stream:Closing"§ the stream or abortXE "ACS stream:Aborting"§ the stream. Failing to release the resources may corrupt the client system, resulting in client failure.

Closing an ACS StreamXE "ACS stream:Closing"§

1. **The application calls acsCloseStream( )XE "acsCloseStream( )"§ to initiate the orderly shutdown of an ACS Stream.**

   After the application calls **acsCloseStream**( )XE "acsCloseStream( )"§ to close an ACS Stream, the application may not request any further services on that Stream. The **acsCloseStream**( ) function is a non-blocking call. The application passes an *acsHandle*XE "acsHandle"§

indicating which ACS Stream to close. Although the application cannot make requests on that Stream, the *acsHandle* remains valid until the application receives the corresponding **ACSCloseStreamConfEvent**XE "ACSCloseStreamConfEvent"§**.**

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

2. The application receives an ACSCloseStreamConfEventXE "ACSCloseStreamConfEvent"§ **event message that corresponds to the** acsCloseStream( )XE "acsCloseStream( )"§ **request.**

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

Aborting an ACS StreamXE "ACS stream:Aborting"§

1. **The application calls acsAbortStream( ).**XE "acsAbortStream( )"§

   An application may use **acsAbortStream**( ) XE "acsAbortStream( )"§ to unilaterally (and synchronously) terminate an ACS Stream when

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

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

Sending CSTA Requests and Responses

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

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

- specify whether it will generate and manage *invokeID*sXE "InvokeID:Application generated"§ internally, or,
- have the TSAPI library generateXE "InvokeID:Library generated"§ unique *invokeID* for each service request.

Once an application specifies this *invokeID* typeXE "InvokeID:Type"§ for an ACS stream, the application cannot change *invokeID* type for the stream.

In general, having the TSAPI library generateXE "InvokeID:Library generated"§ unique *invokeID*s simplifies application design. However, when service requests correspond to entries in a data structure, it may simplify application design to use indexesXE "InvokeID:Application generated"§ into the data structure as *invokeID*sXE "InvokeID:Type"§. Application-generated *invokeIDs* might also point to window handles. Application-generated *invokeIDs* may take on any 32 bit value.

Receiving Events

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

- a blocking modeXE "Events:Blocking for"§, which blocks the application from executing until an event becomes available. Blocking is appropriate in threaded or preemptive operating system environments only (NetWare, UnixWare, OS/2).

- a non-blocking modeXE "Events:Polling for"§ that returns control to the application regardless of whether an event is available.

Blocking on event reports may be appropriate for applications that monitor a Device and only require processing cycles when an event occurs. However, there may be operating system specificXE "Events:Polling for"§ implications. For example, if a Windows application blocks waiting for CSTA events, then it cannot process events from it's Windows event queue.

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

Blocking Event ReceptionXE "Events:Blocking for"§

1. **The application calls acsGetEventBlock( ) XE "acsGetEventBlock( ) "§**

   acsGetEventBlock( )XE "acsGetEventBlock( )"§ function gets the next event or blocks if no events are available. The application passes a ***acsHandle****XE "acsHandle"§* parameter containing the handle of an open ACS Stream or a zero value (indicating that it desires events from any open ACS Stream)XE "Events:From all streams"§.

2. acsGetEventBlock( )**XE "acsGetEventBlock( )"§ returns**

when an event is available.

Non-Blocking Event Reception XE "Events:Polling for"§

1. **The application calls** acsGetEventPoll( ) **XE "acsGetEventPoll( )"§**

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

2. acsGetEventPoll( ) **returns immediately**

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

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

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

In some operating system environments (including the Windows, OS/2, Macintosh, NetWare Client), an application can use an ***Event Service Routine* (ESR)** XE "Event:Service Routine (ESR)"§ to receive asynchronous notification of arriving events.

The ESR mechanism *notifies* the application of arriving events. It does not remove the events from the event queue. The application must use **acsGetEventBlock**( ) or **acsGetEventPoll**( ) to *receive* the message. The application can use an ESR to trigger a specific action when an event arrives in the event queue (i.e. post a Windows™ message for the application, or signal a semaphore in the NetWare® environment). See the manual page for **acsSetESR**( )XE "acsSetESR( )"§ for more information about ESR use in specific operating system environments.

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

Querying for Available Services

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

API Control Services (ACS) Functions and Confirmation Events

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

acsOpenStream ( )

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

Syntax

```
#include <csta.h>

#include <acs.h>

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

Parameters

acsHandle

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

invokeIDType

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

The possible types are: Application-Generated invokeIDs (**APP_GEN_ID**) or Library generated invokeIDs (**LIB_GEN_ID**).

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

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

invokeID

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

streamType

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

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

ST_OAM - requests an OAM stream.

serverID

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

loginID

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

passwd

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

applicationName

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

acsLevelReq

This release of TSAPI ignores this parameter.

apiVer

An application gives the version of TSAPI that it desires in *apiVer*. Future TSAPI versions may provide enhanced services or events that earlier applications will not wish to see for compatibility reasons. This parameter, in the future, will let an application request that TSAPI provide an earlier version of the TSAPI interface. Release 1 does not use this parameter. The **CSTA_API_VERSION** in the *csta.h* header file gives the API version of a Software Development Kit (SDK).

sendQSize

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

sendExtraBufs

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

recvQSize

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

recvExtraBufs

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

privateData

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

Return Values

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

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

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

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

acsOpenStream( ) returns the negative error conditions below:

ACSERR_APIVERDENIED

TSAPI does not provide the version given in

apiVer.

ACSERR_BADPARAMETER
One or more of the parameters is invalid.

ACSERR_DUPSTREAM
An ACS Stream is already established with the advertised service given in *serverID.*

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

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

ACSERR_NORESOURCE
There are insufficient resources to open a ACS Stream.

Comments

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

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

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

Application Notes

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

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

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

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

ACSOpenStreamConfEvent XE "ACSOpenStreamConfEvent"§

This event is generated in response to the **acsOpenStream**( ) function and provides the application with status information regarding the requested open of an ACS Stream with the Telephony Server. The application may only perform the ACS functions **acsEventNotify**( )**, acsSetESR**( ), **acsGetEventBlock**( )**, acsGetEventPoll**( ), and **acsCloseStream**( ) on an *acsHandle* until this confirmation event has been received.

Syntax

The following structure shows only the relevant portions of the unions for this message. See section ***4.3, ACS Data Types*** and ***4.6, CSTA Data Types*** for a complete description of the event structure.

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

typedef struct
{
      ACSEventHeader_t    eventHeader;
      union
      {
            struct
            {
                  InvokeID_t        invokeID;
                  union
                  {
                        ACSOpenStreamConfEvent_t     acsopen;
                  } u;
            }  acsConfirmation;
      } event;} CSTAEvent_t;
typedef struct ACSOpenStreamConfEvent_t
{
      Version_t                  apiVer;
      Version_t                  libVer;
      Version_t                  tsrvVer;
      Version_t                  drvrVer;
} ACSOpenStreamConfEvent_t;
```

Parameters

acsHandle
This is the handle for the ACS Stream.

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

eventType
This is a tag with the value ACS_OPEN_STREAM, which identifies this message as an ACSOpenStreamConfEvent.

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

apiVer
This parameter indicates which version of the API was granted.

libVer
This parameter indicates which version of the Library is running.

tsrvVer
This parameter indicates which version of the TSERVER is running.

drvrVer
This parameter indicates which version of the Driver is running.

Comments

This message is an indication that the ACS Stream requested by the application via the **acsOpenStream**( ) function is available to provide communication with the Telephony Server. The application may now request call control services from the Telephony Server on the acsHandle identifying this ACS

Stream. This message contains the Level of the stream opened, the identification of the server that is providing service, and any Private data returned by the Telephony Server.

Application Notes

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

acsCloseStream( )XE "acsCloseStream ( )"§

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

Syntax

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

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

Parameters

acsHandle

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

invokeID

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

privateData

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

Return Values

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

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

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

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

acsCloseStream( ) returns the negative error conditions below:

ACSERR_BADHDL

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

Comments

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

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

Application Notes

The Client is responsible for receiving the **ACSCloseStreamConfEvent** to free all resources associated with the ACS Stream.

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

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

ACSCloseStreamConfEvent XE "ACSCloseStreamConfEvent"§

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

Syntax

The following structure shows only the relevant portions of the unions for this message. See section ***4.2 ACS Data Types*** and ***4.6 CSTA Data Types*** for a complete description of the event structure.

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


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

Parameters

acsHandle

This is the handle for the opened ACS Stream.

eventClass

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

eventType

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

invokeID

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

Comments

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

ACSUniversalFailureConfEvent XE "ACSUniversalFailureConfEvent"§

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

Syntax

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

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

typedef struct
{
      ACSEventHeader_t    eventHeader;
      union
      {           struct
            {
                  union
                  {
                  ACSUniversalFailureConfEvent_t      failureEvent;
                  } u;
            } acsConfirmation;
      } event;} CSTAEvent_t;
      typedef struct
      {
            int                     failedStatus;
      } ACSUniversalFailureConfEvent_t;
```

Parameters

acsHandle

This is the handle for the ACS Stream.

eventClass

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

eventType

This is a tag with the value ACS_UNIVERSAL_FAILURE_CONF , which identifies this message as an ACSUniversalConfEvent.

failedStatus

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

Comments

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

acsAbortStream( )XE "acsAbortStream( )"§

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

Syntax

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

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

Parameters

acsHandle

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

privateData

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

Return Values

This function always returns zero (0) if successful.

The following are possible negative error conditions for this

function:

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

Comments

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

acsGetEventBlock()XE "acsGetEventBlock()"§

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

Syntax

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

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

Parameters

acsHandle

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

eventBuf

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

eventBufSize

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

privateData

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

numEvents

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

Return Values

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

ACSPOSITIVE_ACK

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

Possible local error returns are (negative returns):

ACSERR_BADHDL

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

handle is passed with this function.

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

Comments

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

There is no confirmation event for this function.

Application Notes

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

acsGetEventPoll()XE "acsGetEventPoll( )"§

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

Syntax

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

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

Parameters

acsHandle

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

eventBuf

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

eventBufSize

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

privateData

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

numEvents

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

Return Values

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

ACSPOSITIVE_ACK

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

Possible local error returns are (negative returns):

ACSERR_BADHDL

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

handle is passed with this function.

ACSERR_NOMESSAGE
The function were no messages available to return to the application.

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

Comments

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

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

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

There is no confirmation event for this function.

Application Notes

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

acsGetFile( ) (UnixWare)XE "acsGetFile( ) (UnixWare)"§

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

Syntax

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

RetCode_t acsGetFile (ACSHandle_t acsHandle);
```

Parameters

acsHandle

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

Return Values

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

Application Notes

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

There is no confirmation event for this function.

acsSetESR( ) (Windows)XE "acsSetESR( ):Windows"§

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

Syntax

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

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

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

Parameters

acsHandle

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

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

esrParam

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

notifyAll

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

Return Values

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

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

Possible local error returns are (negative returns):

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

Comments

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

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

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

There is no confirmation event for this function.

Application Notes

The application can use the ESR mechanism to trigger platform specific events (e.g. post a Windows™ message for the application, or signal a semaphore in the NetWare® environment).

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

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

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

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

Windows Client Note:

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

acsSetESR( ) (Macintosh)xe "acsSetESR( ):Macintosh"§

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

Syntax

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

typedef pascal void (*EsrFunc)(unsigned long esrParam)

enum {
      uppESRFuncProcInfo = kPascalStackBased
            | RESULT_SIZE(SIZE_CODE(0))
            | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(long))
};

#if USESROUTINEDESCRIPTORS
typedef UniversalProcPtr EsrFuncUPP;

#define NewEsrFuncProc(userRoutine)
      (EsrFuncUPP) NewRoutineDescriptor((ProcPtr)(userRoutine),
            uppEsrFuncProcInfo, GetCurrentISA())
#else
typedef EsrFunc       EsrFuncUPP;
#define NewEsrFuncProc(userRoutine)
      (EsrFuncUPP)(userRoutine)
#endif

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

Parameters

acsHandle

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

esr

This is a universal procedure pointer to the ESR (the address of a 680x0 function or routine descriptor). An application passes a NULL pointer indicates to clear an existing ESR**.**

esrParam

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

notifyAll

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

Return Values

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

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

Possible local error returns are (negative returns):

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

Comments

***The ESR mechanism can be used by the application to receive an asynchronous notification of the arrival of an incoming event from the Open ACS Stream. The ESR routine will receive one user-defined parameter. The ESR should* not** call ACS functions, otherwise the results will be indeterminate. The ESR should note the arrival of the incoming event, and complete its

operation as quickly as possible. The application must still call **acsGetEventBlock** or **acsGetEventPoll**( ) to retrieve the event from the Client API Library queue.

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

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

There is no confirmation event for this function.

Application Notes

The application may use the ESR mechanism for asynchronous notification of the arrival of incoming events, particularly when rapid notification is desired. By using the ESR to set an application global, the application may determine whether events have arrived by examining that global rather than using acsGetEventPoll( ) or **acsGetEventBlock**( ).

The ESR function is defined as a universal procedure pointer. Under PPC, providing a native or fat routine descriptor will result in the best performance as there will be no mode switch involved when calling the ESR.

The application may not call ACS functions from within the ESR**.**

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

The ESR function *may* be called while (some level of) interrupts are disabled; refer to Inside Macintosh for information about programming with interrupts disabled. Ensure that the ESR function — and routine descriptor under PPC — remain loaded and page-locked in memory. In particular, do not make synchronous I/O calls or access memory that is not page-locked.

On Macintosh — as with other interrupt service routines — the ESR is prohibited from using the Macintosh memory manager — directly or indirectly. In addition, the ESR must set any global context it needs. On the 680x0 Macintosh, this means that the ESR must set A5 before accessing application globals or making inter-segment jumps; before returning, the ESR *must* restore A5 to its value on entry. On PowerPC, the runtime model automatically manages this context. See references [3]and [5] for more information.

acsSetESR( ) (OS/2 2.1)**XE "acsSetESR( )"§**

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

Syntax

```
#include <os2,h>
#include <csta.h>
#include <acs.h>

typedef void (*EsrFunc)(ULONG esrParam)

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

Parameters

acsHandle

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

This is a pointer to the ESR (the address of a function). This function must use the _Optlink calling convention. A multi-threaded application that registers the same ESR for multiple open streams needs to ensure that this function is reentrant. An application passes a NULL pointer indicates to clear an existing ESR.**.**

esrParam

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

notifyAll

If this parameter is **TRUE** then the ESR will be called for every event**.** If it is **FALSE** then the ESR will only be called each time the receive queue becomes non-empty, i.e. the queue count

changes from zero (0) to one (1). This option may be used to reduce the overhead of notification.

Return Values

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

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

Possible local error returns are (negative returns):

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

Comments

The ESR mechanism can be used by the application to receive an asynchronous notification of the arrival of an incoming event from the Open ACS Stream. The application can use the ESR mechanism to trigger specific events (e.g. post an event semaphore). The ESR routine will receive one user-defined parameter. The ESR should **not** call ACS functions, otherwise the results will be indeterminate. The application must still call **acsGetEventBlock()** or **acsGetEventPoll( )** to actually retrieve the event from the Client API Library queue.

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

The ***esr*** in the **acsSetESR( )** function will replace the current

ESR maintained by the API Client Library. A NULL ***esr*** will disable the current ESR mechanism.

There is no confirmation event for this function.

acsEventNotify( ) (Windows 3.1) XE "acsEventNotify( ):Windows 3.1"§

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

Syntax

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

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

Parameters

acsHandle

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

hwnd

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

msg

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

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

notifyAll

If this parameter is **TRUE** then a message will be posted for every event**.** If it is **FALSE** then a message will only be posted

each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification, or the likelihood of overflowing the application's message queue (see below).

Return Values

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

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

Possible local error returns are (negative returns):

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

Application Notes

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

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

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

There is no confirmation event for this function.

Example

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

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

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

      switch (msg)
      {
      case WM_CREATE:

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

            acsEventNotify (acsHandle, hwnd, WM_ACSEVENT,
      TRUE);

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

      case WM_ACSEVENT:

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

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

            return 0;

      // process other window messages...

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

acsEventNotify( ) (Macintosh) xe "acsEventNotify( ):Macintosh"§

The **acsEventNotify**( ) function allows a Macintosh application to request that it receive an Apple Event when an incoming ACS event is available.

Syntax

```
#include <csta.h>
#include <acs.h>
#include <EPPC.h>       /* for Apple Event types */

RetCode_t acsEventNotify (ACSHandle_t   acsHandle,
                          AEAddressDesc *targetAddr,
                          Boolean       notifyAll );
```

Parameters

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

targetAddr
This is a pointer to an AEAddressDesc data structure. The event notification Apple Events will be sent to address specified by the AEAddressDesc. A NULL **targetAddr** indicates no notification.

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

Return Values

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

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

Possible local error returns are (negative returns):

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

Application Notes

The Apple Events posted as the result of calling **acsEventNotify**( ) have the following attributes:

Event Class	kTSAPIEventClass
Event ID	kTSAPIEventArrived
Required Parameter	
Keyword:	keyTSAPIEventClass
Descriptor Type:	typeShortInteger
Data:	The EventClass_t corresponding to the incoming TSAPI

event.

Required Parameter

Keyword:	keyTSAPIEventType
Descriptor Type:	typeShortInteger
Data:	The EventType_t corresponding to the incoming TSAPI event.

Required Parameter

Keyword:	keyStreamHandle
Descriptor Type:	typeLongInteger
Data:	The ACSHandle_t that may be used to retrieve the incoming TSAPI event.

See reference [4] for information on how to create an AEAddressDesc and extract information from the notification Apple Events.

After calling **acsEventNotify**( ), properly dispose of the AEAddressDesc specified by **targetAddr**.

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

If there are already events in the receive queue waiting to be retrieved when **acsEventNotify**( ) is called, an Apple Event will be sent for each of them.

Applications which process a high volume of incoming events should either set ***notifyAll*** to **TRUE** or use **acsSetESR**( ); the current theoretical upper bound on sending Apple Events is sixty messages per second. In practice — depending on processor speed and available memory — this number may be significantly lower.

There is no confirmation event for this function.

Example

This example uses the acsEventNotify function to enable event notification.

```
/*
 * handleTSAPIEvent - install as AppleEvent handler (callback)
 *    before using acsEventNotify()
 */

pascal OSErr
handleTSAPIEvent(                   const AppleEvent        *theAppleEvent,
                                    const AppleEvent        *reply,
                                    long                    handlerRefcon)
{
        EventClass_t                theTSAPIClass;
        EventType_t                 theTSAPIType;
        ACSHandle_t                 theStream;
        DescType                    actualType;             /* scratch */
        Size                        actualSize;             /* scratch */
        OSErr                       myErr;
        /*
         * other local variables
         */

        /* extract TSAPI event class */

        myErr = AEGetParamPtr ( theAppleEvent, keyTSAPIEventClass,
                                    typeShortInteger, &actualType,
                                    &theTSAPIClass,
                                    sizeof(theTSAPIClass),
                                    &actualSize );
        if ( myErr != noErr )
                return myErr;

        /* extract TSAPI event type */

        myErr = AEGetParamPtr ( theAppleEvent, keyTSAPIEventType,
                                    typeShortInteger, &actualType,
                                    &theTSAPIType, sizeof(theTSAPIType),
                                    &actualSize );
        if ( myErr != noErr )
                return myErr;

        /* extract stream handle */

        myErr = AEGetParamPtr ( theAppleEvent, keyStreamHandle,
                                    typeLongInteger, &actualType,
                                    &theStream, sizeof(theStream),
                                    &actualSize );
        if ( myErr != noErr )
                return myErr;
        /*
         * Dispatch event to user-defined handler function here
         */
```

```
    return noErr;
}
```

```
/* example - cont. */

OSErr
InstallTSAPIEventHandler ( void )
{
        /*
         * This code only works when compiled for 68K; it needs a
         * routine descriptor for handleTSAPIEvent to work with the
         * Mixed Mode manager.
         */

        return AEInstallEventHandler (
                                        kTSAPIEventClass,
                                        kTSAPIEventArrived
                                        (AEEventHandlerUPP)handleTSAPIEvent,
                                        0,
                                        FALSE );
}
```

acsEventNotify( ) (OS/2 2.1) **XE "acsEventNotify( ) (Windows 3.1) "§**

The acsEventNotify( ) function allows an OS/2 PM application to request that a message be posted to its application queue when an incoming ACS event is available.

Syntax

```
#include <os2.h>
#include <csta.h>
#include <acs.h>

RetCode_t               acsEventNotify (
    ACSHandle_t         acsHandle,
    HWND                hwnd,
    ULONG               msg,
    BOOL                notifyAll);
```

Parameters

acsHandle

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

hwnd

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

msg

This is the user-defined message to be posted when an incoming event becomes available. The *mp1* and *mp2* parameters of the message will contain the following members of the ACSEventHeader_t structure:

mp1	acsHandle
SHORT2FROMMP(mp2)	eventClass
SHORT1FROMMP(mp2)	eventType

notifyAll

If this parameter is **TRUE** then a message will be posted for every event**.** If it is **FALSE** then a message will only be posted

each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification, or the likelihood of overflowing the application's message queue (see below).

Return Values

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

ACSPOSITIVE_ACK
The function completed successfully as requested by the application. No errors were detected.
Possible local error returns are (negative returns):

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

Application Notes

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

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

Applications which process a high volume of incoming events may cause the default application queue (10 messages max) to overflow. In this case, increase the size of the application queue that is created by specifying a larger size in the WinCreateMsgQueue() function. Also, the rate of notifications

may be reduced by setting ***notifyAll*** to **FALSE**.
There is no confirmation event for this function.

Example

This example uses the acsEventNotify function to enable event notification.

```
#define WM_ACSEVENT WM_USER + 99

MRESULT EXPENTRY
WndProc (HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2)
{
    // declare local variables...

    switch (msg)
    {
    case WM_CREATE:

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

        acsEventNotify (acsHandle, hwnd, WM_ACSEVENT, TRUE);

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

    case WM_ACSEVENT:

        // mp1 contains an ACSHandle_t
        // SHORT2FROMMP(mp2) contains an EventClass_t
        // SHORT1FROMMP(mp2) contains an EventType_t

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

        return 0;

    // process other window messages...

    }
    return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

acsFlushEventQueue( )XE "acsFlushEventQueue( )"§

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

Syntax

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

RetCode_t ACSFlushEventQueue (ACSHandle_t acsHandle);
```

Parameters

acsHandle

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

Return Values

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

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

Possible local error returns are (negative returns):

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

Comments

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

There is no confirmation event for this function.

Application Notes

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

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

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

acsEnumServerNames( )XE "acsEnumServerNames( )"§

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

Syntax

```
#include <acs.h>

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

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

Parameters

streamType

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

callback

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

lParam

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

Return Values

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

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

The following are possible negative error conditions for this function:

ACSERR_UNKNOWN
The request has failed due to unknown network problems.

Comments

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

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

ACS Unsolicited EventsXE "ACS:Unsolicited Events"§XE "Unsolicited Events"§

This section describes unsolicited ACS Status Events.

ACSUniversalFailureEvent XE "ACSUniversalFailureEvent"§

This event can occur at any time (unsolicited) and can indicate, among other things, a failure or lost of the ACS Stream with the Telephony Server.

Syntax

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

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

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

Parameters

acsHandle

This is the handle for the ACS Stream.

eventClass

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

eventType

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

error

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

Not all of the errors listed below will occur in a ACS Universal Failure message. Some of the errors occur only in error logs generated by the Tserver.

The possible valuesXE "ACSUniversalFailureEvent:Possible values"§ are:

```
typedef enum ACSUniversalFailure_t {
    TSERVER_STREAM_FAILED = 0,
    TSERVER_NO_THREAD = 1,
    TSERVER_BAD_DRIVER_ID = 2,
    TSERVER_DEAD_DRIVER = 3,
    TSERVER_MESSAGE_HIGH_WATER_MARK = 4,
    TSERVER_FREE_BUFFER_FAILED = 5,
    TSERVER_SEND_TO_DRIVER = 6,
    TSERVER_RECEIVE_FROM_DRIVER = 7,
    TSERVER_REGISTRATION_FAILED = 8,
    TSERVER_SPX_FAILED = 9,
    TSERVER_TRACE = 10,
    TSERVER_NO_MEMORY = 11,
    TSERVER_ENCODE_FAILED = 12,
    TSERVER_DECODE_FAILED = 13,
    TSERVER_BAD_CONNECTION = 14,
    TSERVER_BAD_PDU = 15,
    TSERVER_NO_VERSION = 16,
    TSERVER_ECB_MAX_EXCEEDED = 17,
    TSERVER_NO_ECBS = 18,
    TSERVER_NO_SDB = 19,
    TSERVER_NO_SDB_CHECK_NEEDED = 20,
    TSERVER_SDB_CHECK_NEEDED = 21,
    TSERVER_BAD_SDB_LEVEL = 22,
    TSERVER_BAD_SERVERID = 23,
    TSERVER_BAD_STREAM_TYPE = 24,
    TSERVER_BAD_PASSWORD_OR_LOGIN = 25,
    TSERVER_NO_USER_RECORD = 26,
    TSERVER_NO_DEVICE_RECORD = 27,
    TSERVER_DEVICE_NOT_ON_LIST = 28,
    TSERVER_USERS_RESTRICTED_HOME = 30,
    TSERVER_NO_AWAYPERMISSION = 31,
    TSERVER_NO_HOMEPERMISSION = 32,
```

```
    TSERVER_NO_AWAY_WORKTOP = 33,
    TSERVER_BAD_DEVICE_RECORD = 34,
    TSERVER_DEVICE_NOT_SUPPORTED = 35,
    TSERVER_INSUFFICIENT_PERMISSION = 36,
    TSERVER_NO_RESOURCE_TAG = 37,
    TSERVER_INVALID_MESSAGE = 38,
    TSERVER_EXCEPTION_LIST = 39,
    TSERVER_NOT_ON_OAM_LIST = 40,
    TSERVER_PBX_ID_NOT_IN_SDB = 41,
    TSERVER_USER_LICENSES_EXCEEDED = 42,
    TSERVER_OAM_DROP_CONNECTION = 43,
    TSERVER_NO_VERSION_RECORD = 44,
     TSERVER_OLD_VERSION_RECORD = 45,
    TSERVER_BAD_PACKET = 46,
    TSERVER_OPEN_FAILED = 47,
    TSERVER_OAM_IN_USE = 48,
    TSERVER_DEVICE_NOT_ON_HOME_LIST = 49,
    TSERVER_DEVICE_NOT_ON_CALL_CONTROL_LIST = 50,
    TSERVER_DEVICE_NOT_ON_AWAY_LIST = 51,
    TSERVER_DEVICE_NOT_ON_ROUTE_LIST = 52,
    TSERVER_DEVICE_NOT_ON_MONITOR_DEVICE_LIST = 53,
    TSERVER_DEVICE_NOT_ON_MONITOR_CALL_DEVICE_LIST = 54,
    TSERVER_NO_CALL_CALL_MONITOR_PERMISSION = 55,
    TSERVER_HOME_DEVICE_LIST_EMPTY = 56,
    TSERVER_CALL_CONTROL_LIST_EMPTY = 57,
    TSERVER_AWAY_LIST_EMPTY = 58,
    TSERVER_ROUTE_LIST_EMPTY = 59,
    TSERVER_MONITOR_DEVICE_LIST_EMPTY = 60,
    TSERVER_MONITOR_CALL_DEVICE_LIST_EMPTY = 61,
    TSERVER_USER_AT_HOME_WORKTOP = 62,
    TSERVER_DEVICE_LIST_EMPTY = 63,
    TSERVER_BAD_GET_DEVICE_LEVEL = 64,
    TSERVER_DRIVER_UNREGISTERED = 65,
    TSERVER_NO_ACS_STREAM = 66,
    TSERVER_DROP_OAM = 67,
    TSERVER_ECB_TIMEOUT = 68,
    TSERVER_BAD_ECB = 69,
    TSERVER_ADVERTISE_FAILED = 70,
    TSERVER_NETWARE_FAILURE = 71,
    TSERVER_TDI_QUEUE_FAULT = 72,
    TSERVER_DRIVER_CONGESTION = 73,
    TSERVER_NO_TDI_BUFFERS = 74,
    TSERVER_OLD_INVOKEID = 75,
    TSERVER_HWMARK_TO_LARGE = 76,
    TSERVER_SET_ECB_TO_LOW = 77,
    TSERVER_NO_RECORD_IN_FILE = 78,
    TSERVER_DRIVER_CONGESTION = 73,
    DRIVER_DUPLICATE_ACSHANDLE = 1000,
    DRIVER_INVALID_ACS_REQUEST = 1001,
    DRIVER_ACS_HANDLE_REJECTION = 1002,
    DRIVER_INVALID_CLASS_REJECTION = 1003,
    DRIVER_GENERIC_REJECTION = 1004,
    DRIVER_RESOURCE_LIMITATION = 1005,
    DRIVER_ACSHANDLE_TERMINATION = 1006,
    DRIVER_LINK_UNAVAILABLE = 1007
} ACSUniversalFailure_t;
```

Tserver Operation errorsXE "ACSUniversalFailureEvent:Tserver operation errors"§

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

TSERVER_STREAM_FAILED
XE "Tserver Errors:Stream Failed"§The Client Library detected that the ACS Stream failed.

TSERVER_NO_THREAD
XE "Tserver Errors:No Thread"§One or more the threads (processes) that make up the Tserver could not be created.

TSERVER_BAD_DRIVER_ID
XE "Tserver Errors:Bad Driver ID"§One of the threads (processes) that make up the Tserver encountered a bad Driver Identification number during processing.

TSERVER_DEAD_DRIVER
XE "Tserver Errors:Dead Driver"§A Driver has not sent a heart beat messages to the Tserver form the last three minutes. The Driver may be in an inoperable state.

TSERVER_MESSAGE_HIGH_WATER_MARK
XE "Tserver Errors:Message High Water Mark"§The message rate between a client and the Tserver or the Tserver and a Driver has exceeded the high water mark rate.

TSERVER_FREE_BUFFER_FAILED
XE "Tserver Errors:Free Buffer Failed"§The Tserver was unable to free Tserver Driver

Interface (TDI) memory.

TSERVER_SEND_TO_DRIVER
XE "Tserver Errors:Send To Driver"§The Tserver was unable to send a message to a Driver.

TSERVER_RECEIVE_FROM_DRIVER
XE "Tserver Errors:Receive From Driver"§The Tserver was unable to receive a message from a Driver.

TSERVER_REGISTRATION_FAILED
XE "Tserver Errors:Registration Failed"§A Driver's attempt to register with the Tserver failed.

TSERVER_SPX_FAILED
XE "Tserver Errors:Spx Failed"§ A NetWare SPX call failed in the Tserver.

TSERVER_TRACE
XE "Tserver Errors:Trace"§Used by the Tserver for debugging purposes only.

TSERVER_NO_MEMORY
XE "Tserver Errors:No Memory"§The Tserver was unable to allocate a piece of memory.

TSERVER_ENCODE_FAILED
XE "Tserver Errors:Encode Failed"§The Tserver was unable to encode a message for shipment to a client workstation.

TSERVER_DECODE_FAILED
XE "Tserver Errors:Decode Failed"§ The Tserver was unable to decode a message from a client workstation.

TSERVER_BAD_CONNECTION
XE "Tserver Errors:Bad Connection"§The Tserver tried to process a request with a bad client connection ID number.

TSERVER_BAD_PDU
XE "Tserver Errors:Bad PDU"§The Tservers internal table of Protocol Descriptor Units is corrupted.

TSERVER_NO_VERSION
XE "Tserver Errors:No Version"§The Tserver processed a ACSOpenStreamConfEvent from a Driver in which one or more the version fields was not set.

TSERVER_ECB_MAX_EXCEEDED
XE "Tserver Errors:ECB Max Exceeded"§The Tserver can not process a message from the driver because the message is larger than the sum of the ECBs allocated for this driver.

TSERVER_NO_ECBS
XE "Tserver Errors:No ECBS"§The Tserver has no available ECBs to send events to the client.

TSERVER_NO_RESOURCE_TAG
XE "Tserver Errors:No Resource Tag"§The Tserver was unable to get a resource tag for the purpose of allocating memory.

TSERVER_INVALID_MESSAGE
XE "Error:Tserver:See TServer Errors"§XE "Tserver Errors:Invalid Message"§The Tserver received an invalid Tserver OAM message.

Tserver Security Data Base errorsXE "ACSUniversalFailureEvent:Security database errors"§

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

TSERVER_NO_SDB
XE "Error:Tserver No SDB"§XE "Tserver Errors:No SDB"§One or more the files that makeup the Security Data Base is not present on the server or can not be opened.

TSERVER_NO_SDB_CHECK_NEEDED
XE "Error:Tserver No SDB Check Needed"§XE "Tserver Errors:No SDB Check Needed"§The requested service event does not require a Security Data Base check.

TSERVER_SDB_CHECK_NEEDED
XE "Error:Tserver SDB Check Needed"§XE "Tserver Errors:SDB Check Needed"§The requested service event does require a Security Data Base check.

TSERVER_BAD_SDB_LEVEL
XE "Error:Tserver Bad SDB Level"§XE "Tserver Errors:Bad SDB Level"§The Tservers internal table of API calls indicating which level of security to perform on the request is corrupted.

TSERVER_BAD_SERVERID
XE "Error:Tserver Bad Server ID"§XE "Tserver Errors:Bad Server ID"§The Tserver rejected an ACSOpenStream request because the Server ID in the message did not match a Driver

supported by this Tserver.

TSERVER_BAD_STREAM_TYPE
XE "Error:Tserver Bad Stream Type"§XE "Tserver Errors:Bad Stream Type"§The stream type an ACSOpenStream request was invalid.

TSERVER_BAD_PASSWORD_OR_LOGIN
XE "Error:Tserver Bad Password Or Login"§XE "Tserver Errors:Bad Password Or Login"§The Password or Login or both from an ACSOpenStream request did not match an entry in the Bindery on the server the Tserver is running on.

TSERVER_NO_USER_RECORD
XE "Error:Tserver No User Record"§XE "Tserver Errors:No User Record"§No user record was found in the Security Data Base for the login specified in the ACSOpenStream request.

TSERVER_NO_DEVICE_RECORD
XE "Error:Tserver No Device Record"§XE "Tserver Errors:No Device Record"§No device record was found in the Security Data Base for the device specified in the API call.

TSERVER_DEVICE_NOT_ON_LIST
XE "Error:Tserver Device Not On List"§XE "Tserver Errors:Device Not On List"§The specified device in an API call was not found on any device list administered for this user.

TSERVER_USERS_RESTRICTED_HOME
XE "Error:Tserver Users Restricted Home"§XE "Tserver Errors:Users Restricted

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

TSERVER_NO_AWAYPERMISSION
XE "Error:Tserver No Away Permission"§XE "Tserver Errors:No Away Permission"§The Tserver rejected a service request because the device did not match a device associated with an away worktop.

TSERVER_NO_HOMEPERMISSION
XE "Error:Tserver No Home Permisssion"§XE "Tserver Errors:No Home Permisssion"§The Tserver rejected a service request because the device did not match a device associated with a home worktop.

TSERVER_NO_AWAY_WORKTOP
XE "Error:Tserver No Away Worktop"§XE "Tserver Errors:No Away Worktop"§The away worktop the user is working from is not administered in the Security Data Base.

TSERVER_BAD_DEVICE_RECORD
XE "Error:Tserver Bad Device Record"§XE "Tserver Errors:Bad Device Record"§The Tserver read a device record from the Security Data Base that contained corrupted information.

TSERVER_DEVICE_NOT_SUPPORTED
XE "Error:Tserver Device Not Supported"§XE "Tserver Errors:Device Not Supported"§The device in the API call is administered to be supported by a different Tserver.

TSERVER_INSUFFICIENT_PERMISSION
XE "Error:Tserver Insufficient Permission"§XE "Tserver Errors:Insufficient Permission"§The device in the API call is at the users away worktop and the device has a higher permission level than the user, preventing the user from controlling the device.

TSERVER_EXCEPTION_LIST
XE "Error:Tserver Exception List"§XE "Tserver Errors:Exception List"§The device in the API call is on an exception list which is administered as part of the information for this user.

Driver ErrorsXE "ACSUniversalFailureEvent:Driver errors"§

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

DRIVER_DUPLICATE_ACSHANDLE
XE "Error:Driver:See Driver Errors"§XE "Driver errors:Duplicate ACSHandle"§The acsHandle given for an ACSOpenStream request is already in use for a session. The already open session with the acsHandle is remains open.

DRIVER_INVALID_ACS_REQUEST
XE "Driver errors:Invalid ACS Request"§The ACS message contains an invalid or unknown request. The request is rejected.

DRIVER_ACS_HANDLE_REJECTION
XE "Driver ACS Handle Rejection"§XE "Driver errors:ACS Handle Rejection"§A CSTA request was issued with no prior

ACSOpenStream request. The request is rejected.

DRIVER_INVALID_CLASS_REJECTION
XE "**Driver** errors:**Invalid Class Rejection**"§The driver received a message containing an invalid or unknown message class. The request is rejected.

DRIVER_GENERIC_REJECTION
XE "**Driver** errors:**Generic Rejection**"§The driver detected an invalid message for something other than message type or message class. This is an internal error and should be reported.

DRIVER_RESOURCE_LIMITATION
XE "**Driver** errors:**Resource Limitation**"§The driver did not have adequate resources (i.e. memory, etc.) to complete the requested operation. This is an internal error and should be reported.

DRIVER_ACSHANDLE_TERMINATION
XE "**Driver ACSHandle Termination**"§XE "**Driver** errors:**ACSHandle Termination**"§Due to problems with the link to the switch the driver has found it necessary to terminate the session with the given acsHandle. The session will be closed, and all outstanding requests will terminate.

DRIVER_LINK_UNAVAILABLE
XE "**Driver** errors:**Link Unavailable**"§The driver was unable to open the new session because no link was available to the PBX. The link may have been placed in the BLOCKED state, or it may have been taken off-line.

ACS Data TypesXE "ACS Data Types"§XE "Data Types:ACS"§

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

ACS Common Data TypesXE "ACS Data Types:Common"§

This section specifies the common ACS data types.

```
typedef int RetCode_t;

#define ACSPOSITIVE_ACK 0 /* Successful function return */

/* Error Codes */

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

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

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

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

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

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

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

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

#define ACSERR_UNKNOWN                  -9   /* The ACS Stream has
                                             * encounteredan
                                             * unspecified error.
                                             */

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

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

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

#define ACSERR_QUEUE_FULL               -13  /* The send queue is
                                                * full. No message
                                                *has been sent.
                                                */

typedef unsigned long                   InvokeID_t;

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

```
typedef unsigned short ACSHandle_t;

typedef unsigned short                  EventClass_t;

// defines for ACS event classes

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

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

typedef char Boolean;
typedef char Nulltype;

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

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

typedef char ServerID_t[49];

typedef char LoginID_t[49];

typedef char Passwd_t[49];

typedef char AppName_t[21];

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

typedef char Version_t[21];
```

ACS Event Data Types

This section specifies the ACS data types used in the construction of generic *ACSEvent_t* structures (see section 4.6).

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

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

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

CSTA Control Services and Confirmation Events

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

cstaGetAPICaps( )XE " cstaGetAPICaps( )"§

cstaGetAPICaps( ) obtains the CSTA API function and event capabilities which are supported by the Telephony Servers on the system. The servers could be a local client Telephony Server or a remote Telephony Server across a network or internetwork. If a capability is supported then any corresponding confirmation event is also supported.

Syntax

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

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

Parameters

acsHandle

This is the handle to an active ACS Stream.

invokeID

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

Return Values

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

Library-generated Identifiers - if the function call completes successfully it will return a positive

value, i.e. the invoke identifier. If the call fails a negative error (<0) condition will be returned. For library-generated identifiers the return will never be zero (0).

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

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

The following are possible negative error conditions for this function:

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

Comments

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

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

CSTAGetAPICapsConfEventXE "CSTAGetAPICapsConfEvent"§

This event is in response to the cstaGetAPICaps( ) function and it provides an indication of whether the requested function or event is supported by a specific Telephony Server.

Syntax

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

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

typedef struct
{
      ACSEventHeader_t    eventHeader;
      union
            {           struct
            {                 InvokeID_t       invokeID;          union
                  {
                        CSTAGetAPICapsConfEvent_t getAPIcaps;
                  } u;
            } cstaConfirmation;
      } event;} CSTAEvent_t;
typedef struct CSTAGetAPICapsConfEvent_t {
   short       alternateCall;
   short       answerCall;
   short       callCompletion;
   short       clearCall;
   short       clearConnection;
   short       conferenceCall;
   short       consultationCall;
   short       deflectCall;
   short       pickupCall;
   short       groupPickupCall;
   short       holdCall;
   short       makeCall;
   short       makePredictiveCall;
   short       queryMwi;
   short       queryDnd;
   short       queryFwd;
   short       queryAgentState;
   short       queryLastNumber;
   short       queryDeviceInfo;
   short       reconnectCall;
```

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

```
   short        sysStatReqConf;
   short        sysStatEvent;
} CSTAGetAPICapsConfEvent_t;
```

Parameters

acsHandle

This is the handle for the ACS Stream.

eventClass

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

eventType

This is a tag with the value **CSTA_GETAPI_CAPS_CONF** , which identifies this message as an **CSTAGetAPICapsConfEvent.**

getAPIcaps

This structure contains a integer for each possible CSTA capability which indicates whether the capability is supported. A value of 0 indicates the capability is not supported, a positive value indicates the version of the API (this version is distinct from the version of the API requested in the ACSopen) call that is supported.

For this release of the API, all API calls are on version 1.

Comments

This event will provide the application with compatibility information for a specific Telephony Server on a command/event basis. All the commands and events supported by a Telephony Server must be supported as defined in this document.

cstaGetDeviceList( )XE " cstaGetDeviceList( )"§

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

Syntax

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

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

Parameters

acsHandle

This is the handle to an active ACS Stream.

invokeID

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

index

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

CSTAGetDeviceListConfEvent which contains the last batch of devices.

level

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

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

To determine if an ACS stream has permission to do call/call monitoring, use the API call **CSTAQueryCallMonitor.**

Return Values

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

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

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

The application should always check the

CSTAGetDeviceListConfEvent message to ensure that the service request has been acknowledged and processed by the Telephony Server and the switch.

The following are possible negative error conditions for this function:

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

CSTAGetDeviceListConfEventXE "CSTAGetDeviceListConfEvent"§

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

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
                  {
                    CSTAGetDeviceListConfEvent_t getDeviceList;
                  } event;
            } cstaConfirmation;
      } u;} CSTAEvent_t;
typedef enum SDBLevel_t {
   NO_SDB_CHECKING = 1,
   ACS_ONLY = 2,
   ACS_AND_CSTA_CHECKING = 3
} SDBLevel_t;

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

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

```
} DeviceList;

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

Parameters

acsHandle

This is the handle for the ACS Stream.

eventClass

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

eventType

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

invokeID

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

driverSdbLevel

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

NO_SDB_CHECKING	Not Used.
ACS_ONLY	Check ACSOpenStream requests only
ACS_AND_CSTA_CHECKING	Check ACSOpenStream and all

index

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

devlist

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

cstaQueryCallMonitor( )XE "cstaQueryCallMonitor( )"§

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

Syntax

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

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

Parameters

acsHandle

This is the handle to an active ACS Stream.

invokeID

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

Return Values

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

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

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

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

The following are possible negative error conditions for this function:

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

CSTAQueryCallMonitorConfEventXE " CSTAQueryCallMonitorConfEvent"§

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

Syntax

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

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

typedef struct
{
        ACSEventHeader_t    eventHeader;
        union
        {               struct
                {               InvokeID_t      invokeID;               union
                        {
                         CSTAQueryCallMonitorConfEvent_t  queryCallMonitor;
                        } event;
                } cstaConfirmation;
        } u;} CSTAEvent_t;
typedef struct CSTAQueryCallMonitorConfEvent_t {
   Boolean        callMonitor;
} CSTAQueryCallMonitorConfEvent_t;
```

Parameters

acsHandle

This is the handle for the ACS Stream.

eventClass

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

eventType

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

invokeID

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

callMonitor

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

CSTA Event Data TypesXE "CSTA:Event Data Types"§

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

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

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

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

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

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

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

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

typedef struct {
        union
        {
                CSTARouteRegisterAbortEvent_t   registerAbort;
                CSTARouteUsedEvent_t                routeUsed;
                CSTARouteEndEvent_t                 routeEnd;
                CSTAPrivateEvent_t              privateEvent;
                CSTASysStatEvent_t              sysStat;
                CSTASysStatEndedEvent_t         sysStatEnded;
        }u;
} CSTAEventReport;
```

```
typedef struct {
      CSTAMonitorCrossRefID_t       monitorCrossRefId;
      union
      {
            CSTACallClearedEvent_t                callCleared;
            CSTAConferencedEvent_t                conferenced;
            CSTAConnectionClearedEvent_t    connectionCleared;
            CSTADeliveredEvent_t                        delivered;
            CSTADivertedEvent_t                         diverted;
            CSTAEstablishedEvent_t                established;
            CSTAFailedEvent_t                           failed;
            CSTAHeldEvent_t                             held;
            CSTANetworkReachedEvent_t             networkReached;
            CSTAOriginatedEvent_t                 originated;
            CSTAQueuedEvent_t                           queued;
            CSTARetrievedEvent_t                        retrieved;
            CSTAServiceInitiatedEvent_t     serviceInitiated;
            CSTATransferedEvent_t                 transfered;
            CSTACallInformationEvent_t            callInformation;
            CSTADoNotDisturbEvent_t                     doNotDisturb;
            CSTAForwardingEvent_t                 forwarding;
            CSTAMessageWaitingEvent_t             messageWaiting;
            CSTALoggedOnEvent_t                         loggedOn;
            CSTALoggedOffEvent_t                        loggedOff;
            CSTANotReadyEvent_t                         notReady;
            CSTAReadyEvent_t                            ready;
            CSTAWorkNotReadyEvent_t                     workNotReady;
            CSTAWorkReadyEvent_t                        workReady;
            CSTABackInServiceEvent_t              backInService;
            CSTAOutOfServiceEvent_t               outOfService;
            CSTAPrivateStatusEvent_t        privateStatus;
            CSTAMonitorEndedEvent_t               monitorEnded;
      } u;
} CSTAUnsolicitedEvent;
```

```
typedef struct
{
      InvokeID_t        invokeID;
      union
      {
            CSTAAlternateCallConfEvent_t              alternateCall;
            CSTAAnswerCallConfEvent_t                       answerCall;
            CSTACallCompletionConfEvent_t             callCompletion;
            CSTAClearCallConfEvent_t                  clearCall;
            CSTAClearConnectionConfEvent_t            clearConnection;
            CSTAConferenceCallConfEvent_t                   conferenceCall;
            CSTAConsultationCallConfEvent_t           consultationCall;
            CSTADeflectCallConfEvent_t                      deflectCall;
            CSTAPickupCallConfEvent_t                       pickupCall;
            CSTAGroupPickupCallConfEvent_t            groupPickupCall;
            CSTAHoldCallConfEvent_t                               holdCall;
            CSTAMakeCallConfEvent_t                               makeCall;
            CSTAMakePredictiveCallConfEvent_t makePredictiveCall;
            CSTAQueryMwiConfEvent_t                               queryMwi;
            CSTAQueryDndConfEvent_t                               queryDnd;
            CSTAQueryFwdConfEvent_t                               queryFwd;
            CSTAQueryAgentStateConfEvent_t            queryAgentState;
            CSTAQueryLastNumberConfEvent_t            queryLastNumber;
            CSTAQueryDeviceInfoConfEvent_t            queryDeviceInfo;
            CSTAReconnectCallConfEvent_t                    reconnectCall;
            CSTARetrieveCallConfEvent_t               retrieveCall;
            CSTASetMwiConfEvent_t                           setMwi;
            CSTASetDndConfEvent_t                           setDnd;
            CSTASetFwdConfEvent_t                           setFwd;
            CSTASetAgentStateConfEvent_t              setAgentState;
            CSTATransferCallConfEvent_t               ransferCall;
            CSTAUniversalFailureConfEvent_t           universalFailure;
            CSTAMonitorConfEvent_t                          monitorStart;
            CSTAChangeMonitorFilterConfEvent_t        changeMonitorFilter;
            CSTAMonitorStopConfEvent_t                      monitorStop;
            CSTASnapshotDeviceConfEvent_t                   snapshotDevice;
            CSTASnapshotCallConfEvent_t               snapshotCall;
            CSTARouteRegisterReqConfEvent_t           sysStatStart;
            CSTASysStatStopConfEvent_t                      sysStatStop;
            CSTAChangeSysStatFilterConfEvent_t        changeSysStatFilter;
      } u;
} CSTAConfirmationEvent;

#define CSTA_MAX_HEAP 1024
```

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