Chapter 4*CONTROL SERVICES*

This section defines the Application Programming Interface (API) Control ServicesXE "Control Services"§ (ACS) associated with the call control capabilities (CSTA-based services) supported by NetWare Telephony Services. ACS functions deal with the characteristics of the API interface (e.g. opening and closing the ACS interface). CSTA functions deal with the call and messaging mechanisms supported by the API Client Library. The ACS functions and events provide the application with:

- The ability to open and initialize a virtual communication channel (ACS stream for CSTA-based services) with any Telephony Server defined by the system
- The use of a blocked or polling mechanism to receive events
- The initialization of an Event Service Routine (ESR) mechanism for the notification of the arrival of solicited and unsolicited event messages from the server or the API Client Library. This mechanism may be different for each API environment.
- The ability to get a list of Telephony Servers on the network

- The ability to poll the Telephony Server for the support capabilities

Opening an ACS Stream

XE "Opening an ACS Stream"§In order to obtain Telephony Services from the Telephony Server an ACS stream or session must be established and initialized between the client and the server. This client/server connection is used to establish an application control session which is a logical link between the application using the API at the client PC and call processing software on the server. The session enables the application to request certain telephony control functions (e.g. making a call) be performed on its behalf and on behalf of the user. The server software then in turn performs the requested operation, e.g. establishing a call between the end-user telephone on the desktop and another station destination provided by the application. This session is established through the server which acts as the gateway or a "bridge" between the data and voice environments. This gateway/bridging function provides the logical voice/data integration needed at the application level in order to control the telephone on the users desktop. Each application must open its own ACS Stream before any services are requested.

The establishment of the client/server ACS Stream is accomplished using the **acsOpenStream**( )**XE "acsOpenStream**( )**"§** and receiving the complementary **ACSOpenStreamConfEventXE "ACSOpenStreamConfEvent"§** message. The **acsOpenStream**( ) is a request to establish an ACS Stream (for CSTA-based services) with the Telephony Server. The *acsHandle* returned by the **acsOpenStream**( ) function must be used by the application whenever the application accesses the ACS Stream. The ACS Stream may not be used by the application to request services from the Telephony Server until the corresponding **ACSOpenStreamConfEvent** has been received. The **acsOpenStream**( ) function is always a non-blocking call. The **acsOpenStream**( ) function will initiate communications with a

Telephony Server and will return to the application immediately. The application must monitor the *acsHandle* for the **ACSOpenStreamConfEvent** before requesting Telephony Services. After the **ACSOpenStreamConfEvent** has been successfully received for a ACS Stream, call control sessions can be established between the application and the Telephony Server. The application can start monitoring for telephony or other events associated with the user's telephone. The application should always check the **ACSOpenStreamConfEvent** to ensure that the connection has been established as requested by the application. The application is not guaranteed to get the API functional level requested in the **acsOpenStream**( ) function since these parameters depend on security access administration at the server. It is the responsibility of the application to dismantle the ACS Stream via the **acsCloseStream**( )**XE "acsCloseStream**( )**"§** or the **acsAbortStream**( )**XE "acsAbortStream**( )**"§** function to ensure that all system resources associated with the ACS Stream have been released.

The dismantling of the client/server ACS Stream can be accomplished by using the **acsCloseStream**( )**XE "acsCloseStream**( )**"§** function and receiving the complementary **ACSCloseStreamConfEvent** XE "ACSCloseStreamConfEvent "§ message. The **acsCloseStream**( ) function will begin an orderly disconnect of the ACS Stream to the Telephony Server, and the application may not request any services from the Telephony Server after the **acsCloseStream**( ) function has returned. The *acsHandle* that the application passes to the **acsCloseStream**( ) function remains in effect until the application has received the **ACSCloseStreamConfEvent.** The reception of the **ACSCloseStreamConfEvent** by the application frees all system resources associated with the ACS Stream, and invalidates the *acsHandle.* The **ACSCloseStreamConfEvent** is guaranteed to be the last event the application will receive on the ACS Stream. All call control sessions that existed between the application and the server will be dismantled during the procedures associated with the **acsCloseStream()** function**.** The calls on the switch, however, are not dropped due to the close request, just the call associations maintained by the Telephony Server.

The **acsCloseStream**( ) function is always a non-blocking call. The **acsCloseStream**( ) function will return to the application immediately after it has initiated orderly disconnect procedures from the Telephony Server, but the application may still receive events on the *acsHandle* associated with the ACS Stream. The application must continue to poll until it receives the **ACSCloseStreamConfEvent** so that system resources can be released.

If the application does not require confirmation, it can use the **acsAbortStream**( ) **XE "acsAbortStream**( ) **"§**function to unilaterally dismantle an ACS Stream. The **acsAbortStream**( ) function is a non-blocking function and will return to the application immediately. When the **acsAbortStream**( ) function returns, the *acsHandle* is invalid. The API Client Library will clean up all resources associated with this handle, including any events associated with this handle not received by the application. The abort stream request will be passed to the Telephony Server, which will also clean up all resources and call control sessions that existed between the application and the server. The calls on the switch, however, are not dropped due to the abort request, just the call associations maintained by the Telephony Server. There is no confirmation event for an **acsAbortStream**( ) call.

An application may open several ACS Streams, but is restricted to one stream per advertised telephony service. A Telephony Server advertises each registered PBX Driver service. Since the PBX Drivers are vendor dependent and may support multiple CTI links, an application cannot make any correlation between a registered PBX Driver service and the number of underlying physical CTI links.

Sending Requests and Responses

XE "Sending Requests and Responses"§After the ACS Stream is successfully opened and the **ACSOpenStreamConfEvent** is received, the application can make requests of the Telephony Server (and respond to Telephony Services requests) by applying the *acsHandle* to the functions defined in the Switching Function Services section of this document. Each function request to the Telephony Server requires an *invokeID* (which can be application or library generated) that will be returned in the confirmation event of the function call. The *invokeID* can be used by the application to match the confirmation event (or failure event) to the corresponding request. An application specifies whether *invokeID*s will be application or library generated by specifying this choice in one of the parameters to the **acsOpenStream** function. Once the *invokeID* type has been selected for an ACS stream via the **acsOpenStream** function, it cannot be changed for this stream.

Receiving Events

XE "Receiving Events"§
When the ACS Stream is successfully opened, the API Library has at least one message, (i.e. the **ACSOpenStreamConfEvent**) queued for the application. In order to retrieve this event and subsequent messages, the application must use the **acsGetEventBlock**( ) **XE "acsGetEventBlock**( ) **"§**(blocking mode) or the **acsGetEventPoll**( )**XE "acsGetEventPoll**( )**"§** (non-blocking mode) function. The *blocking* and *non-blocking* modes are defined as follows:

- **Blocking -** an application can choose to use the **acsGetEventBlock**( ) function which gets the next event or blocks if no events are available in the API

Library event queue. The API Library event queue contains messages for all ACS streams opened by this application. An application indicates which ACS stream it wishes to receive a message from by specifying the *acsHandle* in the **acsGetEventBlock**( ) call. Using this mechanism, the application that requested an event will be "put to sleep" and be awakened only when an event has occurred. This event handling mechanism is useful for applications which monitor a station on the switch and only require CPU cycles for processing when an event has occurred.

- **Non-Blocking** - If the application cannot be blocked while waiting for an event, a *non-blocking* mechanism is also supported by the API Client Library. This mechanism allows the application to get events at its own schedule or pace by simply polling for the event when events are required. Polling is accomplished using the **acsGetEventPoll**( ) function which is called every time the application wants to process an event. If the **acsGetEventPoll**( ) function call is successful, it returns the next event in the API Client Library event queue. The API Library event queue contains messages for all ACS streams opened by this application. An application indicates which ACS stream it wishes to receive a message from by specifying the *acsHandle* in the **acsGetEventPoll**( ) call. When the API Library event queue is empty the function returns immediately with a "no message" cause code.

The application must poll the API Library event queue often enough so that the queue does not overflow. The API Library will stop acknowledging messages from the Telephony Server when the queue fills up, ultimately resulting in a loss of the stream.

The API library provides a rudimentary ***Event Service Routine*** (ESR)XE "Event Service Routine (ESR)"§ mechanism for the asynchronous notification of the arrival of incoming events. The application may specify an ESR function in **acsSetESR**( )**XE "acsSetESR**( )**"§**. The ESR mechanism is intended only for the *notification* of incoming events from the API Library, and does not *process* incoming events. The application can use the ESR mechanism to trigger platform specific events (i.e. post a Windows™ message for the application, or signal a semaphore in the NetWare® environment). Note that the application must still use **acsGetEventBlock**( ) or **acsGetEventPoll**( ) to *receive* the message. When a message is known to be available, it does not matter which function is used to retrieve it.

API Control Services (ACS) Functions and Confirmation Events

This section defines the ACS functionsXE "ACS functions"§ associated with the Telephony Server's ACS Services. These functions are used to open and manage events on the ACS Stream between client workstation and the Telephony Server.

acsOpenStream ( )

XE "acsOpenStream ( )"§**acsOpenStream**( ) opens a communications session (ACS message stream) to the Telephony Server. This stream is required in order to access the other ACS Control Services supported by the client library and the CSTA client Telephony Services supported by the server and must be called before any other ACS or CSTA services is requested. **acsOpenStream**( ) immediately returns an acsHandle; a confirmarion 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
This is the value of the unique handle to the opened ACS Stream returned by the function call. This handle is determined by the API Client Library and is unique to the ACS Stream being opened. Once the open function is successful, this handle must be used in all other function calls to the API. If the open is successful, the application is guaranteed to have a valid handle available upon return from this call. If the open is not successful, then the

function return code will contain the cause of the failure.

invokeIDType

This parameter sets the type of invoke identifiers which will be use for the Stream being opened by the application.

The possible types are: Application-Generated Invoke IDs (**APP_GEN_ID**) or Library generated invoke identifiers (**LIB_GEN_ID**).

When **APP_GEN_ID** is selected then the application will provide an ID with every function call in the API that requires an *invokeID.* This same invoke ID value will be returned to the application in the confirmation event for the specific instance of a service request. Application-generated invoke IDs can assume 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

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.

streamType

This parameter allows the application to specify the type of stream required by the application. For the API release defined in this document the possible values are:

ST_CSTA - requests that a CSTA call control stream or session be opened to a Telephony Server. This stream can be used for service requests and responses which begin with the prefix "csta" or "CSTA".

ST_OAM - request that an OAM stream is opened with the Telephony Server.

serverID
This is a pointer to a null-terminated string of maximum size **ACS_MAX_SERVICEID**. This string contains the name of the server (in ASCII format) to be used for providing services requested in the **streamType** parameter.

loginID
This is a pointer to a null terminated string of maximum size **ACS_MAX_LOGINID**, which contains the login ID of the user who wishes to use the server specified in the ***serviceID*** parameter.

passwd
This is a pointer to a null terminated string of maximum size **ACS_MAX_PASSWORD**, which contains the password of the user who wishes to use the server specified in the ***serverID*** parameter.

applicationName
This is a pointer to a null terminated string of maximum size **ACS_MAX_APPNAME**, which contains a user selected name for the application being run. The application name is used in a display only mode by administration and maintenance functions and is intended to aid administrators in debugging misbehaved applications.

acsLevelReq
The acsLevel is not supported for the Telephony Server.

This field is ignored by the implementation of this API.

apiVer
This parameter is used to specify the API Version being requested by the application. The API Client Library will support different API functionality depending on the version being used in order to maintain backwards compatibility. As the API is enhanced with new capabilities not supported by older applications, this parameter allows existing applications to request older version of the API for compatibility reasons. The API version of a particular Software Development Kit (SDK) is specified by **CSTA_API_VERSION** in the *csta.h* header file.

sendQSize
This parameter specifies the maximum number of outgoing messages the API Client Library will queue before returning **ACSERR_QUEUE_FULL**. If this parameter is set to zero (0), then a default queue size will be used.

sendExtraBufs
This parameter specifies the number of additional packet buffers to be allocated to the send queue. If this parameter is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If many messages are expected to exceed the size of a network packet, as in the case where private data is used extensively, or if the **ACSERR_NOBUFFERS** error is frequently reported, additional buffers may be allocated via this parameter.

recvQSize
This parameter specifies the maximum number of incoming messages the API Client Library will queue before it ceases acknowledgment to the Telephony Server.

If this parameter is set to zero (0), then a default queue size will be used.

recvExtraBufs
This parameter specifies the number of additional packet buffers to be allocated to the receive queue. If this parameter is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If many messages are expected to exceed the size of a network packet, as in the case where private data is used extensively, or if the **ACSERR_STREAM_FAILED** error is frequently reported, additional buffers may be allocated via this parameter.

privateData
This points to a data structure which defines any implementation-specific initialization 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. A NULL pointer may be used to specify no *private* data.

Return Values

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

- *Library-generated Identifiers* - if the function call completes successfully it will return a positive value, 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 **ACSOpenStreamConfEvent** 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_APIVERDENIED
This return indicates that the API Version requested is invalid and not supported by the existing API Client Library.

ACSERR_BADPARAMETER
One or more of the parameters is invalid.

ACSERR_DUPSTREAM
This return indicates that an ACS Stream is already established with the requested Server.

ACSERR_NODRIVER
This error return value indicates that no API Client Library Driver was found or installed on the system.

ACSERR_NOSERVER
This indicates that the requested Server is not present in the network.

ACSERR_NORESOURCE
This return value indicates that there are insufficient resources to open a ACS Stream.

Comments

The **acsOpenStream**( ) function enables an application to open a network or local communication channel (ACS Stream) with a Server device. The stream to be opened will initiate and establish a ACS client/server session between the application and the Server. This session can be used to access all the server supported services (e.g. for the Telephony Server this would be **cstaMakeCall**, **cstaTransferCall**, etc.). The ACS session must be opened in order to obtain a *acsHandle* to the stream. This handle must be obtained before any other function can be called since a valid handle is required by all service requests.

Once this function is called successfully, the application must use the given handle and wait for a **ACSOpenStreamConfEvent** to determine the status of the ACS Stream. Only one ACS Stream is allowed per application to a single ***serverID***. Multiple calls by the same application to the **acsOpenStream**( ) function are allowed assuming a different ***serverID*** is used in each **acsOpenStream**( ) service request or different stream type.

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 a network.

The Client is responsible for calling the **acsCloseStream**( ) function and receiving the **ACSCloseStreamConfEvent** or calling the **acsAbortStream**( ) function to dismantle the ACS Stream . It is very important that an application close an active stream during its exit or cleanup routine in order to free resources in the client and server for other applications on the network.

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

The application may only call the **acsEventNotify(), acsSetESR()**, **acsGetEventBlock(), acsGetEventPoll**( ) and **acsCloseStream**( ) functions before it has received the **ACSOpenStreamConfEvent**.

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 indicating that a failure has occurred on the stream.

ACSOpenStreamConfEventXE "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 ( )"§

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.

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 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.

ACSCloseStreamConfEventXE "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.

ACSUniversalFailureConfEventXE "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 listed for **ACSUniversalFailureEvent** in the "*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.

Application Notes

None**.**

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

Application Notes

None

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.

acsSetESR( ) XE "acsSetESR( ) "§

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). A NULL pointer indicates no 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**( ).

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);
}
```

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 0, then all active ACS Streams for this application will be flushed.

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 servers 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
ndicates 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 servers, invoking the callback function for each server 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"§

This section covers the unsolicited ACS Status Events.

ACSUniversalFailureEventXE "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 ***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
                  {
                    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 indicate the cause values for the ACS Stream failure defined by the current active 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 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 errors

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

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

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

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

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

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

Tserver Free Buffer Failed
XE "Tserver Free Buffer Failed"§ The Tserver was unable to free Tserver Driver Interface (TDI)   memory.

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

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

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

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

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

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

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

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

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

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

Tserver No Version
XE "Tserver No Version"§ The Tserver processed a

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

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

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

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

Tserver Invalid Message
XE "Tserver Invalid Message"§   The Tserver received an invalid Tserver OAM message.

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

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

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

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

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

Tserver Bad Server ID
XE "Tserver 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 "Tserver Bad Stream Type"§XE "Tserver Bad Stream Type"§ The stream type an ACSOpenStream request was invalid.

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

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

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

Tserver Device Not On List
XE "Tserver Device Not On List"§The specified device in an

API call was not found on any device list administered for this user.

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

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

Tserver No Home Permission
The Tserver rejected a service request because the device did not match a device associated with a home worktop.

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

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

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

Tserver Insufficient Permission
XE "Tserver Insufficient Permission"§The device in the API call is at the users away worktop and the device

has a higher permission level than the user, preventing the user from controlling the device.

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

Driver errors

*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 "Driver 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 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"§A CSTA request was issued with no prior ACSOpenStream request. The request is rejected.

Driver Invalid Class Rejection

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

Driver Generic Rejection

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

Driver Resource Limitation

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

Driver ACSHandle Termination

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

Driver Link Unavailable

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

Comments

None.

Application Notes

None.

ACS Data TypesXE "ACS Data Types"§

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

ACS Common Data Types

This section specifies the common ACS data types.

```
typedef int RetCode_t;

#define     ACSPOSITIVE_ACK 0       /* The function was successful */

/* Error Codes */

#define     ACSERR_APIVERDENIED         -1      /* This return indicates that the
                                                 * API Version requested is invalid
                                                 * and not supported by the
                                                 * existing 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      /* This indicates that the requested
                                                 * Server is not present in the
                                        * network.
                                                 */

#define     ACSERR_NORESOURCE           -6      /* This return value indicates that
                                                 * there are insufficient resources
                                                 * to open a ACS Stream.
                                                 */

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

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

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

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

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

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

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


typedef unsigned longInvokeID_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 TypesXE "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_tfailureEvent;
        } u;
} ACSConfirmationEvent;
```

CSTA Control Services and Confirmation Events

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

cstaGetAPICaps( )XE " cstaGetAPICaps( )"§

This is used to obtain 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.

Comments

None

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	No security checks.
ASC_ONLY	Check ACSOpenStream requests only
ASC_AND_CSTA_CHECKING	Check ACSOpenStream and all applicable CSTA messages

index
This parameter indicates to the client application the

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

devlist

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

Comments

None.

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.

Comments

None

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 ***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
                  {
                   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.

Comments

None.

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 ***section 10 - 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
```