Dialogic®

Dialogic® Conferencing API

Programming Guide and Library Reference

October 2012

05-2506-004

Publication Date: October 2012

Document Number: 05-2506-004

Contents

Revision History

This revision history summarizes the changes made in each published version of this document.

Document No.	Publication Date	Description of Revisions
05-2506-004	October 2012	Added programming guide content to create a combined Programming Guide and Library Reference. Programming Guide content: Added Product Description, Programming Models, Event Handling, Error Handling, Application Development Guidelines, Building Applications. Reorganized and included Using Active Talker and Using Volume Control chapters as part of Application Development Guidelines chapter. Application Development Guidelines: Added Privilege Talker, Monitor Mode, Mute Audio, HD Voice Conferencing (Wideband Audio Conferencing) sections. Added Windows specific information in Creating a Conference section. Building Applications: Added a note about compiling applications in Required Libraries, Linux section. Function Summary by Category: Added cnf_GetPrivilegeTalkerList(), cnf_Listen(), cnf_UnListen(), cnf_GetXmitSlot(). cnf_DisableEvents( ) and cnf_EnableEvents( ): Added ECNF_CONF_EVT_EXCEEDED_CONF_LICENSE. cnf_GetAttributes( ) and cnf_SetAttributes( ): Added ECNF_CONF_ATTR_MAX_ACTIVE_TALKERS, ECNF_PARTY_ATTR_PRIVILEGE, ECNF_PARTY_ATTR_MUTE. cnf_GetPrivilegeTalkerList( ): Added for privilege talker attribute support. cnf_GetXmitSlot( ): Added for monitor mode support. cnf_Listen( ): Added for monitor mode support. cnf_UnListen( ): Added for monitor mode support. Events chapter: Added CNFEV_GET_PRIVILEGE_TALKER and CNFEV_GET_PRIVILEGE_TALKER_FAIL to Termination Events. Added CNFEV_EXCEEDED_CONF_LICENSE to Notification Events. CNF_PRIVILEGE_TALKER_INFO structure: Added. CNF_VISIBLE_PARTY_LIST structure: Updated structure definition. (IPY00080538) SC_TSINFO structure: Added.
05-2506-003 (continued on next page)	October 2007	Function Summary by Category chapter: Added the Multimedia Conferencing Functions section and added new functions to the Device Management Functions section. cnf_AddParty( ): Added information about MCX device in Cautions section. cnf_GetAttributes( ): Added the ECNF_CONF_ATTR_NOTIFY parameter.

Document No.	Publication Date	Description of Revisions
05-2506-003 (continued)	October 2007	cnf_EnableEvents( ) and cnf_DisableEvents( ): Added note about ECNF_BRD_EVT_ACTIVE_TALKER event type. cnf_GetVideoLayout( ): Added function for multimedia conferencing support. cnf_GetActiveTalkerList( ): Added information about MCX device. cnf_GetVisiblePartyList( ): Added function for multimedia conferencing support. cnf_Open( ) and cnf_OpenEx( ): Added description of new device name, MCX conferencing device. cnf_OpenConference( ): Added information about MCX device. cnf_OpenEx( ): Added function to open devices in synchronous and asynchronous mode. cnf_OpenParty( ): Added information about MCX device. cnf_RemoveParty( ): Added information about MCX device in Cautions section. cnf_ResetDevices( ): Added function to reset devices. cnf_SetAttributes( ): Added the ECNF_CONF_ATTR_NOTIFY parameter. cnf_SetVideoLayout( ): Added function for multimedia conferencing support. cnf_SetVisiblePartyList( ): Added function for multimedia conferencing support. Events chapter: Added new termination events for multimedia conferencing support. Data Structures chapter:: Added multimedia conferencing related structures. CNF_VIDEO_LAYOUT_INFO: Added for multimedia conferencing support. CNF_VISIBLE_PARTY_INFO: Added for multimedia conferencing support. CNF_VISIBLE_PARTY_LIST: Added for multimedia conferencing support. Supplementary Reference Information chapter: Added new example code.
05-2506-002	August 2007	Made global changes to reflect Dialogic brand.
05-2506-001	August 2006	Initial version of document.

About This Publication

The following topics provide more information about this publication:

Purpose

This publication describes the features of the Dialogic® Conferencing (CNF) API library and provides guidelines for developing applications using the conferencing API. It also provides a reference to functions, parameters, data structures, and error codes in the conferencing API.

In this document, the term "board" refers to the virtual Dialogic® DM3 board.

Note: The Dialogic® Conferencing (CNF) API is distinct from and presently incompatible with the Dialogic® Conferencing (CNF) API that was previously released in Dialogic® System Release 6.0 on PCI for Windows.

Applicability

This document version is published for Dialogic® Host Media Processing (HMP) Software Release 4.1LIN and Dialogic® Host Media Processing (HMP) Software Release 3.0WIN.

This document may also be applicable to other software releases (including service updates) on Linux or Windows operating systems. Check the Release Guide for your software release to determine whether this document is supported.

Intended Audience

This publication is intended for the following audience:

- Distributors
- System Integrators
- Toolkit Developers
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)

- Original Equipment Manufacturers (OEMs)
- End Users

How to Use This Publication

This document assumes that its readers are familiar with the Linux or Windows operating systems and the C++ programming language.

The information in this document is organized in two major parts:

- *Programming Guide* content, which describes the conferencing software features, programming models, application development guidelines, and feature implementation guidelines.
- *Library Reference* content, which provides an alphabetical reference to the conferencing functions, events, data structures, and error codes.

Related Information

See the following additional information:

- *http://www.dialogic.com/manuals/* (for Dialogic® product documentation)
- *http://www.dialogic.com/support/* (for Dialogic technical support)
- *http://www.dialogic.com/* (for Dialogic® product information)

Product Description 1

This chapter provides an overview of the Dialogic® Conferencing (CNF) API library. Topics include:

1.1 Overview

The Dialogic® Conferencing (CNF) API software supports development of conferencing applications on Dialogic® Host Media Processing (HMP) Software. The conference can take place over an IP network and/or over traditional public switched telephone network (PSTN) lines.

Dialogic® HMP Software performs media processing tasks on general-purpose servers based on Intel architecture without the need for specialized hardware. When installed on a system, Dialogic® HMP Software performs like a virtual Dialogic® DM3 board to the customer application, but media processing takes place on the host processor. In this document, the term "board" represents the virtual Dialogic® DM3 board.

Note: This Dialogic® Conferencing (CNF) API is distinct from and incompatible with the Dialogic® Conferencing (CNF) API that was previously released in Dialogic® System Release 6.0 on PCI for Windows®.

1.2 Key Features

Key features of the Dialogic® Conferencing (CNF) API software include the following:

Asynchronous programming model support
: This model enables multiple channels to be handled in a single process and supports higher density conferencing solutions.

Support for conferees from multiple sources
: Participants in a conference may come from a variety of sources, such as a voice device and an IP media device. The software allows for flexibility to grow and support additional sources.

Conference bridging
: Multiple conferences can be bridged together so that all parties (also called conferees) in two or more established conferences can communicate with one another.

HD voice conferencing (wideband audio conferencing)
: High definition (HD) voice conferencing, also called wideband audio conferencing, is supported using G.722 and G.722.2 (AMR-WB) audio codecs. The conference can consist of a combination of wideband and narrowband parties.

Coach/pupil feature
: Two selected parties can establish a private communication link within the overall conference. The coach is a private member of the conference and is only heard by the pupil. However, the pupil cannot speak privately with the coach.

DTMF digit detection
: The application can determine whether a party has generated a DTMF digit.

Volume control
: A party can adjust the listening volume of the conference using pre-programmed DTMF digits.

DTMF tone clamping
: This feature mutes dual tone multi-frequency (DTMF) tones heard during a conference. Tone clamping applies to the transmitted audio going into the conference and does not affect DTMF function. It can be enabled on a board, conference, or party basis.

Automatic gain control (AGC)
: AGC is an algorithm for normalizing an input signal to a target level. The AGC algorithm discriminates between voiced and unvoiced signals within a conference.

Active talker
: The active talker feature sums the three most active talkers in a conference, so that the conversation doesn't get drowned out when too many people talk at once.

Conference monitoring
: Participants have listen-only access to a conference.

Echo cancellation
: This feature reduces echo from the incoming signal, improving the quality of a conference for all participants.

Tariff tone
: A party can receive a periodic tone for the duration of the conference call.

1.3 Understanding How Conferences are Formed

Developing a conferencing application requires the use of the Dialogic® Conferencing (CNF) API library as well as other Dialogic® API libraries, such as the Dialogic® Standard Runtime Library (SRL) and the Dialogic® Device Management API library. Other libraries include the IP media and voice libraries.

A conference consists of conferees (also known as parties). The maximum number of conferences and parties supported varies with the Dialogic® HMP Software license in use and, if applicable, the media load in use on the board.

A conference is identified by a unique conference device handle, which is registered with the Dialogic® Standard Runtime Library (SRL). A party is identified by a unique SRL party device handle. The virtual board device is the parent device for the conference device and party device; it

has a unique SRL device handle. For more information on the types of conferencing devices, see Section 5.3, "Creating a Conference", on page 22.

The Dialogic® Conferencing (CNF) API is used to open a conference, and to add parties to a conference. However, these parties cannot participate in a conference until they are connected to a technology device handle through the **dev_Connect( )** Dialogic® Device Management API function. Technology device handles are obtained through the respective technology API library functions. For example, the dxxxB1C1 voice channel device handle is obtained from **dx_open( )**.

A conference may be formed from parties that are connected to any one of the following technology device handles:

- voice (dx) device handle
- IP media (ipm) device handle
- digital network interface (dti) device handle

Note: A device handle obtained from **gc_OpenEx( )** in the Dialogic® Global Call API library cannot be used by **dev_Connect( )** to connect a party to a conference. Rather, you can use the device handle returned by **gc_GetResourceH( )** to connect a party to a conference.

1.4 Relationship with Other Libraries

A conferencing application is developed using the Dialogic® Conferencing (CNF) API library as well as other Dialogic® API libraries, including the following:

- Dialogic® Standard Runtime Library (SRL)
- Dialogic® Device Management API Library
- Dialogic® Voice API Library
- Dialogic® IP Media Library API
- Dialogic® Global Call API Library
- Dialogic® Digital Network Interface API Library

1.4.1 Dialogic® Standard Runtime Library (SRL)

The Dialogic® Standard Runtime Library (SRL) provides a common interface for event handling and other functionality common to all devices.

The Dialogic® Conferencing (CNF) API uses three types of devices: virtual board device, conference device, and party device. TheDialogic® Conferencing (CNF) API registers the virtual board device with the Dialogic® Standard Runtime Library (SRL) when **cnf_Open( )** is called. In addition, the conference device and the party device are registered when **cnf_OpenConference( )** and **cnf_OpenParty( )**, respectively, are called. Conferencing events are posted to the SRL, which then delivers these events to the application. For more information about SRL functions, see the *Dialogic® Standard Runtime Library API Library Reference*.

1.4.2 Dialogic® Device Management API Library

The Dialogic® Device Management API library provides run-time control and management of configurable system devices. It includes functions to reserve resources and to manage the connections between devices. It performs all necessary connection-related operations, including time slot management.

The device connection functions enable connection between conferencing devices and other devices on Dialogic® HMP Software, providing the ability for conferencing communication. Before a party can participate in a conference, it must be connected to a supported technology device (such as voice and IP media) using the **dev_Connect( )** function. Conference bridging is also accomplished through the Dialogic® Device Management API library. For more information about device management functions, see the *Dialogic® Device Management API Library Reference*.

1.4.3 Dialogic® Voice API Library

The Dialogic® Voice API provides a collection of functions supporting call processing such as dual tone multifrequency (DTMF) detection, tone signaling, playing and recording. You may add a party to a conference using a device handle obtained from **dx_open( )**. You must then connect the voice device to a conference using **dev_Connect( )**. For more information about voice functions, see the *Dialogic® Voice API Library Reference*.

1.4.4 Dialogic® IP Media Library API

The Dialogic® IP Media Library API provides a collection of functions for media control on IP devices. You may add a party to a conference using a device handle obtained from **ipm_Open( )**. You must then connect the IP media device to a conference using **dev_Connect( )**. For more information about Dialogic® IP Media Library API functions, see the *Dialogic® IP Media Library API Library Reference*.

1.4.5 Dialogic® Global Call API Library

The Dialogic® Global Call API provides a collection of functions supporting call control operations. You may add a party to a conference using a device handle obtained from **gc_GetResourceH( )**. You must then connect the device to a conference using **dev_Connect( )**. For more information about Dialogic® Global Call API functions, see the *Dialogic® Global Call API Library Reference*.

1.4.6 Dialogic® Digital Network Interface API Library

The Dialogic® Digital Network Interface API is used to manage digital network interface devices. You may add a party to a conference using a device handle obtained from **dt_open( )**. You must then connect the device to a conference using **dev_Connect( )**. For more information about Dialogic® Digital Network Interface API functions, see the *Dialogic® Digital Network Interface Software Reference*.

Programming Models 2

This chapter describes the programming models supported by the Dialogic® Conferencing (CNF) API software. The following topics are covered:

2.1 Programming Models Overview

The Dialogic® Conferencing (CNF) API software supports application development using asynchronous programming models. By usage, the asynchronous models are often said to use asynchronous **mode**. Asynchronous mode programming is introduced briefly in this chapter and described in more detail in the *Dialogic® Standard Runtime Library API Programming Guide*.

Note: The Dialogic® Conferencing (CNF) API library is implemented as an asynchronous only library. If desired, you can implement synchronous functionality in the application itself.

2.2 Asynchronous Programming Model

Asynchronous mode programming is characterized by allowing other processing to take place while a function executes. In asynchronous mode programming, multiple channels are handled in a single process rather than in separate processes as required in synchronous mode programming.

An asynchronous mode function typically receives an event from the Dialogic® Standard Runtime Library (SRL) indicating completion (termination) of the function in order for the application to continue processing a call on a particular channel. A function called in the asynchronous mode returns control to the application after the request is passed to the device driver. A termination event is returned when the requested operation completes.

Caution: In general, when a function is called in asynchronous mode, and an associated termination event exists, the **cnf_Close( )** function should not be called until the termination event has been received.

For Linux environments, the asynchronous models provided for application development include:

Asynchronous (Polled)

In this model, the application polls for or waits for events using the **sr_waitevt( )** function. When an event is available, event information may be retrieved using SRL event handling functions such as **sr_getevttype( )**. Retrieved event information is valid until the **sr_waitevt( )** function is called again. Typically, the polled model is used for applications that do not need to use event handlers to process events.

Asynchronous with Event Handlers

This model may be run in non-signal mode only. Event handlers can be enabled or disabled for specific events on specific devices.

Event Handling 3

All conferencing events are retrieved using Dialogic® Standard Runtime Library (SRL) event retrieval mechanisms, including event handlers. The SRL is a device-independent library containing event management functions and Standard Attribute functions. This chapter lists SRL functions that are typically used by conferencing applications.

3.1 Dialogic® Standard Runtime Library Event Management Functions

SRL event management functions retrieve and handle device termination events for certain library functions. Applications typically use the following functions:

sr_enbhdlr( )
enables event handler

sr_dishdlr( )
disables event handler

sr_getevtdev( )
gets device handle

sr_getevttype( )
gets event type

sr_waitevt( )
waits for next event

sr_waitevtEx( )
waits for events on certain devices

Note: See the *Dialogic® Standard Runtime Library API Library Reference* for function details.

3.2 Dialogic® Standard Runtime Library Standard Attribute Functions

SRL Standard Attribute functions return general device information, such as the device name or the last error that occurred on the device. Applications typically use the following functions:

ATDV_ERRMSGP( )
pointer to string describing the error that occurred during the last function call on the specified device

ATDV_LASTERR( )
error that occurred during the last function call on a specified device. See the function description for possible errors for the function.

ATDV_NAMEP( )
pointer to device name

ATDV_SUBDEVS( )
number of subdevices

Note: See the *Dialogic® Standard Runtime Library API Library Reference* for function details.

Error Handling 4

This chapter describes error handling for the Dialogic® Conferencing (CNF) API software.

All Dialogic® Conferencing (CNF) API functions return a value that indicates the success or failure of the function call. Success is indicated by a return value of CNF_SUCCESS. Failure is indicated by a value of CNF_ERROR.

If a function fails, call the Dialogic® Standard Runtime Library API functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )** for the reason for failure. These functions are described in the *Dialogic® Standard Runtime Library API Library Reference*.

If an error occurs during execution of an asynchronous function, the CNFEV_ERROR event is sent to the application. No change of state is triggered by this event. Upon receiving the CNFEV_ERROR event, the application can retrieve the reason for the failure using the Dialogic® Standard Runtime Library API functions **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**.

Application Development Guidelines 5

This chapter contains guidelines for developing Dialogic® Conferencing (CNF) API applications. The following topics are covered:

5.1 Using Symbolic Defines

The numerical values of defines may not remain the same as new versions of the software are released. It is recommended that you do not use a numerical value in your application when an equivalent symbolic define is available. Symbolic defines are found in the header files; for example, *cnflib.h*, *cnfevts.h, cnferrs.h,* and *srllib.h*.

5.2 Using Conferencing Devices

The types of devices used in the Dialogic® Conferencing (CNF) API library and their naming convention are as follows:

- virtual board device, called cnfB*x*, where *x* is the logical board number
- conference device, called cnfB*x*C*y*, where *x* is the logical board number and *y* is the conference device channel

- party device, called ptyB*x*P*z*, where *x* is the logical board number and *z* is the party device channel

All devices are identified by a unique SRL handle. All subsequent references to the opened device must be made using the handle, until the device is closed.

The virtual board device is the parent device for both the conference device and the party device. You must open a virtual board device before opening a conference device or party device. After a board device is opened, you can open and initialize all conference devices at once, and/or all party devices at once. A conference device and a party device are independent; that is, you can open a party device without first opening a conference device.

5.3 Creating a Conference

The following steps describe how to create a conference. See the Glossary for information on the terms used here. See the *Dialogic® Conferencing API Library Reference* for details on conferencing functions and data structures.

Note: These steps provide general guidelines. They do not cover all tasks required to write a conferencing application.

1. Use the asynchronous programming model, and enable a Dialogic® Standard Runtime Library (SRL) event handler for the various devices used by the conferencing software (virtual board, conference, and party) via **sr_enbhdlr( )**.
2. Open the virtual board device handle using **cnf_Open( )**. The device naming convention for the virtual board is **cnfBx**, where **x** is the board number starting at 1. You must have a virtual board device before you can open a conference device or a party device.
3. Get a count of the resources on this board using **cnf_GetDeviceCount( )**. This count is a snapshot in time. The CNF_DEVICE_COUNT_INFO data structure contains information about the number of devices on this board, such as the maximum number of conferences and parties, as well as the number of free conferences and free parties. The maximum number of conferences and parties supported varies with the Dialogic® Host Media Processing (HMP) Software license in use and, if applicable, the media load in use on the board. Having a count of the resources enables you to properly manage these resources.

   For information about resource limitations on Windows, see Resource Limitations on Windows Operating Systems following these steps.
4. If desired, specify attributes for the board using **cnf_SetAttributes( )**. Attributes are contained in the CNF_ATTR data structure. Use **cnf_GetAttributes( )** to return the current attributes for the board.
5. If desired, enable notification events for the board using **cnf_EnableEvents( )**. Events are contained in the CNF_EVENT_INFO data structure. For example, the application can be notified dynamically whenever a conference is opened or a party is added.
6. At this point, you can choose to open and set up all conferences; or you can choose to open one conference at a time as needed. Similarly, you can also choose to open and set up all parties, or open one party at a time as needed. The steps that follow show how to open one conference, then add a party to this opened conference. Repeat the steps as appropriate for your use case.

7. Using **cnf_OpenConference( )**, create a new conference to which parties will be added. This function takes the virtual board device handle returned by **cnf_Open( )** as an argument. It returns a unique SRL device handle for the conference. The conference created consumes a conference resource.
8. If desired, specify attributes for the conference using **cnf_SetAttributes( )**. Attributes are contained in the CNF_ATTR data structure. Use **cnf_GetAttributes( )** to return the current attributes for the conference.
9. If desired, enable notification events for the conference using **cnf_EnableEvents( )**. Events are contained in the CNF_EVENT_INFO data structure.
10. Open a party device handle using **cnf_OpenParty( )**. This function returns a unique SRL device handle for the party.
11. If desired, you can specify attributes for a party using **cnf_SetAttributes( )**. Attributes are contained in the CNF_ATTR data structure. Use **cnf_GetAttributes( )** to return the current attributes for the party.
12. Before a party can participate in a conference, you must connect this party to a supported technology device using **dev_Connect( )**. Examples of supported technology devices include a voice device (dxxxB1C1) and an IP device (ipmB1C1). See Section 1.3, "Understanding How Conferences are Formed", on page 12 for details on supported technology devices. See the *Dialogic® Device Management API Library Reference* for details on device management functions.

    Note: Depending on your use case, you can choose to issue **dev_Connect( )** either before or after performing the **cnf_AddParty( )** operation in Step 13. If you issue **dev_Connect( )** after adding a party, you must wait for this function to successfully complete before streaming can take place.
13. Using **cnf_AddParty( )**, add a party to the conference created in step 7. This function takes the party device handle returned by **cnf_OpenParty( )** as an argument. The party created consumes a party resource.
14. Add more parties to the conference as needed. There is a limit to the number of parties that can be added to a conference (the count of resources was obtained in step 3). However, if the limit is reached, you can add parties using the conference bridging feature. For more information on bridging, see Section 5.4, "Conference Bridging", on page 24.
15. Terminate your application in an orderly fashion. For example, disable events, close all devices, and so on. For more information, see Section 5.5, "Terminating an Application", on page 25.

Resource Limitations on Windows Operating Systems

When a license has voice, RTP, and conferencing devices, there are cases where not all of the number of conference parties as mentioned in the license will be opened due to an internal limitation of 2048 transmit time slots on Windows. This is because the Dialogic Windows stack has an internal limitation on the maximum number of time slots it can allocate system wide, currently 2048 time slots. This limitation of 2048 is only on Windows operating systems due to memory constraints.

The Dialogic stack allocates the transmit time slots for voice and RTP devices when the devices are created during board download, while the transmit time slots for the audio conference party is allocated when a party is added to a conference. Each device requires a different number of time

slots, hence Dialogic stack may not be able to open the exact number of devices mentioned in the license if the time slot usage exceeds the 2048 time slot limit.

Consider the following examples provided herein. Note that each voice device and each RTP requires one transmit time slot each, while each conference party requires two transmit time slots internally.

```
License of 520r520v0e520c0s0f0i0m_host.lic namely 520 G.711, 520 Voice and 520 Conference
Parties
```

At board download, all voice and RTP channels together will consume 1040 transmit time slots as 520 voice devices will require 520 transmit time slots and 520 RTP devices will require 520 transmit time slots. Thus, 2048 minus 520 twice will leave 1008 transmit time slots for conference. As each conference party requires two transmit time slots per party, the Dialogic Stack will allow only 504 parties (1008/2) and the addition of 505th party will fail. Even when the license allows up to 520 conference parties, the combination of voice, RTP and conference party devices limits the number of parties to 504.

```
License of 510r510v0e510c0s0f0i0m_host.lic namely 510 G.711, 510 Voice and 510 Conference
Parties
```

At board download, all voice and RTP channels together will consume 1020 transmit time slots as 510 voice devices will require 510 transmit time slots and 510 RTP devices will require 510 transmit time slots. Thus, 2048 minus 510 twice will leave 1028 transmit time slots for conference. As each conference party requires two transmit time slots per party, the Dialogic Stack will allow 514 parties (1028/2). In this scenario, the maximum number of conference parties specified in the license (510 parties) can be achieved.

5.4 Conference Bridging

If a conference expands beyond the number of parties permitted by the Dialogic® HMP Software license in use and, if applicable, the media load in use on the board, you can create a second conference to support additional conferees. The two conferences are connected via a conference bridge. Conference bridging allows all parties in two or more conferences to speak with and/or listen to one another.

The following guidelines for creating a conference bridge assume that you have already created two conferences and added the desired number of parties for each conference using the instructions in Section 5.3, “Creating a Conference”, on page 22.

- Dedicate a party (party1) in conference A to serve as the bridge to conference B. Likewise, dedicate a party (party2) in conference B to serve as the bridge to conference A.
- Connect party1 in conference A to party2 in conference B using **dev_Connect( )**, a function in the Dialogic® Device Management API library. See the *Dialogic® Device Management API Library Reference* for details on device management functions.

The following rules apply to conference bridging:

- Each bridge that is created consumes two licensed party resources, one from each of the conferences involved in the bridge.

- Even though two (or more) conferences can be bridged together, the attributes and settings of each conference remain unchanged. The application is responsible for managing each conference and conference related events separately.
- The coach/pupil feature does not span conference bridges. Coach and pupil must be in the same conference.

5.5 Terminating an Application

Party resources and conference resources are not released when an application terminates. The conferencing software is designed in this way to allow conferences to stay active when a process exits. Therefore, you are responsible for terminating the application properly. Similarly, if an error condition abnormally terminates the application, individual conferences will not be closed nor will individual channels be closed. In this case, design the application to recover and manage the existing conferences or to shut down devices in an orderly fashion.

When your process completes, devices should be shut down in an orderly fashion. Tasks that are performed to terminate an application generally include:

- disabling events by calling **cnf_DisableEvents( )**
- closing all devices using the appropriate function such as **cnf_CloseParty( )**, **cnf_CloseConference( )**, **cnf_Close( )**, **dx_close( )**, and so on
- breaking the connection between the party device and other supported device using **dev_Disconnect( )**

Note: Standard Runtime Library event management functions (such as **sr_dishdlr( )**, which disables an event handler) must be called before closing the device that is sending the handler event notifications. See Chapter 3, "Event Handling" for more information about handling events.

5.6 Data Structure Considerations

Take note of the following consideration when working with data structures:

- Each data structure in the conferencing library has a version number field. This version number is used to ensure that an application is binary compatible with future changes to this data structure. This field is currently reserved for future use. Use the version number as specified in the header file, *cnflib.h*, and as documented in the *Dialogic® Conferencing API Library Reference*.

5.7 Multiprocessing Considerations

Having multiple processes acting on the same board is undesirable. It is recommended to use a single process per board, or a single process for all boards, rather than more than one process acting on the same board. Consider the scenario where there are multiple boards in the system and each board is being controlled by a different process.

The following considerations apply when multiple processes control the same board:

- You must provide your own synchronization to manage resources in each process.
- If process A creates a conference and process B wants to use that conference, process A must pass the name of the conference to process B.
- If process A deletes a conference and process B has a handle to that conference, then process B can no longer use that conference. Process A must notify process B of its action.

5.8 Multithreading Considerations

The following considerations apply to multithreading:

- The conferencing library supports multithreading. You can manage multiple conferences or multiple boards within the same thread; however, it is not recommended that you manage the same conference or the same board across multiple threads.
- The resource counts returned by **cnf_GetDeviceCount( )** are a snapshot in time. If another thread is adding/deleting a party or creating/deleting a conference, the counts will change and the thread will no longer have the most current count. There is a gap between the time you issue this function and when you actually use the resources. Be sure that threads use synchronization when making decisions based on the counts returned by **cnf_GetDeviceCount( )**.
- While the API functions allow for concurrent use of party, conference and board handles, you must be aware of "logical" concurrency issues, such as maintaining the count of resources.

  The **cnf_GetDeviceCount( )** function returns a snapshot of available parties and maximum parties that can be added to a conference. Because it is a snapshot of the state of the firmware at any given time, the values returned are only valid until other parties and conferences are added or removed.

  In a multithreaded application, you should maintain local counts that are obtained when the application initializes (through **cnf_GetDeviceCount( )**) and protect those counts with mutexes as needed; for example, if two or more threads in the application need to make decisions based on the number of parties and conferences available at any given time. By doing so, race conditions can be avoided; for example, if a thread thinks one more party resource is available while another thread consumes it.

5.9 Volume Control

A party in a conference may wish to change the volume level of the received signal. This is accomplished using the volume control feature.

The **cnf_SetDTMFControl( )** function allows the application to define the DTMF digits that cause the volume level to be adjusted up, down, or back to the default. This function points to the CNF_DTMF_CONTROL_INFO structure which specifies whether volume control is enabled or not and contains details on the digits used for volume control. Volume control is enabled on a board basis.

The **cnf_GetDTMFControl( )** function returns information on the DTMF digits used to control the volume.

The default volume or origin is 0 dB. Volume is incremented or decremented by 2 dB at a time. The maximum value for the volume is 18 dB and the minimum value is -18 dB.

5.10 Active Talker

An active talker refers to a party in a conference who is providing "non-silence" energy. Active talkers are determined by the loudness or strength of their "non-silence" energy. The active talker feature sums the three most active talkers in a conference, so that the conversation doesn't get drowned out when too many people talk at once. The active talker feature also provides data on active talkers through the **cnf_GetActiveTalkerList( )** function.

The active talker feature is enabled on a board basis. To turn on the active talker feature, use **cnf_SetAttributes( )** with the ECNF_BRD_ATTR_ACTIVE_TALKER enumeration enabled. To retrieve a list of active talkers, use **cnf_GetActiveTalkerList( )**.

Note: The active talker feature does not span conference bridges; that is, there is no active talker summing across conference bridges and active talkers are reported separately for each conference.

The **cnf_GetActiveTalkerList( )** function provides a snapshot of the active talkers at a given moment. By default, the snapshot is updated every second. To change this value and specify how frequently the active talker status is updated, use the **cnf_SetAttributes( )** function with the ECNF_BRD_ATTR_NOTIFY_INTERVAL enumeration and specify a value in 10 msec units. If a low value is used, it can affect system performance due to the more frequent updating of the status (which results in a high quantity of internal notification messages). If a high value is used, it will result in less frequent updating on active talkers, but the non-silence energy by a conferee may not be reported if it occurs between notification updates. For example, if the notification interval is set to 2 seconds and a conferee only says "yes" or "no" quickly in between notifications, that vocalization by the conferee will not be reported.

5.11 Privilege Talker

Using the privilege talker feature, the application can explicitly delegate which conference participants are always included in the conference summation output process.

Without the privilege talker feature, Dialogic® HMP Software determines the conference parties included in each summation cycle based on their speech level, resulting in the "loudest" active talkers for a given sample cycle being selected for conference summation. Now, when the Privilege Party attribute is assigned to a conference party, that party's input, providing its speech level is

greater than zero, is always included in the output summation process along with the loudest remaining Normal/Pupil parties within the active talker limit defined for the conference.

5.11.1 Implementation

To use the privilege talker feature, enable the ECNF_PARTY_ATTR_PRIVILEGE attribute using **cnf_SetAttributes( )**. This attribute is defined in the ECNF_PARTY_ATTR enumeration.

When the ECNF_PARTY_ATTR_PRIVILEGE is enabled for a conference participant, that party is summed into the conference output as long as its speech level is greater than zero, the loudest remaining "Normal / Pupil" parties are within the maximum active talker limit defined for the conference. For example, if the active talker limit is six and there are three designated privilege parties, then the conference output would consist of the summation of three privilege parties and three of the loudest Normal/Pupil parties.

When the ECNF_PARTY_ATTR_PRIVILEGE is disabled for a conference participant, that party no longer has precedence in the conference summation, and is not included in the conference's summed output unless his speech level meets the criteria for inclusion as one of loudest talkers.

To set the maximum number of active talkers, use **cnf_SetAttributes( )** and the ECNF_CONF_ATTR_MAX_ACTIVE_TALKERS attribute. This attribute is defined in the ECNF_CONF_ATTR enumeration.

Note: Do not set ECNF_CONF_ATTR_MAX_ACTIVE_TALKERS to less than the current privilege party count for the conference; otherwise, the function returns an error.

You can return the privilege talker attributes using **cnf_GetAttributes( )**.

You can return a list of privilege talkers on a specified conference device using **cnf_GetPrivilegeTalkerList( )**.

5.11.2 Conference Party Types

Normal, Coach, Broadcast, and Pupil are "types" that can be assigned to a conference party. A party can only be assigned one type at the same time. Normal is the default type when adding or creating a party to the conference. The Privilege Party attribute can be enabled or disabled for any of the four types at any time; however, the summation behavior is dependent on the following:

Normal
: When a conference party is type Normal, then the behavior of the system is as described above.

Coach
: When a conference party is type Coach, enabling or disabling the Privilege Party attribute will not affect how the Coach party is summed by the system. That is, the Coach will only be heard by parties of type Pupil and is never summed into the conference output. Its input is post processed and summed into the conference output for Pupils only.

Broadcast

When a conference party is type Broadcast, enabling or disabling the Privilege Party attribute will not affect how the Broadcast party is summed by the system. In this case, only the Broadcast party is heard by all parties.

Pupil

When a conference party is type Pupil, enabling or disabling the Privilege Party attribute will result in Normal party behavior as described above. When the Privilege Party attribute is enabled, the Pupil gets summed. When disabled, the Pupil gets summed in accordance with the Active Talker Detection algorithm selection process.

5.12 Monitor Mode

A conferencing application is able to form audio connections from multiple HMP devices listening half duplex to a single conference party and from multiple conference parties listening half duplex to a single device transmitting on the TDM bus.

Without monitor mode, there is no way to create multiple connections from or to one conference party with the conferencing API. This limited the number of HMP devices "listening" to a conference to the number of conference resources in a license.

With this feature, an application can listen to conference output by retrieving a conference party transmit TDM bus time slot. Multiple HMP devices can then listen to conference output in half-duplex mode by using their technology-specific TDM bus listen APIs (e.g., **ipm_listen( )**). This feature also adds the ability for a conference party to listen in half-duplex mode to any TDM time slot from traditional devices.

5.12.1 Implementation

To implement this feature, use **cnf_Listen( )**, **cnf_UnListen( )** and **cnf_GetXmitSlot( )**. The data structure, SC_TSINFO, is used with the functions to provide time slot information.

Note: It is recommended that the two types of connection methods, **dev_Connect( )/dev_Disconnect( )** and **cnf_listen( )/cnf_unlisten( )**, not be used simultaneously. If they are, then the application must take extreme caution to insure that the connections are properly managed.

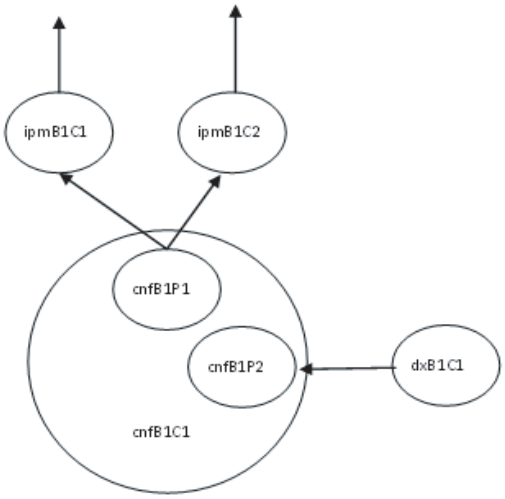
5.12.2 Use Cases

This section provides two possible monitor mode use cases for reference.

Multiple IPM devices listen to a single conference party

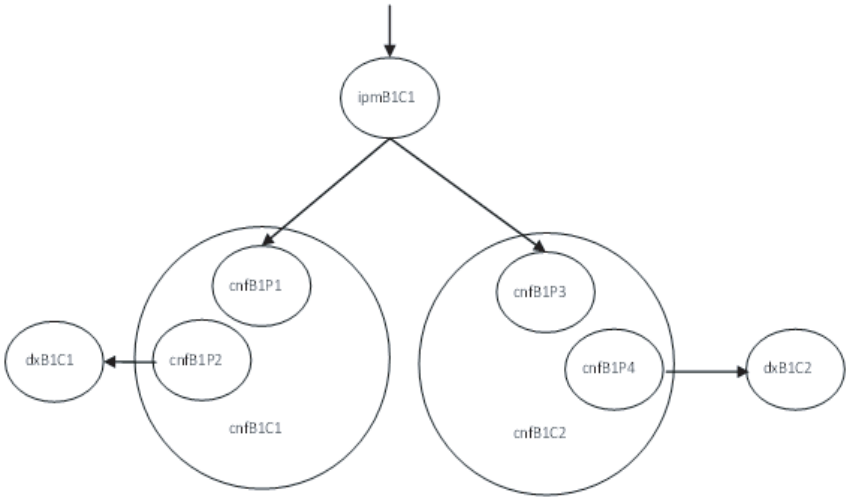
In this use case, multiple externally facing devices, such as IPM devices, make half-duplex listening connections to a conference party. Here, the summed media from the conference can be

broadcast to one or more external connections by using only one conference party. In the illustration below, the arrows designate the direction of the data.



Multiple conference parties listen to a single IPM device

In this use case, multiple conference parties in different conferences make half-duplex connections to a single device such as an IPM device. Here, the single inbound media stream from the external connection can be summed into two different conferences.



5.13 Mute Audio

The application has the ability to mute and un-mute parties of a multimedia conference so that the muted parties will not be heard by other members of the conference.

This feature provides a programmatic interface allowing the application to mute and un-mute the audio stream received from a conference party. This feature applies when using the MCX board device only.

Note: This feature does not support a CNF board device at this time.

5.13.1 Implementation

Use **cnf_SetAttributes( )** and the party attribute, ECNF_PARTY_ATTR_MUTE, to mute or un-mute a conference party.

The behavior of the ECNF_PARTY_ATTR_MUTE party attribute is as follows:

- When the party attribute is set to ECNF_ATTR_STATE_ENABLED for party X, party X is no longer heard by conference participants.
- When a muted party is un-muted by setting the party attribute for that party to ECNF_ATTR_STATE_DISABLED, that party is once again eligible to be summed and heard by other participants of the conference. However:
  - If the un-muted party is a coach, the coach is capable of being heard immediately by the pupil.
  - If the un-muted party is a pupil, the pupil is eligible to be summed and heard by other conference participants. The un-muted party needs to be qualified once again by the active talker algorithm as one of ‘n’ active talkers to be summed.
  - If the un-muted party is a Privileged Party, that party, when speaking, is capable of being heard immediately by other conference participants.

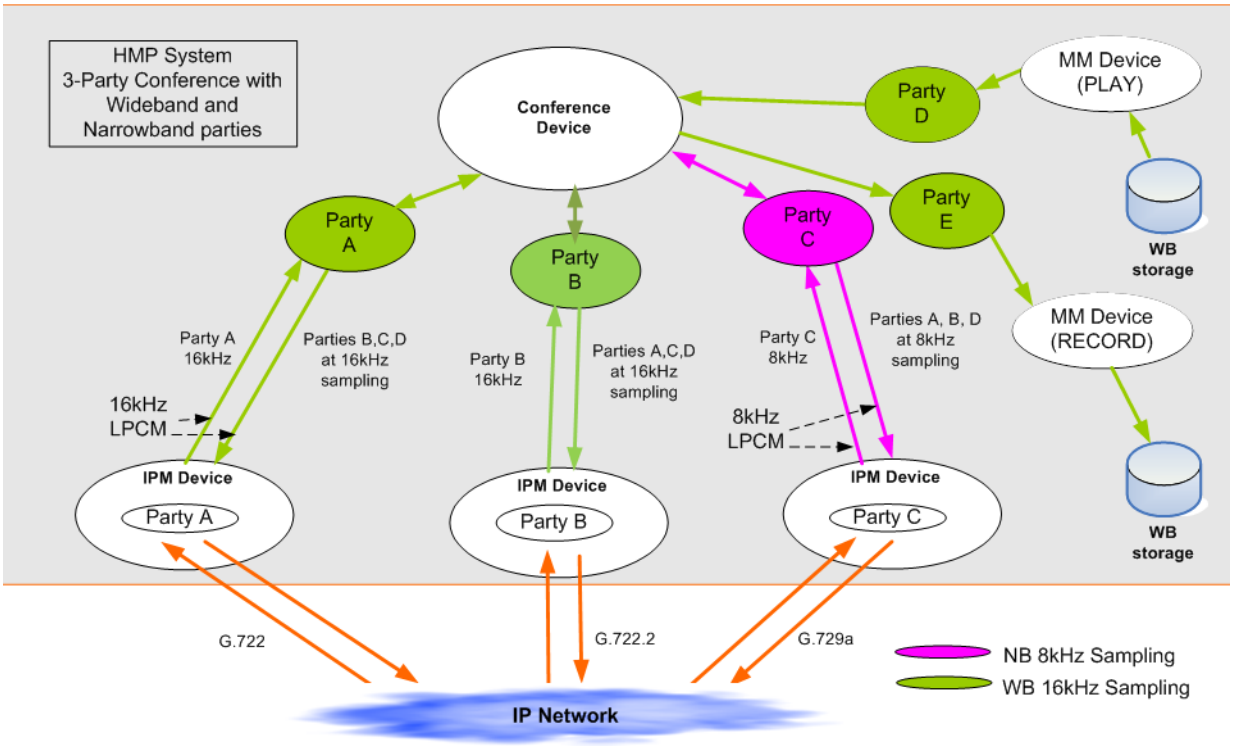
  All other un-muted full-duplex conference parties are eligible to be summed and heard by other conference participants.

5.14 HD Voice Conferencing (Wideband Audio Conferencing)

The conferencing library supports high definition (HD) voice conferencing, also called wideband audio conferencing, using G.722 and G.722.2 (AMR-WB) audio codecs. The conference can consist of a combination of wideband and narrowband parties. In order to achieve wideband fidelity audio, you must use wideband audio codecs for HD voice conferencing.

Conferences can include both wideband and narrowband parties. Wideband parties will hear other wideband conferencing parties with high fidelity audio quality while narrowband parties will be heard at reduced narrowband quality. Narrowband parties will hear all parties at the reduced narrowband quality.

Wideband conferencing is illustrated in the following figure where a multi-party conference consists of party A using G.722, party B using G.722.2, and party C using G.729a coding. The figure shows the flow of how the output of the conference is being recorded for training and review purposes at a later time. Conferences can also play stored wideband audio content into the conference (for example, background music).



5.14.1 Implementation

An application uses the conferencing library with an MCX resource device to implement the HD voice conferencing feature. No conferencing application changes are needed to use this feature beyond using the MCX resource device with wideband device management connection methods.

The following conferencing features are supported:

- AGC (Automatic Gain Control)
- ATD (Active Talker Detection)
- Volume control
- Tone Detection
- Tone Clamping / Tone Distribution
- Coach/pupil
- Conference bridging
- Tariff tone

5.14.2 Licensing Requirements

Wideband conferencing requires the HD Voice Conferencing license as well as HD audio coder licenses (G.722 and G.722.2) for wideband parties in the conference. If you want to use video in an HD conference, you would substitute the Video Conferencing license for the HD Voice Conferencing license.

The following table shows the set of conferencing licenses (along with the resource required) and capabilities that are currently supported.

License (and Resource)	Audio NB	Audio WB	Video
Conferencing (CNF)	Yes	No	No
HD Voice Conferencing (MCX)	Yes	Yes	No
Video Conferencing (MCX)	Yes	Yes	Yes

5.14.3 Conferencing Licenses Exceeded Events

The ECNF_CONF_EVT_EXCEEDED_CONF_LICENSE event notifies the host when conferencing licenses have been exceeded. To enable or disable this event, use **cnf_EnableEvents( )** and **cnf_DisableEvents( )**.

Note: The ECNF_CONF_EVT_EXCEEDED_CONF_LICENSE event is valid for MCX devices only.

The CNFEV_EXCEEDED_CONF_LICENSE notification event notifies the application when the maximum Conference licenses have been exceeded. To enable this event, use **cnf_EnableEvents( )**. When using wideband voice conferencing, the parties can be added to a conference without initially consuming a Conferencing license. This event notifies the application if Conferencing licenses are exhausted as it starts the media on the parties in the conference. The daata type is CNF_EXCEEDED_CONF_LICENSE_EVENT_INFO.

Building Applications 6

This chapter provides information on building applications using the Dialogic® Conferencing (CNF) API library. The following topics are discussed:

6.1 Compiling and Linking

The following topics discuss compiling and linking requirements:

- Include Files
- Required Libraries

6.1.1 Include Files

Function prototypes and symbolic defines are determined in include files, also known as header files. Applications that use Dialogic® Conferencing (CNF) API library functions must contain statements for include files in this form, where <filename> represents the include file name:

```
#include <filename.h>
```

The following header files must be included in the application code **in the order shown** prior to calling the Dialogic® Conferencing (CNF) API library functions:

srllib.h
Contains function prototypes and equates for the Dialogic® Standard Runtime Library.

Note: *srllib.h* must be included in code before all other Dialogic® header files.

cnflib.h
The primary header file for the Dialogic® Conferencing (CNF) API library. Contains function prototypes and symbolic defines.

cnferrs.h
Contains equates for conferencing error codes.

cnfevts.h
Contains equates for conferencing event codes.

devmgmt.h
Contains function prototypes and symbolic defines for the Dialogic® Device Management API library.

If you use other library functions such as voice or IP media, you will have to include the header files for that library:

dxxxlib.h
Contains function prototypes and symbolic defines for the Dialogic® Voice API library.

dtilib.h
Contains function prototypes and symbolic defines for the Dialogic® Digital Network Interface API library.

gclib.h
The primary header file for the Dialogic® Global Call API library; contains function prototypes and symbolic defines for this library.

ipmerror.h
Contains variables for Dialogic® IP Media Library API error codes.

ipmlib.h
Contains function prototypes and symbolic defines for the Dialogic® IP Media Library API.

6.1.2 Required Libraries

Windows®

In Windows®, you must link the following library files when compiling your conferencing application:

libsrlmt.lib
Dialogic® Standard Runtime Library API file. Required in all applications.

libdxxmt.lib
Dialogic® Device Management API library file. Required only if the application uses Dialogic® Voice API library functions directly; for example, **dx_open( )**.

libdtimt.lib
Dialogic® Digital Network Interface API library file. Required only if the application uses Dialogic® Digital Network Interface API library functions directly; for example, **dt_open( )**.

libgc.lib
the primary Dialogic® Global Call APIlibrary file. Required only if the application uses Dialogic® Global Call API library functions directly; for example, **gc_GetResourceH( )**.

libipm.lib
the primary Dialogic® IP Media Library API file. Required only if the application uses Dialogic® IP Media Library API functions directly; for example, **ipm_Open( )**.

libdevmgmt.lib
Dialogic® Device Management API library file. Required in a conferencing application.

libcnf.lib
Dialogic® Conferencing (CNF) API library file. Required in a conferencing application.

Linux

In Linux, you must link the following library files in the order shown when compiling your conferencing application:

libsrl.so
Dialogic® Standard Runtime Library API file. Required in all applications. Specify `-lsrl` in makefile.

libdxxx.so
the primary Dialogic® Voice API library file. Required only if the application uses Dialogic® Voice API library functions directly; for example, **dx_open( )**. Specify `-ldxxx` in makefile.

libdti.so
Dialogic® Digital Network Interface API library file. Required only if the application uses Dialogic® Digital Network Interface API library functions directly; for example, **dt_open( )**. Specify `-ldti` in makefile.

libgc.so
the primary Dialogic® Global Call APIlibrary file. Required only if the application uses Dialogic® Global Call API library functions directly; for example, **gc_GetResourceH( )**. Specify `-lgc` in makefile.

libipm.so
the primary Dialogic® IP Media Library API file. Required only if the application uses Dialogic® IP Media Library API functions directly; for example, **ipm_Open( )**. Specify `-lipm` in makefile.

libdevmgmt.so
Dialogic® Device Management API library file. Required in a conferencing application. Specify `-ldevmgmt` in makefile.

libcnf.so
Dialogic® Conferencing (CNF) API library file. Required in a conferencing application. Specify `-lcnf` in makefile.

By default, the library files are located in the directory given by the INTEL_DIALOGIC_LIB environment variable.

Note: When compiling an application, you must list Dialogic libraries before all other libraries such as operating system libraries.

6.2 Variables for Compiling and Linking

The following variables provide a standardized way of referencing the directories that contain header files and shared objects:

INTEL_DIALOGIC_INC
Variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
Variable that points to the directory where shared library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands. The following is an example of a compiling and linking command that uses these variables:

```
cc -I${INTEL_DIALOGIC_INC} -o myapp myapp.c -L${INTEL_DIALOGIC_LIB} -lcnf -srl
```

Note: It is strongly recommended that you use these variables when compiling and linking applications. The name of the variables will remain constant, but the values may change in future releases.

Function Summary by Category 7

This chapter describes the categories into which the Dialogic® Conferencing (CNF) API library functions can be logically grouped. The topics in this chapter are:

7.1 Device Management Functions

Device management functions allow you to open and close devices. There are three types of devices: board device, conference device, and party device. The board device is the parent device for both the conference and party devices. Thus, you must open a board device before you can open a conference device or a party device.

cnf_Close( )
closes a board device

cnf_CloseConference( )
closes a conference device

cnf_CloseParty( )
closes a party device

cnf_Open( )
opens a board device

cnf_OpenConference( )
opens a conference device

cnf_OpenEx( )
opens a virtual board device in synchronous or asynchronous mode

cnf_OpenParty( )
opens a party device

cnf_ResetDevices( )
resets all open devices that were improperly closed

7.2 Conference Management Functions

Conference management functions allow you add and remove parties to a conference.

cnf_AddParty( )
adds one or more parties to a conference

cnf_RemoveParty( )
removes one or more parties from a conference

7.3 Configuration Functions

Configuration functions allow you to alter, examine, and control the configuration of an open device.

cnf_DisableEvents( )
disables one or more events

cnf_EnableEvents( )
enables one or more events

cnf_GetAttributes( )
gets one or more device attributes

cnf_GetDTMFControl( )
gets DTMF digits control information

cnf_SetAttributes( )
sets one or more device attributes

cnf_SetDTMFControl( )
sets DTMF digits control information

7.4 Auxiliary Functions

Auxiliary functions provide supplementary functionality to help you manage conferences and resources:

cnf_GetActiveTalkerList( )
gets a list of active talkers on a board or in a conference

cnf_GetDeviceCount( )
gets conference and party count information

cnf_GetPartyList( )
gets a list of added parties in a conference

cnf_GetPrivilegeTalkerList( )
gets a list of privilege talkers in a conference

7.5 Multimedia Conferencing Functions

Multimedia conferencing functions manage the multimedia conferencing features:

cnf_GetVideoLayout( )
gets the current video layout

cnf_GetVisiblePartyList( )
gets the visible party list

cnf_SetVideoLayout( )
sets the video layout

cnf_SetVisiblePartyList( )
sets the visible party list

7.6 TDM Routing Functions

TDM routing functions are used in TDM bus configurations.

cnf_GetXmitSlot( )
gets the TDM bus slot number of a party transmit channel

cnf_Listen( )
connects a party receive channel to a TDM bus time slot

cnf_UnListen( )
disconnects a party receive channel from the TDM bus

7.7 Error Processing Function

The error processing function provides error information:

cnf_GetErrorInfo( )
gets error information for a failed function

Function Information 8

This chapter contains a detailed description of each Dialogic® Conferencing (CNF) API function, presented in alphabetical order. A general description of the function syntax is given before the detailed function information.

All function prototypes are in the *cnflib.h* header file.

8.1 Function Syntax Conventions

The conferencing functions typically use the following format:

```
datatype cnf_Function (deviceHandle, parameter1, parameter2, ... parametern)
```

where:

datatype
: refers to the data type; for example, CNF_RETURN and SRL_DEVICE_HANDLE (see *cnflib.h* and *srllib.h* for a definition of data types)

cnf_Function
: represents the name of the function

deviceHandle
: refers to an input field representing the type of device handle (board, conference, or party)

parameter1, parameter2, ... parameter*n*
: represent input or output fields

cnf_AddParty( )

Name: CNF_RETURN cnf_AddParty (a_CnfHandle, a_pPtyInfo, a_pUserInfo)

Inputs:
- SRL_DEVICE_HANDLE a_CnfHandle — conference device handle
- CPCNF_PARTY_INFO a_pPtyInfo — pointer to party information structure
- void * a_pUserInfo — pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Conference Management

Mode: asynchronous

Description

The **cnf_AddParty( )** function adds one or more parties to a conference that has already been created. The CNF_PARTY_INFO structure contains a list of party devices to be added.

Parties must be connected to a voice device (dx_) or other supported device (such as ip_), through the **dev_Connect( )** function, before or after being added to a conference in order to have the party actively participate in the conference. See the *Dialogic® Device Management API Library Reference* for more information on the **dev_Connect( )** function.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pPtyInfo	points to a party information structure, CNF_PARTY_INFO, which contains a list of party devices to be added.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_ADD_PARTY
indicates successful completion of the function; that is, a party was added to a conference
Data Type: CNF_PARTY_INFO

CNFEV_ADD_PARTY_FAIL
indicates that the function failed
Data Type: CNF_PARTY_INFO

Cautions

Adding multiple parties to a conference is supported only when using an mcxBx device. If you are using a cnfBx device, this function will fail if more than one party is specified.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_RemoveParty( )**
- **cnf_OpenParty( )**
- **cnf_CloseParty( )**
- **cnf_CloseConference( )**

cnf_Close( )

Name: CNF_RETURN cnf_Close (a_BrdHandle, a_pCloseInfo)

Inputs: SRL_DEVICE_HANDLE a_BrdHandle • SRL handle to the virtual board device

CPCNF_CLOSE_INFO a_pCloseInfo • reserved for future use

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: synchronous

Description

The **cnf_Close( )** function closes a virtual board device that was previously opened using **cnf_Open( )**. This function does not affect any subdevices that were opened using this virtual board device. All conference and party devices opened using this virtual board device will still be valid after the virtual board device has been closed.

Parameter	Description
a_BrdHandle	specifies an SRL handle for a virtual board device obtained from a previous open
a_pCloseInfo	reserved for future use. Set to NULL.

Cautions

- Once a device is closed, a process can no longer act on the given device via the device handle.
- The only process affected by **cnf_Close( )** is the process that called the function.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

- cnf_Open( )

cnf_CloseConference( )

Name: CNF_RETURN cnf_CloseConference (a_CnfHandle, a_pCloseInfo)

Inputs: SRL_DEVICE_HANDLE a_CnfHandle
- conference device handle

CPCNF_CLOSE_CONF_INFO a_pCloseInfo
- reserved for future use

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: synchronous

Description

The **cnf_CloseConference( )** function closes a conference device handle that was previously opened using **cnf_OpenConference( )**. When the conference is closed, all added parties in this conference are indirectly removed. It is up to you to decide whether to close the party devices or add them to another conference.

Parameter	Description
a_CnfHandle	specifies a conference device handle obtained from a previous open
a_pCloseInfo	reserved for future use. Set to NULL.

Cautions

- Once a device is closed, a process can no longer act on the given device via the device handle.
- This function closes the conference device on all processes in which it is being used. It is up to you to synchronize the creation and deletion of conference devices between processes.
- The **a_pCloseInfo** parameter is reserved for future use and must be set to NULL.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_OpenConference( )**
- **cnf_Open( )**

cnf_CloseParty( )

Name: CNF_RETURN cnf_CloseParty (a_PtyHandle, a_pCloseInfo)

Inputs: SRL_DEVICE_HANDLE a_PtyHandle — party device handle

CPCNF_CLOSE_PARTY_INFO a_pCloseInfo — reserved for future use

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: synchronous

Description

The **cnf_CloseParty( )** function closes a party device handle that was previously opened using **cnf_OpenParty( )**. If the party device is currently added to a conference, this function removes it from the conference before closing it. .

Parameter	Description
a_PtyHandle	specifies a party device handle obtained from a previous open
a_pCloseInfo	reserved for future use. Set to NULL.

Cautions

- Once a device is closed, a process can no longer act on the given device via the device handle.
- This function closes the party device on all processes in which it is being used. It is up to you to synchronize the creation and deletion of party devices between processes.
- The **a_pCloseInfo** parameter is reserved for future use and must be set to NULL.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

- **cnf_OpenParty( )**
- **cnf_CloseConference( )**

cnf_DisableEvents( )

Name: CNF_RETURN cnf_DisableEvents (a_DevHandle, a_pEventInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_DevHandle	• device handle
CPCNF_EVENT_INFO a_pEventInfo	• pointer to event information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: asynchronous

Description

The **cnf_DisableEvents( )** function disables one or more notification events that were previously enabled using **cnf_EnableEvents( )**. The function only applies to the process in which it was called.

Parameter	Description
a_DevHandle	specifies a device handle on which to disable events
a_pEventInfo	points to the event information structure, CNF_EVENT_INFO, which stores information about events to be enabled or disabled.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Events for a board device are defined in the ECNF_BRD_EVT data type; events for a conference device are defined in the ECNF_CONF_EVT data type. Events are disabled by default.

The ECNF_BRD_EVT data type is an enumeration that defines the following values:

ECNF_BRD_EVT_ACTIVE_TALKER
board level notification event for active talker

ECNF_BRD_EVT_CONF_CLOSED
board level notification event for conference closed

ECNF_BRD_EVT_CONF_OPENED
board level notification event for conference opened

ECNF_BRD_EVT_PARTY_ADDED
board level notification event for party added

ECNF_BRD_EVT_PARTY_REMOVED
board level notification event for party removed

The ECNF_CONF_EVT data type is an enumeration that defines the following values:

ECNF_CONF_EVT_ACTIVE_TALKER
conference level notification event for active talker

ECNF_CONF_EVT_DTMF_DETECTION
conference level notification event for DTMF detected

ECNF_CONF_EVT_EXCEEDED_CONF_LICENSE
conference level notification event for conference licenses exceeded

ECNF_CONF_EVT_PARTY_ADDED
conference level notification event for party added

ECNF_CONF_EVT_PARTY_REMOVED
conference level notification event for party removed

Note: The ECNF_BRD_EVT_ACTIVE_TALKER event type is only supported on a CNF board device and not supported on an MCX board device.

For more information on events, see Chapter 9, "Events".

Termination Events

CNFEV_DISABLE_EVENT
indicates successful completion of this function; that is, one or more events were disabled
Data Type: CNF_EVENT_INFO

CNFEV_DISABLE_EVENT_FAIL
indicates that the function failed
Data Type: CNF_EVENT_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_EVENT
invalid device event

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_EnableEvents( )**

cnf_EnableEvents( )

Name: CNF_RETURN cnf_EnableEvents (a_DevHandle, a_pEventInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_DevHandle	• device handle
CPCNF_EVENT_INFO a_pEventInfo	• pointer to event information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: asynchronous

Description

The **cnf_EnableEvents( )** function enables one or more notification events in the process in which it is called. Notification events can only be enabled on a board or on a conference; they cannot be enabled for a party. Notification events are disabled by default.

Notification events are different from asynchronous function termination events, such as CNFEV_OPEN, which cannot be disabled.

Parameter	Description
a_DevHandle	specifies a device handle on which to enable events
a_pEventInfo	points to the event information structure, CNF_EVENT_INFO, which stores information about events to be enabled or disabled.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Events for a board device are defined in the ECNF_BRD_EVT data type; events for a conference device are defined in the ECNF_CONF_EVT data type. Events are disabled by default.

The ECNF_BRD_EVT data type is an enumeration that defines the following values:

ECNF_BRD_EVT_ACTIVE_TALKER
board level notification event for active talker

ECNF_BRD_EVT_CONF_CLOSED
board level notification event for conference closed

ECNF_BRD_EVT_CONF_OPENED
board level notification event for conference opened

ECNF_BRD_EVT_PARTY_ADDED
board level notification event for party added

ECNF_BRD_EVT_PARTY_REMOVED
board level notification event for party removed

The ECNF_CONF_EVT data type is an enumeration that defines the following values:

ECNF_CONF_EVT_ACTIVE_TALKER
conference level notification event for active talker

ECNF_CONF_EVT_DTMF_DETECTION
conference level notification event for DTMF detected

ECNF_CONF_EVT_EXCEEDED_CONF_LICENSE
conference level notification event for conference licenses exceeded

ECNF_CONF_EVT_PARTY_ADDED
conference level notification event for party added

ECNF_CONF_EVT_PARTY_REMOVED
conference level notification event for party removed

Note: The ECNF_BRD_EVT_ACTIVE_TALKER event type is only supported on a CNF board device and not supported on an MCX board device.

For more information on events, see Chapter 9, "Events".

Termination Events

CNFEV_ENABLE_EVENT
indicates successful completion of this function; that is, one or more events were enabled
Data Type: CNF_EVENT_INFO

CNFEV_ENABLE_EVENT_FAIL
indicates that the function failed
Data Type: CNF_EVENT_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_EVENT
invalid device event

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_DisableEvents( )**

cnf_GetActiveTalkerList( )

Name: CNF_RETURN cnf_GetActiveTalkerList (a_DevHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_DevHandle
- device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Auxiliary

Mode: asynchronous

Description

The **cnf_GetActiveTalker( )** function returns a list of active talkers on the specified device. A device can be a board or a conference.

Although this function takes both board and conference device handles, board device handles are only supported on a CNF board device and not on an MCX board device. Conference device handles are supported on both MCX and CNF conference devices. For a board device, all active talkers for that board are returned regardless of the conference to which they belong. For a conference device, only active talkers within that specific conference are returned.

Parameter	Description
a_DevHandle	specifies the device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_ACTIVE_TALKER
indicates successful completion of this function; that is, list of active talkers returned
Data Type: CNF_ACTIVE_TALKER_INFO

CNFEV_GET_ACTIVE_TALKER_FAIL
indicates that the function failed
Data Type: CNF_ACTIVE_TALKER_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

None.

cnf_GetAttributes( )

Name: CNF_RETURN cnf_GetAttributes (a_DevHandle, a_pAttrInfo, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_DevHandle
- device on which to get attributes

CPCNF_ATTR_INFO a_pAttrInfo
- pointer to attribute information structure

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: asynchronous

Description

The **cnf_GetAttributes( )** function gets the values of one or more device attributes. A device can be a board, a conference, or a party. The values for the attributes are returned in a structure provided in the CNFEV_GET_ATTRIBUTE event.

Parameter	Description
a_DevHandle	specifies the device handle on which to get attributes
a_pAttrInfo	points to the attribute information structure, CNF_ATTR_INFO. This structure in turn points to the CNF_ATTR structure, which specifies an attribute and its value.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Attributes for each type of device are defined in the ECNF_BRD_ATTR, ECNF_CONF_ATTR, and ECNF_PARTY_ATTR enumerations.

The ECNF_BRD_ATTR data type is an enumeration that defines the following values:

ECNF_BRD_ATTR_ACTIVE_TALKER
enables or disables board level active talker.

ECNF_BRD_ATTR_NOTIFY_INTERVAL
changes the default firmware interval for active talker notification events on the board. The value must be passed in 10 msec units. The default setting is 100 (1 second).

ECNF_BRD_ATTR_TONE_CLAMPING
enables or disables board level tone clamping to reduce the level of DTMF tones heard on a per party basis on the board.

The ECNF_CONF_ATTR data type is an enumeration that defines the following values:

ECNF_CONF_ATTR_DTMF_MASK
specifies a mask for the DTMF digits used for volume control. The digits are defined in the ECNF_DTMF_DIGIT enumeration. The ECNF_DTMF_DIGIT values can be ORed to form the mask using the ECNF_DTMF_MASK_OPERATION enumeration. For a list of ECNF_DTMF_DIGIT values, see the description for CNF_DTMF_CONTROL_INFO.

ECNF_CONF_ATTR_MAX_ACTIVE_TALKERS
sets maximum active talkers. Possible values are 2 to 10 (default).

ECNF_CONF_ATTR_NOTIFY
enables or disables conference notification tone.

ECNF_CONF_ATTR_TONE_CLAMPING
enables or disables conference level tone clamping. Overrides board level value.

The ECNF_PARTY_ATTR data type is an enumeration that defines the following values:

ECNF_PARTY_ATTR_AGC
enables or disables automatic gain control.

ECNF_PARTY_ATTR_BROADCAST
enables or disables broadcast mode. One party can speak while all other parties are muted.

ECNF_PARTY_ATTR_COACH
sets party to coach. Coach is heard by pupil only.

ECNF_PARTY_ATTR_ECHO_CANCEL
enables or disables echo cancellation. Provides 128 taps (16 msec) of echo cancellation.

ECNF_PARTY_ATTR_MUTE
mutes or un-mutes the audio stream received from a conference party.

ECNF_PARTY_ATTR_PRIVILEGE
enables or disables privilege talker. When enabled, party is always included in the conference summation output process, providing its speech level is greater than zero.

ECNF_PARTY_ATTR_PUPIL
sets party to pupil. Pupil hears everyone including the coach.

ECNF_PARTY_ATTR_TARIFF_TONE
enables or disables tariff tone. Party receives periodic tone for duration of the call.

ECNF_PARTY_ATTR_TONE_CLAMPING
enables or disables DTMF tone clamping for the party. Overrides board and conference level values.

Termination Events

CNFEV_GET_ATTRIBUTE
indicates successful completion of this function; that is, attribute values were returned

Data Type: CNF_ATTR_INFO

CNFEV_GET_ATTRIBUTE_FAIL
indicates that the function failed

Data Type: CNF_ATTR_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_ATTR
: invalid attribute

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

- **cnf_SetAttributes( )**

cnf_GetDeviceCount( )

Name: CNF_RETURN cnf_GetDeviceCount (a_BrdHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_BrdHandle
- board device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Auxiliary

Mode: asynchronous

Description

The **cnf_GetDeviceCount( )** function returns the number of conference and party devices available on the specified virtual board device. See the CNF_DEVICE_COUNT_INFO structure for more on the type of information returned.

Parameter	Description
a_BrdHandle	specifies the virtual board device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_DEVICE_COUNT
indicates successful completion of this function; that is, device count returned
Data Type: CNF_DEVICE_COUNT_INFO

CNFEV_GET_DEVICE_COUNT_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_AddParty( )**
- **cnf_RemoveParty( )**

cnf_GetDTMFControl( )

Name: CNF_RETURN cnf_GetDTMFControl (a_BrdHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_BrdHandle
- SRL handle to the virtual board device

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: asynchronous

Description

The **cnf_GetDTMFControl( )** function returns information about the DTMF digits used to control the conference behavior, such as volume level. The DTMF digit information is stored in the CNF_DTMF_CONTROL_INFO structure.

Parameter	Description
a_BrdHandle	specifies the SRL handle to the virtual board device obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_DTMF_CONTROL
indicates successful completion of this function; that is, DTMF digit information was returned
Data Type: CNF_DTMF_CONTROL_INFO

CNFEV_GET_DTMF_CONTROL_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_SetDTMFControl( )**

cnf_GetErrorInfo( )

Name: CNF_RETURN cnf_GetErrorInfo (a_pErrorInfo)

Inputs: PCNF_ERROR_INFO * a_pErrorInfo • pointer to error information structure

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Error Processing

Mode: synchronous

Description

The **cnf_GetErrorInfo( )** function obtains error information about a failed function and provides it in the CNF_ERROR_INFO structure. To retrieve the information, this function must be called immediately after the Dialogic® Conferencing (CNF) API function failed.

Parameter	Description
a_pErrorInfo	points to the error information structure, CNF_ERROR_INFO

Cautions

- The **cnf_GetErrorInfo( )** function can only be called in the same thread in which the routine that had the error was called. The **cnf_GetErrorInfo( )** function cannot be called to retrieve error information for a function that returned error information in another thread.
- The Dialogic® Conferencing (CNF) API only keeps the error information for the last Dialogic® Conferencing (CNF) API function call. Therefore, you should check and retrieve the error information immediately if a Dialogic® Conferencing (CNF) API function fails.

Errors

Do not call the **cnf_GetErrorInfo( )** function recursively if it returns CNF_ERROR to indicate failure. A failure return generally indicates that the **a_pErrorInfo** parameter is NULL or invalid.

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

None.

cnf_GetPartyList( )

Name: CNF_RETURN cnf_GetPartyList (a_CnfHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_CnfHandle
- conference device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Auxiliary

Mode: asynchronous

Description

The **cnf_GetPartyList( )** function returns a list of party devices currently added to the specified conference.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_PARTY_LIST
indicates successful completion of this function; that is, list of added parties returned
Data Type: CNF_PARTY_INFO

CNFEV_GET_PARTY_LIST_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_AddParty( )**
- **cnf_RemoveParty( )**

cnf_GetPrivilegeTalkerList( )

get privilege talker list

Name: CNF_RETURN cnf_GetPrivilegeTalkerList(a_CnfHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_CnfHandle — conference device handle

void * a_pUserInfo — pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Auxiliary

Mode: asynchronous

Description

The **cnf_GetPrivilegeTalkerList( )** function returns a list of privilege talkers on a specified conference device.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_PRIVILEGE_TALKER
indicates successful completion of this function; that is, list of privilege talkers returned
Data Type: CNF_PRIVILEGE_TALKER_INFO

CNFEV_GET_PRIVILEGE_TALKER_FAIL
indicates that the function failed
Data Type: CNF_PRIVILEGE_TALKER_INFO_FAIL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for general example code.

See Also

None.

cnf_GetVideoLayout( )

Name: CNF_RETURN cnf_GetVideoLayout(a_CnfHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_CnfHandle
- conference device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Multimedia Conferencing

Mode: asynchronous

Description

The **cnf_GetVideoLayout( )** function gets the video layout on the specified conference device. The video layout handle will be provided in the termination event. Please refer to the Dialogic® Media Toolkit Library Reference for more information on the layout builder functions. These functions can be used to access the video layout information using the handle returned. Only CUSTOM layout types are modifiable. Once received, the video layout handle can be modified using the **cnf_SetVideoLayout( )** function, which allows the user to configure the layout prior to setting it on the conference device.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_VIDEO_LAYOUT
indicates successful completion of this function; that is, video layout returned
Data Type: CNF_VIDEO_LAYOUT_INFO

CNFEV_GET_VIDEO_LAYOUT_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_SetVideoLayout( )**

cnf_GetVisiblePartyList( )

get visible party list

Name: CNF_RETURN cnf_GetVisiblePartyList(a_CnfHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_CnfHandle
- conference device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Multimedia Conferencing

Mode: asynchronous

Description

The **cnf_GetVisiblePartyList( )** function returns the current visible party list on a specified conference device.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_GET_VISIBLE_PARTY_LIST
indicates successful completion of this function; that is, list of added parties returned
Data Type: CNF_VISIBLE_PARTY_LIST

CNFEV_GET_VISIBLE_PARTY_LIST_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_SetVisiblePartyList( )**

cnf_GetXmitSlot( )

get TDM bus time slot number of party transmit channel

Name: int cnf_GetXmitSlot(a_PtyHandle, a_pTimeslotInfo)

Inputs: SRL_DEVICE_HANDLE a_PtyHandle
- valid party device handle

SC_TSINFO * a_pTimeslotInfo
- pointer to TDM bus time slot information structure

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: TDM routing

Mode: asynchronous

Description

The **cnf_GetXmitSlot( )** function returns the time division multiplexing (TDM) bus time slot number of the conference party transmit channel. The TDM bus time slot information is contained in an SC_TSINFO structure that includes the number of the TDM bus time slot connected to the conference party transmit channel.

Parameter	Description
a_PtyHandle	specifies a party device handle obtained from a previous open
a_pTimeslotInfo	specifies a pointer to the SC_TSINFO structure

Cautions

This function fails when an invalid SC_TSINFO structure pointer or value(s) is specified. The data structure must be initialized to one for the sc_numts field, and have memory allocated for one long element for the sc_tsarrayp pointer field.

Errors

If the function fails with a CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error.

Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_INVALID_PARM
invalid parameter

ECNF_SUBSYSTEM
internal subsystem error

Example

```
#include <stdlib.h>
#include <stdio.h>

// Dialogic headers
#include "srllib.h"
#include "cnflib.h"

#define MAX_DEVNAME100
#define SRWAITTIMEOUT 10000

long ProcessEvt();

int main(int argc, char* argv[])
{
    char cnfbdname[MAX_DEVNAME] = {"cnfB1"};
    SRL_DEVICE_HANDLE cnfbdh = -1;
    SRL_DEVICE_HANDLE cnfh = -1;
    SRL_DEVICE_HANDLE cnfptyh = -1;
    long ts;
    SC_TSINFO scts;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled (non-signal) mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 )
    {
        printf( "Error: cannot set srl mode\n" );
        exit( 1 );
    }

    cnfbdh = cnf_OpenEx(cnfbdname, NULL, NULL, EV_SYNC);
    if (cnfbdh == -1)
    {
        printf("Error during call to cnf_OpenEx\n");
        /* perform error processing */
        exit(1);
    }

  /* open conferences */
    cnfh = cnf_OpenConference(cnfbdh, NULL, NULL, NULL);
    if (cnfh == -1)
    {
        printf("Error during call to cnf_OpenConference\n");
        /* perform error processing */
        exit(1);
    }

    if(sr_waitevt(SRWAITTIMEOUT) != -1)
    {
        if (!ProcessEvt())
        {
            /* perform error processing */
            exit(1);
        }
    }
    else
    {
        printf("Error during call to sr_waitevt\n");
        /* perform error processing */
        exit(1);
    }

    cnfptyh = cnf_OpenParty(cnfbdh, NULL, NULL, NULL);
    if (cnfptyh == -1)
    {
        printf("Error during call to cnf_OpenParty\n");
```

```
        /* perform error processing */
        exit(1);
    }

    if(sr_waitevt(SRWAITTIMEOUT) != -1)
    {
        if (!ProcessEvt())
        {
            /* perform error processing */
            exit(1);
        }
    }
    else
    {
        printf("Error during call to sr_waitevt\n");
        /* perform error processing */
        exit(1);
    }

    scts.sc_numts = 1;
    scts.sc_tsarrayp = &ts;

    if (cnf_GetXmitSlot(cnfptyh, &scts))
    {
        printf("Error during call to cnf_GetXmitSlot\n");
        /* perform error processing */
        exit(1);
    }

    printf("Party %s (cnfptyh=%ld) is transmitting on %ld\n", ATDV_NAMEP(cnfptyh), cnfptyh, ts);
    return 0;
}

long ProcessEvt()
{
    long ret = 1;
    int devh;
    int evttype;
    long evtlen;
    void* datap;

    printf("ProcessEvt()\n");

    devh = sr_getevtdev();
    evttype = sr_getevttype();
    evtlen = sr_getevtlen();
    datap = sr_getevtdatap();

    switch(evttype)
    {
    case CNFEV_OPEN_CONF:
        printf("Received CNFEV_OPEN_CONF\n");
        break;

    case CNFEV_OPEN_CONF_FAIL:
        printf("Received CNFEV_OPEN_CONF_FAIL\n");
        ret = 0;
        break;

    case CNFEV_OPEN_PARTY:
        printf("Received CNFEV_OPEN_PARTY\n");
        break;

    case CNFEV_OPEN_PARTY_FAIL:
        printf("Received CNFEV_OPEN_PARTY_FAIL\n");
        ret = 0;
        break;
```

```
    default:
        printf("Unhandled event: devh(%d); evttype(0x%x)", devh, evttype);
        break;
    }

    return ret;
}
```

See Also

- cnf_Listen( )
- cnf_UnListen( )

cnf_Open( )

Name: SRL_DEVICE cnf_Open (a_szBrdName, a_pOpenInfo, a_pUserInfo)

Inputs:
- const char * a_szBrdName — pointer to board device name
- CPCNF_OPEN_INFO a_pOpenInfo — reserved for future use
- void * a_pUserInfo — pointer to user-defined data

Returns: board device handle if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: asynchronous

Description

The **cnf_Open( )** function opens an audio only conference (CNF) board device or a multimedia conference (MCX) board device. The naming convention of a CNF board device is "cnfBx" while an MCX board device is "mcxBx" where x is the board number starting from 1. All subsequent references to the opened device must be made using the handle until the device is closed.

All conference and party devices opened using a board handle will open the corresponding type of conference or party device.

Parameter	Description
a_szBrdName	points to a board device name
a_pOpenInfo	reserved for future use. Set to NULL.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_OPEN
indicates successful completion of this function; that is, a virtual board device was opened
Data Type: NULL

CNFEV_OPEN_FAIL
indicates that the function failed
Data Type: NULL

Note: If CNFEV_OPEN_FAIL is received, you must call **cnf_Close( )** to clean up the operation.

Cautions

- Before closing CNF devices, ensure that events are disabled by calling **cnf_DisableEvents( )**; otherwise, the firmware process will stop executing (also known as KILLTASK).

- In applications that spawn child processes from a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.
- The **a_pOpenInfo** parameter is reserved for future use and must be set to NULL.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Possible errors for this function include:

ECNF_INVALID_NAME
: invalid device name

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_Close( )**

cnf_Listen( )

connect party receive channel to TDM bus time slot

Name: int cnf_Listen(a_PtyHandle, a_pTimeslotInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_PtyHandle	• valid party device handle
SC_TSINFO *a_pTimeslotInfo	• pointer to TDM bus time slot information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: TDM routing

Mode: asynchronous

Description

The **cnf_Listen( )** function connects a party receive channel to a TDM bus time slot, using information stored in the SC_TSINFO data structure. The function then sets up a half-duplex connection. For a full-duplex connection, the receive channel of the other device must be connected to the party transmit channel.

The **cnf_Listen( )** function returns immediately before the operation is completed with CNF_SUCCESS, providing argument validation passed, otherwise a CNF_ERROR is returned. After the operation completes, a notification event is received and, if successful, the party receive channel is connected to the TDM bus time slot. Although multiple party channels may listen (be connected) to the same TDM bus time slot, the receive channel of a given party device can connect to only one TDM bus time slot.

Parameter	Description
a_PtyHandle	specifies a party device handle obtained from a previous open
a_pTimeslotInfo	specifies a pointer to the SC_TSINFO data structure
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_LISTEN
indicates successful completion of this function, at which point the party device's receive channel is connected to the TDM bus time slot originally specified in **a_pTimeslotInfo**

CNFEV_LISTEN_FAIL
indicates that the function failed

Cautions

- This function fails when an invalid party handle is specified or when an invalid TDM bus time slot number is specified.
- It is recommended that the two types of connection methods, **dev_Connect( )/dev_Disconnect( )** and **cnf_listen( )/cnf_unlisten( )**, not be used simultaneously. If they are, then the application must take extreme caution to insure that the connections are properly managed.

Errors

The CNF_ERROR_INFO data structure provides error information for the device handle when an API function fails. Upon failure, call **cnf_GetErrorInfo( )** to return error information for the device in the CNF_ERROR_INFO structure. Error codes are returned as: ECNF_xxxxx

Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_INVALID_PARM
invalid parameter

ECNF_SUBSYSTEM
internal subsystem error

The CNF_EVENT_INFO data structure provides event information for the device handle when a notification event is enabled or disabled. This structure is used by and enabled by **cnf_EnableEvents( )**. Use **sr_waitevt( )**, **sr_enbhdlr( )** or other SRL functions to collect an event code, depending on the programming model in use. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*. Error codes are returned as: CNFEV_xxxxxx

Possible errors for this function include:

CNFEV_INVALID_DEVICE
invalid device handle

CNFEV_INVALID_PARM
invalid parameter

CNFEV_SUBSYSTEM
internal subsystem error

Example

```
#include <stdlib.h>
#include <stdio.h>

// Dialogic headers
#include "srllib.h"
#include "dxxxlib.h"
#include "cnflib.h"

#define MAX_DEVNAME100
#define SRWAITTIMEOUT 10000
```

```
long ProcessEvt();

int main(int argc, char* argv[])
{
    char cnfbdname[MAX_DEVNAME] = {"cnfB1"};
    char dxdevname[MAX_DEVNAME] = {"dxxxB1C1"};
    long devh = -1;
    SRL_DEVICE_HANDLE cnfbdh = -1;
    SRL_DEVICE_HANDLE cnfh = -1;
    SRL_DEVICE_HANDLE cnfptyh = -1;
    long ts;
    SC_TSINFO scts;
    int mode = SR_POLLMODE;

    /* Set SRL to run in polled (non-signal) mode */
    if( sr_setparm( SRL_DEVICE, SR_MODEID, &mode ) == -1 )
    {
        printf( "Error: cannot set srl mode\n" );
        exit( 1 );
    }

    cnfbdh = cnf_OpenEx(cnfbdname, NULL, NULL, EV_SYNC);
    if (cnfbdh == -1)
    {
        printf("Error during call to cnf_OpenEx\n");
        /* perform error processing */
        exit(1);
    }

    /* open conferences */
    cnfh = cnf_OpenConference(cnfbdh, NULL, NULL, NULL);
    if (cnfh == -1)
    {
        printf("Error during call to cnf_OpenConference\n");
        /* perform error processing */
        exit(1);
    }

    if(sr_waitevt(SRWAITTIMEOUT) != -1)
    {
        if (!ProcessEvt())
        {
            /* perform error processing */
            exit(1);
        }
    }
    else
    {
        printf("Error during call to sr_waitevt\n");
        /* perform error processing */
        exit(1);
    }

    cnfptyh = cnf_OpenParty(cnfbdh, NULL, NULL, NULL);
    if (cnfptyh == -1)
    {
        printf("Error during call to cnf_OpenParty\n");
        /* perform error processing */
        exit(1);
    }

    if(sr_waitevt(SRWAITTIMEOUT) != -1)
    {
        if (!ProcessEvt())
        {
            /* perform error processing */
            exit(1);
```

```
    }
}
else
{
    printf("Error during call to sr_waitevt\n");
    /* perform error processing */
    exit(1);
}

/* open a voice device */
devh = dx_open(dxdevname, 0);
if (devh == -1)
{
    printf("Error during call to dx_open\n");
    /* perform error processing */
    exit(1);
}

scts.sc_numts = 1;
scts.sc_tsarrayp = &ts;

if (dx_getxmitslot(devh, &scts) == -1)
{
    printf("Error during call to dx_getxmitslot\n");
    /* perform error processing */
    exit(1);
}

printf("Voice device %s (devh=%ld) is transmitting on %ld\n",
       ATDV_NAMEP(devh), devh, ts);

if (cnf_Listen(cnfptyh, &scts, NULL) == -1)
{
    printf("Error during call to cnf_Listen\n");
    /* perform error processing */
    exit(1);
}

printf("Successful call to cnf_Listen\n");

if(sr_waitevt(SRWAITTIMEOUT) != -1)
{
    if (!ProcessEvt())
    {
        /* perform error processing */
        exit(1);
    }
}
else
{
    printf("Error during call to sr_waitevt\n");
    /* perform error processing */
    exit(1);
}

if (cnf_UnListen(cnfptyh, NULL) == -1)
{
    printf("Error during call to cnf_UnListen\n");
    /* perform error processing */
    exit(1);
}

printf("Successful call to cnf_UnListen\n");

if(sr_waitevt(SRWAITTIMEOUT) != -1)
{
    if (!ProcessEvt())
```

```
        {
            /* perform error processing */
            exit(1);
        }
    }
    else
    {
        printf("Error during call to sr_waitevt\n");
        /* perform error processing */
        exit(1);
    }

    return 0;
}

long ProcessEvt()
{
    long ret = 1;
    int devh;
    int evttype;
    long evtlen;
    void* datap;

    printf("ProcessEvt()\n");

    devh = sr_getevtdev();
    evttype = sr_getevttype();
    evtlen = sr_getevtlen();
    datap = sr_getevtdatap();

    switch(evttype)
    {
    case CNFEV_OPEN_CONF:
        printf("Received CNFEV_OPEN_CONF\n");
        break;

    case CNFEV_OPEN_CONF_FAIL:
        printf("Received CNFEV_OPEN_CONF_FAIL\n");
        ret = 0;
        break;

    case CNFEV_OPEN_PARTY:
        printf("Received CNFEV_OPEN_PARTY\n");
        break;

    case CNFEV_OPEN_PARTY_FAIL:
        printf("Received CNFEV_OPEN_PARTY_FAIL\n");
        ret = 0;
        break;

    case CNFEV_LISTEN:
        printf("Received CNFEV_LISTEN\n");
        break;

    case CNFEV_LISTEN_FAIL:
        printf("Received CNFEV_LISTEN_FAIL\n");
        ret = 0;
        break;

    case CNFEV_UNLISTEN:
        printf("Received CNFEV_UNLISTEN\n");
        break;

    case CNFEV_UNLISTEN_FAIL:
        printf("Received CNFEV_UNLISTEN_FAIL\n");
        ret = 0;
        break;
```

```
        default:
            printf("Unhandled event: devh(%d); evttype(0x%x)", devh, evttype);
            break;
        }

        return ret;
}
```

See Also

- **cnf_GetXmitSlot( )**
- **cnf_UnListen( )**

cnf_OpenConference( )

Name: SRL_DEVICE_HANDLE cnf_OpenConference (a_nBrdHandle, a_szCnfName, a_pOpenInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_nBrdHandle	• SRL handle to the virtual board device
const char * a_szCnfName	• pointer to conference name
CPCNF_OPEN_CONF_INFO a_pOpenInfo	• reserved for future use
void * a_pUserInfo	• pointer to user-defined data

Returns: conference device handle if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: asynchronous

Description

The **cnf_OpenConference( )** function opens a new conference device or an existing conference device. The type of conference device opened is determined by the board device handle used to open the device. If a CNF board device is used, a CNF conference device is opened; and if an MCX board device is used, an MCX conference device is opened.

To open a new conference, set the **a_szCnfName** parameter to NULL and specify the virtual board device handle on which to open the new conference. This function opens a conference device and returns a unique SRL handle to identify the device. All subsequent references to the opened device must be made using the handle until the device is closed.

The number of conference devices that can be opened is fixed per virtual board and you may open all conference devices during initialization or dynamically at runtime. To determine the number of conference devices available, use **cnf_GetDeviceCount( )**.

Parameter	Description
a_nBrdHandle	specifies an SRL handle to the virtual board device
a_szConfName	points to an existing conference device. Set to NULL to open a new conference.
a_pOpenInfo	reserved for future use. Set to NULL.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_OPEN_CONF
indicates successful completion of this function; that is, a conference device was opened

Data Type: CNF_OPEN_CONF_RESULT

CNFEV_OPEN_CONF_FAIL
indicates that the function failed

Data Type: CNF_OPEN_CONF_RESULT

Note: If CNFEV_OPEN_CONF_FAIL is received, you must call **cnf_CloseConference( )** to clean up the operation.

Cautions

- Before closing CNF devices, ensure that events are disabled by calling **cnf_DisableEvents( )**; otherwise, the firmware process will stop executing (also known as KILLTASK).
- In applications that spawn child processes from a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.
- The **a_pOpenInfo** parameter is reserved for future use and must be set to NULL.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_INVALID_NAME
invalid device name

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

- **cnf_CloseConference( )**

cnf_OpenEx( )

Name SRL_DEVICE_HANDLE cnf_OpenEx (a_szBrdName, a_pOpenInfo, a_pUserInfo, a_usMode)

Inputs:

const char * a_szBrdName	• pointer to virtual board device name
CPCNF_OPEN_INFO a_pOpenInfo	• reserved for future use
void * a_pUserInfo	• pointer to user-defined data
unsigned short a_usMode	• synchronous/asynchronous mode specifier

Returns: Virtual board SRL device handle if successful
CNF_ERROR on failure

Includes: cnflib.h

Category: Device Management

Mode: synchronous/asynchronous

Description

The **cnf_OpenEx( )** function opens an audio only conference (CNF) board device or a multimedia conference (MCX) board device. The naming convention of a CNF board device is "cnfBx" while an MCX board device is "mcxBx" where x is the board number starting from 1. All subsequent references to the opened device must be made using the handle until the device is closed.

All conference and party devices opened using a board handle will open the corresponding type of conference or party device.

The **cnf_OpenEx( )** function allows you to choose synchronous or asynchronous mode. If you require operation in synchronous mode, use **cnf_OpenEx( )** instead of **cnf_Open( )**.

Parameter	Description
a_szBrdName	points to a virtual board device name
a_pOpenInfo	reserved for future use. Must be set to NULL.
a_pUserInfo	points to user-defined data. If none, set to NULL.
a_usMode	specifies synchronous/asynchronous mode. Valid values are: • EV_SYNC • EV_ASYNC ***Note:*** There is no default setting for mode.

If this function is called in synchronous mode, then if successful, the returned SRL handle is a valid handle that can be used to further communicate with the board device.

If this function is called in the asynchronous mode, then if successful, the returned SRL handle will not be valid until the CNFEV_OPEN event is reported on the SRL handle to indicate successful

initialization of the device. If a failure occurs, the device is not opened and the CNFEV_OPEN_FAIL event will be reported on the SRL handle returned from **cnf_OpenEx( )**.

Termination Events

The following is a list of events that can be returned as a completion to this request when used in asynchronous mode.

CNFEV_OPEN
: indicates successful completion of this function; that is, a virtual board device was opened
Data Type: NULL

CNFEV_OPEN_FAIL
: indicates that the function failed
Data Type: NULL

Note: Application must call **cnf_Close( )** to clean up if CNFEV_OPEN_FAIL is received.

Cautions

- In applications that spawn child processes from a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.
- The **a_pOpenInfo** parameter is reserved for future use and must be set to NULL.
- The same virtual board device can be opened in multiple processes; one process can delete a conference running on another process on the same virtual board device. It is up to you to synchronize access to the same virtual board device from multiple processes.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Refer to **cnf_GetErrorInfo( )** for a list of possible error values.

Example

See Section 12.1, “Conferencing Example Code and Output”, on page 147 for complete example code.

See Also

- **cnf_Close( )**

cnf_OpenParty( )

Name: CNF_RETURN cnf_OpenParty (a_nBrdHandle, a_szPtyName, a_pOpenInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_nBrdHandle	• SRL handle to the virtual board device
const char * a_szPtyName	• pointer to party device name
CPCNF_OPEN_PARTY_INFO a_pOpenInfo	• reserved for future use
void * a_pUserInfo	• pointer to user-defined data

Returns: party device handle if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Device Management

Mode: asynchronous

Description

The **cnf_OpenParty( )** function opens a new party device or an existing party device. The type of party device opened is determined by the board device handle used to open the device. If a CNF board device is used, a CNF party device is opened; and if an MCX board device is used, an MCX party device is opened.

To open a new party, set the **a_szPtyName** parameter to NULL and specify the virtual board device handle on which to open the new party. This function opens a party device and returns a unique SRL handle to identify the device. All subsequent references to the opened device must be made using the handle until the device is closed.

The number of party devices that can be opened is fixed per virtual board and you may open all party devices during initialization or dynamically at runtime. To determine the number of party devices available, use **cnf_GetDeviceCount( )**.

Parameter	Description
a_nBrdHandle	specifies the SRL handle to the virtual board device
a_szPtyName	points to an existing party device. Set to NULL to open a new party.
a_pOpenInfo	reserved for future use. Set to NULL.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_OPEN_PARTY
indicates successful completion of this function; that is, a party device was opened
Data Type: CNF_OPEN_PARTY_RESULT

CNFEV_OPEN_PARTY_FAIL
indicates that the function failed

Data Type: CNF_OPEN_PARTY_RESULT

Note: If CNFEV_OPEN_PARTY_FAIL is received, you must call **cnf_CloseParty( )** to clean up the operation.

Cautions

- In applications that spawn child processes from a parent process, the device handle is not inheritable by the child process. Make sure devices are opened in the child process.
- The **a_pOpenInfo** parameter is reserved for future use and must be set to NULL.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_INVALID_NAME
invalid device name

ECNF_SUBSYSTEM
internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_CloseParty( )**
- **cnf_CloseConference( )**

cnf_RemoveParty( )

Name: CNF_RETURN cnf_RemoveParty (a_CnfHandle, a_pPtyInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_CnfHandle	• conference device handle
CPCNF_PARTY_INFO a_pPtyInfo	• pointer to party information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Conference Management

Mode: asynchronous

Description

The **cnf_RemoveParty( )** function removes one or more parties from a conference. The CNF_PARTY_INFO structure contains a list of party devices to be removed. The removed party or parties can be added to a different conference; or they can be closed.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pPtyInfo	points to a party information structure, CNF_PARTY_INFO
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_REMOVE_PARTY
indicates successful completion of this function; that is, a party device was added
Data Type: CNF_PARTY_INFO

CNFEV_REMOVE_PARTY_FAIL
indicates that the function failed
Data Type: CNF_PARTY_INFO

Cautions

When using a CNF conference device, only one party at a time can be removed from the conference. This function will fail if more than one party is specified. Removing multiple parties from a conference is supported on an MCX conference device.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_AddParty( )**
- **cnf_CloseParty( )**
- **cnf_CloseConference( )**

cnf_ResetDevices( )

Name: CNF_RETURN cnf_ResetDevices(SRL_DEVICE_HANDLE a_BrdHandle, CPCNF_RESET_DEVICES_INFO a_pResetInfo, void *a_pUserInfo)

Inputs:

a_BrdHandle	• SRL handle to the virtual board device
a_pResetInfo	• reserved for future use
a_pUserInfo	• pointer to user defined data

Returns: CNF_SUCCESS for success
CNF_ERROR for failure

Includes: cnflib.h

Category: Device Management

Mode: Asynchronous

Description

The **cnf_ResetDevices( )** function resets all devices that may have been opened and not closed by a previous process for the specified board. This function should only be used to recover conference and party devices that were not properly closed due to an abnormal or improper shutdown of some process, and should not be used otherwise.

Parameter	Description
a_BrdHandle	specifies an SRL handle to the virtual board device
a_pResetInfo	reserved for future use. If none, set to NULL.
a_pUserInfo	points to user-defined data

Events

If CNF_SUCCESS is returned, the user is notified of the completion status of this request via one of the events listed below, otherwise CNF_ERROR will be returned.

CNFEV_RESET_DEVICES
Reset devices successful or no devices to recover

CNFEV_RESET_DEVICES_FAIL
Reset devices failure

Cautions

This function should only be used to recover previously opened devices that were not closed due to an abnormal shutdown of a process. The most common use of this function is to call it at the beginning of an application in order to make sure that the firmware conferencing resources are properly reset. The function will return the CNFEV_RESET_DEVICES event if it successfully recovered one or more CNF devices, or if there were no devices to recover.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Refer to **cnf_GetErrorInfo( )** for a list of possible error values.

Example

```
#include <cnflib.h>
int main(int argc, char *argv[])
{
    SRL_DEVICE_HANDLE BrdDevice; /* Virtual board device handle. */

    if ((BrdDevice = cnf_Open("brdB1", NULL, NULL)) == CNF_ERROR)
    {
         cout << "cnf_Open failed !!" << endl;
         /* process error */
         return 0;
    }
    else
    {
         if (sr_waitevt(10000) == -1)
         {
              cout << "sr_waitevt TIMEOUT failure" << endl;
              /* process error */
              return 0;
         }
         else
         {
              unsigned int unEvent = sr_getevttype();
              switch(unEvent)
              {
                  case CNFEV_OPEN:
                      /* Open successful - May now use BrdDevice handle */
                      break;

                  case CNFEV_OPEN_FAIL:
                      /* Open failed - Process failure and must close device */
                      cnf_Close(BrdDevice, NULL);
                      exit(0);
                      break;

                  default:
                      /* Received some other event - Process this event */
                      break;
              };
         }
    }

    /**
     * We could use the cnf_GetDeviceCount( ) function to determine if we have
      * any allocated conference or party devices that need to deallocated or
     * we could decide to always reset the board devices by default. If so,
     * we use the cnf_ResetDevices to force a deallocation of these devices.
     */

    if ((cnf_ResetDevices(BrdDevice, NULL, NULL)) == CNF_ERROR
    {
        cout << "cnf_ResetDevices failed !!" << endl;
        /* process error */
        return 0;
    }
    else
    {
       if (sr_waitevt(10000) == -1)
       {
```

```
            cout << "sr_waitevt TIMIEOUT failure" << endl;
            /* process error */
            return 0;
        }
        else
        {
            unsigned int unEvent = sr_getevttype();
            switch(unEvent)
            {
                 case CNFEV_RESET_DEVICES:
                      /* Reset devices successful */
                      break;

                 case CNFEV_RESET_DEVICES_FAIL:
                      /* Reset devices failure - lets use SRL to find reason */
                      break;

                 default:
                      /* Received some other event - process this event */
                      Break;
            };
        }
    }
}
```

See Also

None.

cnf_SetAttributes( )

Name: CNF_RETURN cnf_SetAttributes (a_DevHandle, a_pAttrInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_DevHandle	• device on which to get attributes
CPCNF_ATTR_INFO a_pAttrInfo	• pointer to attribute information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: Asynchronous

Description

The **cnf_SetAttributes( )** function sets the values for one or more attributes on a device. A device can be a board, a conference, or a party.

Parameter	Description
a_DevHandle	specifies the device handle on which to set attributes
a_pAttrInfo	points to the attribute information structure, CNF_ATTR_INFO. This structure in turn points to the CNF_ATTR data structure, which specifies an attribute and its value.
a_pUserInfo	points to user-defined data. If none, set to NULL.

Attributes for each type of device are defined in the ECNF_BRD_ATTR, ECNF_CONF_ATTR, and ECNF_PARTY_ATTR enumerations.

The ECNF_BRD_ATTR data type is an enumeration that defines the following values:

ECNF_BRD_ATTR_ACTIVE_TALKER
enables or disables board level active talker.

ECNF_BRD_ATTR_NOTIFY_INTERVAL
changes the default firmware interval for active talker notification events on the board. The value must be passed in 10 msec units. The default setting is 100 (1 second).

ECNF_BRD_ATTR_TONE_CLAMPING
enables or disables board level tone clamping to reduce the level of DTMF tones heard on a per party basis on the board.

The ECNF_CONF_ATTR data type is an enumeration that defines the following values:

ECNF_CONF_ATTR_DTMF_MASK
specifies a mask for the DTMF digits used for volume control. The digits are defined in the ECNF_DTMF_DIGIT enumeration. The ECNF_DTMF_DIGIT values can be ORed to form the mask using the ECNF_DTMF_MASK_OPERATION enumeration. For a list of ECNF_DTMF_DIGIT values, see the description for CNF_DTMF_CONTROL_INFO.

ECNF_CONF_ATTR_MAX_ACTIVE_TALKERS
sets maximum active talkers. Possible values are 2 to 10 (default).

ECNF_CONF_ATTR_NOTIFY
enables or disables conference notification tone.

ECNF_CONF_ATTR_TONE_CLAMPING
enables or disables conference level tone clamping. Overrides board level value.

The ECNF_PARTY_ATTR data type is an enumeration that defines the following values:

ECNF_PARTY_ATTR_AGC
enables or disables automatic gain control.

ECNF_PARTY_ATTR_BROADCAST
enables or disables broadcast mode. One party can speak while all other parties are muted.

ECNF_PARTY_ATTR_COACH
sets party to coach. Coach is heard by pupil only.

ECNF_PARTY_ATTR_ECHO_CANCEL
enables or disables echo cancellation. Provides 128 taps (16 msec) of echo cancellation.

ECNF_PARTY_ATTR_MUTE
mutes or un-mutes the audio stream received from a conference party.

ECNF_PARTY_ATTR_PRIVILEGE
enables or disables privilege talker. When enabled, party is always included in the conference summation output process, providing its speech level is greater than zero.

ECNF_PARTY_ATTR_PUPIL
sets party to pupil. Pupil hears everyone including the coach.

ECNF_PARTY_ATTR_TARIFF_TONE
enables or disables tariff tone. Party receives periodic tone for duration of the call.

ECNF_PARTY_ATTR_TONE_CLAMPING
enables or disables DTMF tone clamping for the party. Overrides board and conference level values.

Termination Events

CNFEV_SET_ATTRIBUTE
indicates successful completion of this function; that is, attribute values were set

Data Type: CNF_ATTR_INFO

CNFEV_SET_ATTRIBUTE_FAIL
indicates that the function failed

Data Type: CNF_ATTR_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_ATTR
: invalid attribute

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_GetAttributes( )**

cnf_SetDTMFControl( )

Name: CNF_RETURN cnf_SetDTMFControl (a_BrdHandle, a_pDTMFInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_BrdHandle	• SRL handle to the virtual board device
CPCNF_DTMF_CONTROL_INFO a_pDTMFInfo	• pointer to volume control information structure
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Configuration

Mode: asynchronous

Description

The **cnf_SetDTMFControl( )** function returns information about the DTMF digits used to control the conference behavior. The DTMF digit information is stored in the CNF_DTMF_CONTROL_INFO structure.

Parameter	Description
a_BrdHandle	specifies an SRL handle to the virtual board device obtained from a previous open
a_pDTMFInfo	points to the DTMF volume control information structure, CNF_DTMF_CONTROL_INFO
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_SET_DTMF_CONTROL
indicates successful completion of this function; that is, DTMF digit information was set
Data Type: CNF_DTMF_CONTROL_INFO

CNFEV_SET_DTMF_CONTROL_FAIL
indicates that the function failed
Data Type: CNF_DTMF_CONTROL_INFO

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_GetDTMFControl( )**

cnf_SetVideoLayout( )

Name: CNF_RETURN cnf_SetVideoLayout(a_CnfHandle, a_pLayoutInfo, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_CnfHandle	• conference device handle
CPCNF_VIDEO_LAYOUT_INFO a_pLayoutInfo	• pointer to the layout information
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Multimedia Conferencing

Mode: asynchronous

Description

The **cnf_SetVideoLayout( )** function sets the specified video layout on the conference device. The video layout handle can be the same handle as the one acquired when calling the **cnf_GetVideoLayout( )** function, or a new handle created using the **lb_CreateLayoutTemplate( )** function. If the layout handle being set is acquired using the **cnf_GetVideoLayout( )** function, the acquired handle must be from the same device on which the layout is to be set.

Refer to the *Dialogic® Media Toolkit API Library Reference* for more information about the **lb_CreateLayoutTemplate( )** function and other Layout Builder functions.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pLayoutInfo	points to the layout information structure
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

A termination event will return the unique layout handle for the conference device, but this handle may or may not be the same handle as the one used when setting the video layout. You must use the handle returned by the termination event for all future calls requiring a layout handle on a given conference device.

CNFEV_SET_VIDEO_LAYOUT
indicates successful completion of this function; that is, video layset is set
Data Type: CNF_PARTY_INFO

CNFEV_SET_VIDEO_LAYOUT_FAIL
indicates that the function failed
Data Type: NULL

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_GetVideoLayout( )**

cnf_SetVisiblePartyList( )

Name: CNF_RETURN cnf_SetVisiblePartyList(a_CnfHandle, a_pPartyList, a_pUserInfo)

Inputs:

SRL_DEVICE_HANDLE a_CnfHandle	• conference device handle
CNF_VISIBLE_PARTY_LIST a_pPartyList	• pointer to visible party list
void * a_pUserInfo	• pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: Multimedia Conferencing

Mode: asynchronous

Description

The **cnf_SetVisiblePartyList( )** function sets which parties are visible in the video layout regions. Video layout regions are specified in the visible party list information structure. Region handles are acquired using the **lb_GetRegionList( )** function. Refer to the *Dialogic® Media Toolkit API Library Reference* for more information about the **lb_GetRegionList( )** and other Layout Builder functions.

Parameter	Description
a_CnfHandle	specifies the conference device handle obtained from a previous open
a_pPartyList	Points to the visible party list information structure
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_SET_VISABLE_PARTY_LIST
indicates successful completion of this function; that is, list of visible parties returned
Data Type: CNF_VISIBLE_PARTY_LIST

CNFEV_SET_VISIBLE_PARTY_LIST_FAIL
indicates that the function failed
Data Type: CNF_VISIBLE_PARTY_LIST

Cautions

None.

Errors

If this function fails with CNF_ERROR, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the Standard Runtime Library (SRL) Standard Attribute functions, **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**, to obtain the error code and error message. Possible errors for this function include:

ECNF_INVALID_DEVICE
: invalid device handle

ECNF_SUBSYSTEM
: internal subsystem error

Example

See Section 12.1, "Conferencing Example Code and Output", on page 147 for complete example code.

See Also

- **cnf_GetVisiblePartyList( )**

cnf_UnListen( )

discconnect party receive channel from TDM bus

Name: int cnf_UnListen(a_PtyHandle, a_pUserInfo)

Inputs: SRL_DEVICE_HANDLE a_PtyHandle
- valid party device handle

void * a_pUserInfo
- pointer to user-defined data

Returns: CNF_SUCCESS if successful
CNF_ERROR if failure

Includes: srllib.h
cnflib.h

Category: TDM routing

Mode: asynchronous

Description

The **cnf_UnListen( )** function disconnects the conference party receive channel from the TDM bus. The function returns immediately before the operation completes with CNF_SUCCESS, providing argument validation passed; otherwise CNF_ERROR is returned. After the operation completes, a notification event is received and, if successful, the party receive channel is disconnected from the TDM bus time slot.

Calling the **cnf_Listen( )** function to connect to a different TDM bus time slot automatically breaks an existing connection. Thus, when changing connections, there is no need to call the c**cnf_UnListen( )** function first.

Parameter	Description
a_PtyHandle	specifies a party device handle obtained from a previous open
a_pUserInfo	points to user-defined data. If none, set to NULL.

Termination Events

CNFEV_UNLISTEN
indicates successful completion of this function, at which point the party device's receive channel is disconnected from the TDM bus

CNFEV_UNLISTEN_FAIL
indicates that the function failed

Cautions

- This function fails when an invalid party handle is specified or when an invalid TDM bus time slot number is specified.
- It is recommended that the two types of connection methods, **dev_Connect( )/dev_Disconnect( )** and **cnf_listen( )/cnf_unlisten( )**, not be used simultaneously. If they are, then the application must take extreme caution to insure that the connections are properly managed.

Errors

The CNF_ERROR_INFO data structure provides error information for the device handle when an API function fails. Upon failure, call **cnf_GetErrorInfo( )** to return error information for the device in the CNF_ERROR_INFO structure. Error codes are returned as: ECNF_xxxxx

Possible errors for this function include:

ECNF_INVALID_DEVICE
invalid device handle

ECNF_SUBSYSTEM
internal subsystem error

The CNF_EVENT_INFO data structure provides event information for the device handle when a notification event is enabled or disabled. This structure is used by and enabled by **cnf_EnableEvents( )**. Use **sr_waitevt( )**, **sr_enbhdlr( )** or other SRL functions to collect an event code, depending on the programming model in use. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*. Error codes are returned as: CNFEV_xxxxxx

Possible errors for this function include:

CNFEV_INVALID_DEVICE
invalid device handle

CNFEV_SUBSYSTEM
internal subsystem error

Example

For an example, see the example for **cnf_Listen( )**.

See Also

- **cnf_GetXmitSlot( )**
- **cnf_Listen( )**

Events 9

This chapter provides information about the events that may be returned by the Dialogic® Conferencing (CNF) API software. Topics include:

9.1 Event Types

An event indicates that a specific activity has occurred on a channel. The host library reports channel activity to the application program in the form of events, which allows the program to identify and respond to a specific occurrence on a channel. Events provide feedback on the progress and completion of functions and indicate the occurrence of other channel activities. Dialogic® Conferencing (CNF) API library events are defined in the *cnfevts.h* header file.

Events in the Dialogic® Conferencing (CNF) API library can be categorized as follows:

termination events
: These events are returned after the completion of a function call operating in asynchronous mode. The Dialogic® Conferencing (CNF) API library provides a pair of termination events for a function, to indicate successful completion or failure. A termination event is only generated in the process that called the function.

notification events
: These events are requested by the application and provide information about the function call. They are produced in response to a condition specified by the event; for example, the CNFEV_PARTY_ADDED event is generated each time a party is added to a conference. Notification events are enabled or disabled using **cnf_EnableEvents( )** and **cnf_DisableEvents( )**, respectively. Notification events in the conferencing library are disabled by default.

Use **sr_waitevt( )**, **sr_enbhdlr( )** or other SRL functions to collect an event code, depending on the programming model in use. For more information, see the *Dialogic® Standard Runtime Library API Library Reference*.

9.2 Termination Events

The following termination events, listed in alphabetical order, may be returned by the Dialogic® Conferencing (CNF) API software.

CNFEV_ADD_PARTY
: Termination event for **cnf_AddParty( )**. Party added successfully.

CNFEV_ADD_PARTY_FAIL
Termination event for **cnf_AddParty( )**. Add party operation failed.

CNFEV_DISABLE_EVENT
Termination event for **cnf_DisableEvents( )**. Events disabled successfully.

CNFEV_DISABLE_EVENT_FAIL
Termination event for **cnf_DisableEvents( )**. Disable events operation failed.

CNFEV_ENABLE_EVENT
Termination event for **cnf_EnableEvents( )**. Events enabled successfully.

CNFEV_ENABLE_EVENT_FAIL
Termination event for **cnf_EnableEvents( )**. Enable events operation failed.

CNFEV_GET_ACTIVE_TALKER
Termination event for **cnf_GetActiveTalkerList( )**. Active talker list retrieved successfully.

CNFEV_GET_ACTIVE_TALKER_FAIL
Termination event for **cnf_GetActiveTalkerList( )**. Get active talker list operation failed.

CNFEV_GET_ATTRIBUTE
Termination event for **cnf_GetAttributes( )**. Attributes retrieved successfully.

CNFEV_GET_ATTRIBUTE_FAIL
Termination event for **cnf_GetAttributes( )**. Get attributes operation failed.

CNFEV_GET_DEVICE_COUNT
Termination event for **cnf_GetDeviceCount( )**. Device count retrieved successfully.

CNFEV_GET_DEVICE_COUNT_FAIL
Termination event for **cnf_GetDeviceCount( )**. Get device count operation failed.

CNFEV_GET_DTMF_CONTROL
Termination event for **cnf_GetDTMFControl( )**. DTMF digits for volume control retrieved successfully.

CNFEV_GET_DTMF_CONTROL_FAIL
Termination event for **cnf_GetDTMFControl( )**. Get DTMF digits for volume control operation failed.

CNFEV_GET_PARTY_LIST
Termination event for **cnf_GetPartyList( )**. Party list retrieved successfully.

CNFEV_GET_PARTY_LIST_FAIL
Termination event for **cnf_GetPartyList( )**. Get party list operation failed.

CNFEV_GET_PRIVILEGE_TALKER
Termination event for **cnf_GetPrivilegeTalkerList( )**. Privilege talker list retrieved successfully.

CNFEV_GET_PRIVILEGE_TALKER_FAIL
Termination event for **cnf_GetPrivilegeTalkerList( )**. Get privilege talker list operation failed.

CNFEV_GET_VIDEO_LAYOUT
Termination event for **cnf_GetVideoLayout( )**. Video layout retrieved successfully.

CNFEV_GET_VIDEO_LAYOUT_FAIL
Termination event for **cnf_GetVideoLayout( )**. Get video layout failed.

CNFEV_GET_VISIBLE_PARTY_LIST
Termination event for **cnf_GetPartyList( )**. Visible party list retrieved successfully.

CNFEV_GET_VISIBLE_PARTY_LIST_FAIL
Termination event for **cnf_GetPartyList( )**. Get visible party failed.

CNFEV_OPEN
Termination event for **cnf_Open( )**. Board device handle opened successfully.

CNFEV_OPEN_CONF
Termination event for **cnf_OpenConference( )**. Conference device handle opened successfully.

CNFEV_OPEN_CONF_FAIL
Termination event for **cnf_OpenConference( )**. Open conference operation failed.

CNFEV_OPEN_FAIL
Termination event for **cnf_Open( )**. Open board operation failed.

CNFEV_OPEN_PARTY
Termination event for **cnf_OpenParty( )**. Party device handle opened successfully.

CNFEV_OPEN_PARTY_FAIL
Termination event for **cnf_OpenParty( )**. Open party operation failed.

CNFEV_REMOVE_PARTY
Termination event for **cnf_RemoveParty( )**. Party removed successfully.

CNFEV_REMOVE_PARTY_FAIL
Termination event for **cnf_RemoveParty( )**. Remove party operation failed.

CNFEV_SET_ATTRIBUTE
Termination event for **cnf_SetAttributes( )**. Attribute(s) set successfully.

CNFEV_SET_ATTRIBUTE_FAIL
Termination event for **cnf_SetAttributes( )**. Set attribute(s) operation failed.

CNFEV_SET_DTMF_CONTROL
Termination event for **cnf_SetDTMFControl( )**. DTMF digits for volume control set successfully.

CNFEV_SET_DTMF_CONTROL_FAIL
Termination event for **cnf_SetDTMFControl( )**. Set DTMF digit operation failed.

CNFEV_SET_VIDEO_LAYOUT
Termination event for **cnf_SetVideoLayout( )**. Video layout set successfully.

CNFEV_SET_VIDEO_LAYOUT_FAIL
Termination event for **cnf_SetVideoLayout( )**. Set video layout failed.

CNFEV_SET_VISIBLE_PARTY_LIST
Termination event for **cnf_SetVisiblePartyList( )**. Visible party list set successfully.

CNFEV_SET_VISIBLE_PARTY_LIST_FAIL
Termination event for **cnf_SetVisiblePartyList( )**. Set visible party list failed.

9.3 Notification Events

The following notification events, listed in alphabetical order, may be returned by the conferencing software:

CNFEV_ACTIVE_TALKER
Notification event for active talker. Active talker feature is set using **cnf_SetAttributes( )**. Notification event is enabled using **cnf_EnableEvents( )**.

Data Type: CNF_ACTIVE_TALKER_INFO

CNFEV_CONF_CLOSED
Notification event for a conference that has been closed. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A.

Data Type: CNF_CONF_CLOSED_EVENT_INFO

CNFEV_CONF_OPENED
Notification event for a conference that has been opened. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A.

Data Type: CNF_CONF_OPENED_EVENT_INFO

CNFEV_DTMF_DETECTED
Notification event when DTMF digit has been detected in the conference. Enabled using **cnf_EnableEvents( )**.

Data Type: CNF_DTMF_EVENT_INFO

CNFEV_ERROR
General error event. Returned when an unexpected error occurs while processing a notification event.

CNFEV_EXCEEDED_CONF_LICENSE
Notification event that the maximum Conferencing licenses have been exceeded. Enabled using **cnf_EnableEvents( )**. When using wideband voice conferencing, the parties can be added to a conference without initially consuming a Conferencing license. This event notifies the application if Conferencing licenses are exhausted as it starts the media on the parties in the conference.

Data Type: CNF_EXCEEDED_CONF_LICENSE_EVENT_INFO

CNFEV_PARTY_ADDED
Notification event for a party that has been added. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A.

Data Type: CNF_PARTY_ADDED_EVENT_INFO

CNFEV_PARTY_CLOSED
Notification event for a party that has been closed. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A. This event is only supported on an MCX board device.

Data Type: CNF_PARTY_CLOSED_EVENT_INFO

CNFEV_PARTY_OPENED

Notification event for a party that has been opened. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A. This event is only supported on an MCX board device.

Data Type: CNF_PARTY_OPENED_EVENT_INFO

CNFEV_PARTY_REMOVED

Notification event for a party that has been removed, either directly through **cnf_RemoveParty( )** or indirectly through **cnf_CloseConference( )**. Enabled using **cnf_EnableEvents( )**. Useful in multiprocessing; for example, when process B wants to be notified of activity in process A.

Data Type: CNF_PARTY_REMOVED_EVENT_INFO

Data Structures 10

This chapter provides an alphabetical reference to the data structures used by the Dialogic® Conferencing (CNF) API software. The following data structures are described:

CNF_ACTIVE_TALKER_INFO

```
typedef struct CNF_ACTIVE_TALKER_INFO
{
      unsigned int unVersion;           /* version of structure */
      unsigned int unPartyCount;        /* number of party handles in list */
      SRL_DEVICE_HANDLE *pPartyList;    /* pointer to list of party handles */
} CNF_ACTIVE_TALKER_INFO, *PCNF_ACTIVE_TALKER_INFO;
typedef const CNF_ACTIVE_TALKER_INFO * CPCNF_ACTIVE_TALKER_INFO;
```

Description

The CNF_ACTIVE_TALKER_INFO data structure provides active talker information after the application receives the CNFEV_ACTIVE_TALKER notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_ACTIVE_TALKER_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_ACTIVE_TALKER_INFO_VERSION_0.

unPartyCount
: specifies the number of party handles in the list.

unPartyList
: points to a list of party handles.

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_ATTR

```
typedef struct CNF_ATTR
{
      unsigned int unVersion;    /* version of structure */
      unsigned int nAttrType;    /* attribute type */
      unsigned unAttrValue;      /* attribute value */
} CNF_ATTR, *PCNF_ATTR;
```

Description

The CNF_ATTR data structure specifies the attributes of a party, conference, or board. This structure is contained in the CNF_ATTR_INFO structure, and is used by the **cnf_SetAttributes( )** and **cnf_GetAttributes( )** functions.

Field Descriptions

The fields of the CNF_ATTR data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_ATTR_VERSION_0.

nAttrType
: specifies the type of attribute: board, conference, or party. The attribute type is defined in the ECNF_BRD_ATTR, ECNF_CONF_ATTR, and ECNF_PARTY_ATTR enumerations. All attributes are disabled by default.

pAttrValue
: specifies the value of the attribute. For attributes that can be enabled or disabled, the attribute value is defined in the ECNF_ATTR_STATE enumeration. Possible values include:
  - ECNF_ATTR_STATE_DISABLED – attribute is disabled
  - ECNF_ATTR_STATE_ENABLED – attribute is enabled

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_ATTR_INFO

```
typedef struct CNF_ATTR_INFO
{
      unsigned int unVersion;    /* version of structure */
      unsigned int nAttrCount;   /* number of attributes in list */
      PCNF_ATTR pAttrList;       /* pointer to attribute list */
} CNF_ATTR_INFO, *PCNF_ATTR_INFO;
```

Description

The CNF_ATTR_INFO data structure contains information about the attributes of a party, conference, or board. This structure is used by the **cnf_SetAttributes( )** and **cnf_GetAttributes( )** functions.

Field Descriptions

The fields of the CNF_ATTR_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_ATTR_INFO_VERSION_0.

nAttrCount
: specifies the number of attributes in the list.

pAttrList
: points to the attribute list. See the CNF_ATTR data structure for more information.

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_CLOSE_CONF_INFO

```
typedef struct CNF_CLOSE_CONF_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_CLOSE_CONF_INFO, *PCNF_CLOSE_CONF_INFO;
typedef const CNF_CLOSE_CONF_INFO * CPCNF_CLOSE_CONF_INFO;
```

Description

The CNF_CLOSE_CONF_INFO structure is used by the **cnf_CloseConference( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_CLOSE_INFO

```
typedef struct CNF_CLOSE_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_CLOSE_INFO, *PCNF_CLOSE_INFO;
typedef const CNF_CLOSE_INFO * CPCNF_CLOSE_INFO;
```

Description

The CNF_CLOSE_INFO data structure is used by the **cnf_Close( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_CLOSE_PARTY_INFO

```
typedef struct CNF_CLOSE_PARTY_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_CLOSE_PARTY_INFO, *PCNF_CLOSE_PARTY_INFO;
typedef const CNF_CLOSE_PARTY_INFO * CPCNF_CLOSE_PARTY_INFO;
```

Description

The CNF_CLOSE_PARTY_INFO data structure is used by the **cnf_CloseParty( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_CONF_CLOSED_EVENT_INFO

```
typedef struct CNF_CONF_CLOSED_EVENT_INFO
{
      unsigned int unVersion;            /* version of structure */
      const char *szConfName;            /* conference device name */
} CNF_CONF_CLOSED_EVENT_INFO, *PCNF_CONF_CLOSED_EVENT_INFO;
typedef const CNF_CONF_CLOSED_EVENT_INFO * CPCNF_CONF_CLOSED_EVENT_INFO;
```

Description

The CNF_CONF_CLOSED_EVENT_INFO data structure provides information about the conference after the application receives the CNFEV_CONF_CLOSED notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_CONF_CLOSED_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_CONF_CLOSED_EVENT_INFO_VERSION_0.

szConfName
: points to the conference device name

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_CONF_OPENED_EVENT_INFO

```
typedef struct CNF_CONF_OPENED_EVENT_INFO
{
      unsigned int unVersion;            /* version of structure */
      SRL_DEVICE_HANDLE ConfHandle;      /* conference device handle */
      const char *szConfName;            /* conference device name */
} CNF_CONF_OPENED_EVENT_INFO, *PCNF_CONF_OPENED_EVENT_INFO;
typedef const CNF_CONF_OPENED_EVENT_INFO * CPCNF_CONF_OPENED_EVENT_INFO;
```

Description

The CNF_CONF_OPENED_EVENT_INFO data structure provides information about the conference after the application receives the CNFEV_CONF_OPENED notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_CONF_OPENED_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_CONF_OPENED_EVENT_INFO_VERSION_0.

ConfHandle
: specifies the conference device handle

szConfName
: points to the conference device name

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_DEVICE_COUNT_INFO

```
typedef struct CNF_DEVICE_COUNT_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unFreePartyCount;     /* number of free parties */
      unsigned int unMaxPartyCount;      /* number of maximum parties */
      unsigned int unFreeConfCount;      /* number of free conferences */
      unsigned int unMaxConfCount;       /* number of maximum conferences */
} CNF_DEVICE_COUNT_INFO, *PCNF_DEVICE_COUNT_INFO;
typedef const CNF_DEVICE_COUNT_INFO * CPCNF_DEVICE_COUNT_INFO;
```

Description

The CNF_DEVICE_COUNT_INFO data structure stores information about the number of devices on a board. This structure is used by the **cnf_GetDeviceCount( )** function.

Field Descriptions

The fields of the CNF_DEVICE_COUNT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_DEVICE_COUNT_INFO_VERSION_0.

unFreePartyCount
: specifies the number of free parties remaining on the board

unMaxPartyCount
: specifies the maximum number of parties that can be opened on the board

unFreeConfCount
: specifies the number of free conferences remaining on the board

unMaxConfCount
: specifies the maximum number of conferences that can be opened on the board

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_DTMF_CONTROL_INFO

```
typedef struct CNF_DTMF_CONTROL_INFO
{
      unsigned int unVersion;               /* version of structure */
      ECNF_ATTR_STATE eDTMFControlState;     /* enable/disable DMTF control */
      ECNF_DTMF_DIGIT eVolumeUpDigit;        /* volume up digit */
      ECNF_DTMF_DIGIT eVolumeDownDigit;      /* volume down digit */
      ECNF_DTMF_DIGIT eVolumeResetDigit;        /* volume reset digit */
} CNF_DTMF_CONTROL_INFO, *PCNF_DTMF_CONTROL_INFO;
typedef const CNF_DTMF_CONTROL_INFO * CPCNF_DTMF_CONTROL_INFO;
```

Description

The CNF_DTMF_CONTROL_INFO data structure stores information about DTMF values used to control the volume of a conference. This structure is used by the **cnf_SetDTMFControl( )** and **cnf_GetDTMFControl( )** functions.

Field Descriptions

The fields of the CNF_DTMF_CONTROL_INFO data structure are described as follows:

unVersion
specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_DTMF_CONTROL_INFO_VERSION_0.

eDTMFControlState
enables or disables DTMF digits used to control the volume of a conference. The ECNF_ATTR_STATE data type is an enumeration that defines the following values:

- ECNF_ATTR_STATE_DISABLED – attribute is disabled
- ECNF_ATTR_STATE_ENABLED – attribute is enabled

eVolumeUpDigit
specifies the DTMF digit used to increase the volume. The volume increment is 2 dB. The ECNF_DTMF_DIGIT data type is an enumeration that defines the following values:

- ECNF_DTMF_DIGIT_1 – specifies DTMF 1
- ECNF_DTMF_DIGIT_2 – specifies DTMF 2
- ECNF_DTMF_DIGIT_3 – specifies DTMF 3
- ECNF_DTMF_DIGIT_4 – specifies DTMF 4
- ECNF_DTMF_DIGIT_5 – specifies DTMF 5
- ECNF_DTMF_DIGIT_6 – specifies DTMF 6
- ECNF_DTMF_DIGIT_7 – specifies DTMF 7
- ECNF_DTMF_DIGIT_8 – specifies DTMF 8
- ECNF_DTMF_DIGIT_9 – specifies DTMF 9
- ECNF_DTMF_DIGIT_0 – specifies DTMF 0
- ECNF_DTMF_DIGIT_STAR – specifies DTMF *
- ECNF_DTMF_DIGIT_POUND – specifies DTMF #
- ECNF_DTMF_DIGIT_A – specifies DTMF A
- ECNF_DTMF_DIGIT_B – specifies DTMF B
- ECNF_DTMF_DIGIT_C – specifies DTMF C
- ECNF_DTMF_DIGIT_D – specifies DTMF D

eVolumeDownDigit
specifies the DTMF digit used to decrease the volume. The volume decrement is 2 dB. The ECNF_DTMF_DIGIT data type is an enumeration that defines the values for DTMF digits. See eVolumeUpDigit for a list of values.

eVolumeResetDigit
specifies the DTMF digit used to reset the volume to its default level. The default volume and origin is 0 dB. The ECNF_DTMF_DIGIT data type is an enumeration that defines the values for DTMF digits. See eVolumeUpDigit for a list of values.

Example

For an example of this data structure, see Section 12.1, "Conferencing Example Code and Output", on page 147.

CNF_DTMF_EVENT_INFO

```
typedef struct CNF_DTMF_EVENT_INFO
{
      unsigned int unVersion;               /* version of structure */
      SRL_DEVICE_HANDLE PartyHandle;        /* party device handle */
      ECNF_DTMF_DIGIT eDigit;               /* detected DTMF digit */
} CNF_DTMF_EVENT_INFO, *PCNF_DTMF_EVENT_INFO;
typedef const CNF_DTMF_EVENT_INFO * CPCNF_DTMF_EVENT_INFO;
```

Description

The CNF_DTMF_EVENT_INFO data structure provides DTMF digit information to the party after the application receives the CNFEV_DTMF_EVENT notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_DTMF_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_DTMF_EVENT_INFO_VERSION_0.

PartyHandle
: specifies the party device handle

eDigit
: specifies the DTMF digit that was detected. The ECNF_DTMF_DIGIT data type is an enumeration that defines the following values:
  - ECNF_DTMF_DIGIT_1 – specifies DTMF 1
  - ECNF_DTMF_DIGIT_2 – specifies DTMF 2
  - ECNF_DTMF_DIGIT_3 – specifies DTMF 3
  - ECNF_DTMF_DIGIT_4 – specifies DTMF 4
  - ECNF_DTMF_DIGIT_5 – specifies DTMF 5
  - ECNF_DTMF_DIGIT_6 – specifies DTMF 6
  - ECNF_DTMF_DIGIT_7 – specifies DTMF 7
  - ECNF_DTMF_DIGIT_8 – specifies DTMF 8
  - ECNF_DTMF_DIGIT_9 – specifies DTMF 9
  - ECNF_DTMF_DIGIT_0 – specifies DTMF 0
  - ECNF_DTMF_DIGIT_STAR – specifies DTMF *
  - ECNF_DTMF_DIGIT_POUND – specifies DTMF #
  - ECNF_DTMF_DIGIT_A – specifies DTMF A
  - ECNF_DTMF_DIGIT_B – specifies DTMF B
  - ECNF_DTMF_DIGIT_C – specifies DTMF C
  - ECNF_DTMF_DIGIT_D – specifies DTMF D

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_ERROR_INFO

```
typedef struct CNF_ERROR_INFO
{
      unsigned int unVersion;           /* version of structure */
      unsigned int unErrorCode;         /* error code */
      const char *szErrorString;        /* error string */
      const char *szAdditionalInfo;     /* additional error information string */
} CNF_ERROR_INFO, *PCNF_ERROR_INFO;
typedef const CNF_ERROR_INFO * CPCNF_ERROR_INFO;
```

Description

The CNF_ERROR_INFO data structure provides error information for the device handle when an API function fails. This structure is used by the **cnf_GetErrorInfo( )** function.

Field Descriptions

The fields of the CNF_ERROR_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_ERROR_INFO_VERSION_0.

unErrorCode
: specifies the error code

szErrorString
: points to the error message

szAdditionalInfo
: points to additional error information

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_EVENT_INFO

```
typedef struct CNF_EVENT_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unEventCount;         /* number of events in list */
      unsigned int *punEventList;        /* pointer to event list */
} CNF_EVENT_INFO, *PCNF_EVENT_INFO;
typedef const CNF_EVENT_INFO * CPCNF_EVENT_INFO;
```

Description

The CNF_EVENT_INFO data structure provides event information for the device handle when a notification event is enabled or disabled. This structure is used by the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_EVENT_INFO_VERSION_0.

unEventCount
: specifies the number of events in the list.

punEventList
: points to a list of events.

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_OPEN_CONF_INFO

```
typedef struct CNF_OPEN_CONF_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_OPEN_CONF_INFO, *PCNF_OPEN_CONF_INFO;
typedef const CNF_OPEN_CONF_INFO * CPCNF_OPEN_CONF_INFO;
```

Description

The CNF_OPEN_CONF_INFO data structure is used by the **cnf_OpenConference( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_OPEN_CONF_RESULT

```
typedef struct CNF_OPEN_CONF_RESULT
{
      unsigned int unVersion;           /* version of structure */
      const char * szConfName;          /* conference device name */
      SRL_DEVICE_HANDLE ConfHandle;     /* conference device handle */
} CNF_OPEN_CONF_RESULT, *PCNF_OPEN_CONF_RESULT;
typedef const CNF_OPEN_CONF_RESULT * CPCNF_OPEN_CONF_RESULT;
```

Description

The CNF_OPEN_CONF_RESULT data structure contains result information returned with the CNFEV_OPEN_CONF event. This termination event is returned by the **cnf_OpenConference( )** function.

Field Descriptions

The fields of the CNF_OPEN_CONF_RESULT data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_OPEN_CONF_RESULT_VERSION_0.

szConfName
: specifies the conference device name

ConfHandle
: specifies the conference device handle

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_OPEN_INFO

```
typedef struct CNF_OPEN_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_OPEN_INFO, *PCNF_OPEN_INFO;
typedef const CNF_OPEN_INFO * CPCNF_OPEN_INFO;
```

Description

The CNF_OPEN_INFO data structure is used by the **cnf_Open( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_OPEN_PARTY_INFO

```
typedef struct CNF_OPEN_PARTY_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unRFU;                /* reserved for future use */
} CNF_OPEN_PARTY_INFO, *PCNF_OPEN_PARTY_INFO;
typedef const CNF_OPEN_PARTY_INFO * CPCNF_OPEN_PARTY_INFO;
```

Description

The CNF_OPEN_PARTY_INFO data structure is used by the **cnf_OpenParty( )** function.

Note: This structure is reserved for future use. NULL must be passed.

CNF_OPEN_PARTY_RESULT

```
typedef struct CNF_OPEN_PARTY_RESULT
{
      unsigned int unVersion;            /* version of structure */
      const char * szPartyName;           /* party device name */
      SRL_DEVICE_HANDLE PartyHandle;      /* party device handle */
} CNF_OPEN_PARTY_RESULT, *PCNF_OPEN_PARTY_RESULT;
typedef const CNF_OPEN_PARTY_RESULT * CPCNF_OPEN_PARTY_RESULT;
```

Description

The CNF_OPEN_PARTY_RESULT data structure contains result information returned with the CNFEV_OPEN_PARTY event. This termination event is returned by the **cnf_OpenParty( )** function.

Field Descriptions

The fields of the CNF_OPEN_PARTY_RESULT data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_OPEN_PARTY_RESULT_VERSION_0.

szPartyName
: specifies the party device name

PartyHandle
: specifies the party device handle

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_PARTY_ADDED_EVENT_INFO

```
typedef struct CNF_PARTY_ADDED_EVENT_INFO
{
      unsigned int unVersion;            /* version of structure */
      SRL_DEVICE_HANDLE ConfHandle;      /* conference device handle */
      const char *szConfName;            /* conference device name */
      SRL_DEVICE_HANDLE PartyHandle;     /* party device handle */
      const char *szPartyName;           /* party device name */
} CNF_PARTY_ADDED_EVENT_INFO, *PCNF_PARTY_ADDED_EVENT_INFO;
typedef const CNF_PARTY_ADDED_EVENT_INFO * CPCNF_PARTY_ADDED_EVENT_INFO;
```

Description

The CNF_PARTY_ADDED_EVENT_INFO data structure provides information about the party after the application receives the CNFEV_PARTY_ADDED notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_PARTY_ADDED_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_PARTY_ADDED_EVENT_INFO_VERSION_0.

ConfHandle
: specifies the conference device handle

szConfName
: points to the conference device name

PartyHandle
: specifies the party device handle

szPartyName
: points to the party device name

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_PARTY_INFO

```
typedef struct CNF_PARTY_INFO
{
      unsigned int unVersion;            /* version of structure */
      unsigned int unPartyCount;         /* number of party handles in list */
      SRL_DEVICE_HANDLE *pPartyList;     /* pointer to list of party handles */
} CNF_PARTY_INFO, *PCNF_PARTY_INFO;
typedef const CNF_PARTY_INFO * CPCNF_PARTY_INFO;
```

Description

The CNF_PARTY_INFO data structure stores information on a party that is opened, added or removed. This structure is used by the **cnf_OpenParty( )**, **cnf_AddParty( )**, and **cnf_RemoveParty( )** functions. This structure is also returned as the data to several events; for example, the CNF_OPEN_PARTY termination event.

Field Descriptions

The fields of the CNF_PARTY_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_PARTY_INFO_VERSION_0.

unPartyCount
: specifies the number of party handles in the list.

pPartyList
: points to a list of party handles.

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_PARTY_REMOVED_EVENT_INFO

```
typedef struct CNF_PARTY_REMOVED_EVENT_INFO
{
      unsigned int unVersion;            /* version of structure */
      SRL_DEVICE_HANDLE ConfHandle;      /* conference device handle */
      const char *szConfName;            /* conference device name */
      SRL_DEVICE_HANDLE PartyHandle;     /* party device handle */
      const char *szPartyName;           /* party device name */
} CNF_PARTY_REMOVED_EVENT_INFO, *PCNF_PARTY_REMOVED_EVENT_INFO;
typedef const CNF_PARTY_REMOVED_EVENT_INFO * CPCNF_PARTY_REMOVED_EVENT_INFO;
```

Description

The CNF_PARTY_REMOVED_EVENT_INFO data structure provides information about the party after the application receives the CNFEV_PARTY_REMOVED notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_PARTY_REMOVED_EVENT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_PARTY_REMOVED_EVENT_INFO_VERSION_0.

ConfHandle
: specifies the conference device handle

szConfName
: points to the conference device name

PartyHandle
: specifies the party device handle

szPartyName
: points to the party device name

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_PRIVILEGE_TALKER_INFO

```
typedef struct CNF_PRIVILEGE_TALKER_INFO
{
      unsigned int unVersion;                /* version of structure */
      unsigned int unPartyCount;             /* number of party handles in list */
      SRL_DEVICE_HANDLE *pPartyList;         /* pointer to list of party handles */
} CNF_PRIVILEGE_TALKER_INFO, *PCNF_PRIVILEGE_TALKER_INFO;
typedef const CNF_PRIVILEGE_TALKER_INFO * CPCNF_PRIVILEGE_TALKER_INFO;
```

Description

The CNF_PRIVILEGE_TALKER_INFO data structure provides privilege talker information after the application receives the CNFEV_PRIVILEGE_TALKER notification event. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_PRIVILEGE_TALKER_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_PRIVILEGE_TALKER_INFO_VERSION_0.

unPartyCount
: specifies the number of party handles in the list

pPartyList
: points to a list of party handles

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_VIDEO_LAYOUT_INFO

```
typedef struct CNF_VIDEO_LAYOUT_INFO
{
      unsigned int unVersion;            /* version of structure */
      ELB_LAYOUT_SIZE   unLayoutSize;    /* layout screen size */
      LB_FRAME_HANDLE   LayoutHandle;    /* layout region handle */
} CNF_VIDEO_LAYOUT_INFO, *PCNF_CVIDEO_LAYOUT_INFO;
typedef const CNF_VIDEO_LAYOUT_INFO, *PCNF_CVIDEO_LAYOUT_INFO;
```

Description

The CNF_VIDEO_LAYOUT_INFO data structure provides information about the video layout. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_VIDEO_LAYOUT_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_VIDEO_LAYOUT_INFO_VERSION_0.

unLayoutSize
: specifies the layout screen size. Possible values include:
  - ELB_LAYOUT_SIZE_SUBQCIF – Layout size [128 x 96]
  - ELB_LAYOUT_SIZE_QCIF – Layout size [176 x 144]
  - ELB_LAYOUT_SIZE_CIF – Layout size [352 x 288]

LayoutHandle
: points to the layout region handle

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_VISIBLE_PARTY_INFO

```
typedef struct CNF_VISIBLE_PARTY_INFO
{
      unsigned int unVersion;            /* version of structure */
      SRL_DEVICE_HANDLE PartyHandle;     /* party device handle */
      LB_FRAME_HANDLE RegionHandle;      /* layout region handle */
} CNF_VISIBLE_PARTY_INFO, *PCNF_VISIBLE_PARTY_INFO;
typedef const CNF_VISIBLE_PARTY_INFO * CPCNF_VISIBLE_PARTY_INFO;
```

Description

The CNF_VISIBLE_PARTY_INFO data structure provides information about the visible party in a specified conference. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_VISIBLE_PARTY_INFO data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_VISIBLE_PARTY_INFO_VERSION_0.

PartyHandle
: specifies the party device handle

RegionHandle
: specifies the region handle

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

CNF_VISIBLE_PARTY_LIST

```
typedef struct CNF_VISIBLE_PARTY_LIST
{
      unsigned int unVersion;                /* version of structure */
      unsigned int unCount;                  /* size of visible party list */
      PCNF__VISIBLE_PARTY_INFO pPartyList;   /* pointer to visible party info list */
} CNF_VISIBLE_PARTY_LIST, *PCNF_VISIBLE_PARTY_LIST;
typedef const CNF_VISIBLE_PARTY_LIST * CPCNF_VISIBLE_PARTY_LIST;
```

Description

The CNF_VISIBLE_PARTY_LIST data structure provides information about the parties visible in a specified conference. Notification events are enabled using the **cnf_EnableEvents( )** function.

Field Descriptions

The fields of the CNF_VISIBLE_PARTY_LIST data structure are described as follows:

unVersion
: specifies the version of the data structure. Used to ensure that an application is binary compatible with future changes to this data structure. The current version of this data structure is CNF_VISIBLE_PARTY_LIST_VERSION_0.

unCount
: specifies the visible party list size

pPartyList
: points to the visible party information list

Example

For an example of this data structure, see Section 12.1, “Conferencing Example Code and Output”, on page 147.

SC_TSINFO

```
typedef struct {
      unsigned long   sc_numts;
      long           *sc_tsarrayp;
} SC_TSINFO;
```

Description

The SC_TSINFO data structure defines TDM bus time slot information. It may contain CT Bus or HMP soft CT Bus time slot information. It is used by the **cnf_GetXmitSlot( )** and **cnf_Listen( )** functions.

Field Descriptions

The fields of the SC_TSINFO structure are described as follows:

sc_numts
: specifies the number of TDM bus time slots to follow; must be set to 1

sc_tsarrayp
: specifies the time slot ID number

Example

See **cnf_Listen( )** for an example of how to use the SC_TSINFO structure.

Error Codes 11

This chapter describes the error codes used by the Dialogic® Conferencing (CNF) API software. Error codes are defined in *cnferrs.h*.

Dialogic® Conferencing (CNF) API library functions return a value that indicates the success or failure of a function call. Success is indicated by CNF_SUCCESS, and failure is indicated by CNF_ERROR. If a library function returns CNF_ERROR to indicate failure, use **cnf_GetErrorInfo( )** to obtain the reason for the error. Alternatively, you can use the standard attribute function **ATDV_LASTERR( )** to return the error code and **ATDV_ERRMSGP( )** to return the error description. These functions are described in the *Dialogic® Standard Runtime Library API Library Reference*.

Note: The following functions cannot use the Dialogic® Standard Runtime Library standard attribute functions to process errors: **cnf_Close( )**, **cnf_CloseConference( )**, and **cnf_CloseParty( )**.

If an error occurs during execution of an asynchronous function, an error event, preceded by "CNFEV_" is sent to the application. No change of state is triggered by this event. Upon receiving the CNFEV_ERROR event, the application can retrieve the reason for the failure using **ATDV_LASTERR( )** and **ATDV_ERRMSGP( )**.

The error codes used by the conferencing software are described as follows:

ECNF_FIRMWARE
: firmware error

ECNF_INVALID_ATTR
: invalid device attribute

ECNF_INVALID_DEVICE
: invalid device

ECNF_INVALID_EVENT
: invalid device event

ECNF_INVALID_HANDLE
: invalid device handle

ECNF_INVALID_NAME
: invalid device name

ECNF_INVALID_PARM
: invalid parameter

ECNF_INVALID_STATE
: invalid device state for requested operation

ECNF_LIBRARY
: library error

ECNF_MEMORY_ALLOC
: memory allocation error

ECNF_NOERROR
no error

ECNF_SUBSYSTEM
internal subsystem error

ECNF_SYSTEM
system error

ECNF_UNSUPPORTED_API
API not currently supported

ECNF_UNSUPPORTED_FUNC
requested functionality not supported

ECNF_UNSUPPORTED_TECH
technology not currently supported

Supplementary Reference Information 12

This chapter provides reference information about the following topic:

12.1 Conferencing Example Code and Output

Written in the C++ programming language, the example code exercises Dialogic® Conferencing (CNF) API functions and data structures. It is intended to illustrate how the Dialogic® Conferencing (CNF) API functions and data structures are used in a simple application. It is not intended to be used in a production environment.

The output from the example code is provided in Figure 2, "Conferencing (CNF) Example Code Output", on page 180 and Figure 3, "Conferencing (MCX) Example Code Output", on page 187.

Figure 1. Conferencing Example Code

```
#include <cnflib.h>
#include <lb_mtklib.h>
#include <srllib.h>
#include <iostream>

#ifdef WIN32
#else
#include <unistd.h>
#endif

using namespace std;

#define MAX_CNF_BRD_ATTR   (ECNF_BRD_ATTR_END_OF_LIST - CNF_BRD_ATTR_BASE)
#define MAX_CNF_CONF_ATTR  (ECNF_CONF_ATTR_END_OF_LIST - CNF_CONF_ATTR_BASE)
#define MAX_CNF_PTY_ATTR   (ECNF_PARTY_ATTR_END_OF_LIST - CNF_PARTY_ATTR_BASE)

LB_FRAME_HANDLE g_LayoutHandle;

/**
 * @struct SRL_METAEVENT
 */
struct SRL_METAEVENT
{
   long              EventType;        ///< Event type
   SRL_DEVICE_HANDLE EventDevice;      ///< Event device handle
   void *            pEventData;       ///< Pointer to event data
   long              EventDataLength;  ///< Event data length
   void *            pEventUserInfo;   ///< Pointer to user defined data
};
typedef SRL_METAEVENT * PSRL_METAEVENT;

/**
 * @enum CNF_TYPE
 */
```

```
typedef enum ECNF_TYPE
{
   ECNF_TYPE_CNF  = 0,
   ECNF_TYPE_MCX  = 1
} ECNF_TYPE;


/**
 * @fn srl_GetMetaEvent
 */
void srl_GetMetaEvent(PSRL_METAEVENT a_pMetaEvent);

/**
 * @fn ProcessErrorInformation
 */
void ProcessErrorInformation();

/**
 * @fn ProcessMetaEvent
 */
void ProcessMetaEvent(char * a_szString);

/**
 * @fn Process conferencing event(s) functions.

 */

void Process_AddParty_Event();
void Process_Board_Event();
void Process_DisableEvents_Event();
void Process_EnableEvents_Event();
void Process_GetActiveTalkerList_Event();
void Process_GetAttributes_Event();
void Process_GetDeviceCount_Event();
void Process_GetDTMFControl_Event();
void Process_GetPartyList_Event();
void Process_GetVideoLayout_Event();
void Process_GetVisiblePartyList_Event();
void Process_OpenBoard_Event();
void Process_OpenConference_Event();
void Process_OpenParty_Event();
void Process_RemoveParty_Event();
void Process_ResetDevices_Event();
void Process_SetAttributes_Event();
void Process_SetDTMFControl_Event();
void Process_SetVideoLayout_Event();
void Process_SetVisiblePartyList_Event();


/**
 * @fn main
 */
int main(int nArgCount, char *pArgList[])
{
   cout << "Conferencing (CNF/MCX) Sample Application" << endl;
   cout << "=========================================" << endl << endl;

   std::string szBrdName = "cnfB1";
   ECNF_TYPE eType = ECNF_TYPE_CNF;

   switch (nArgCount)
   {
      case 1:

         // Use default cnfB1 board name.
         break;
```

```
      case 2:
         // Use user specified board name.
         szBrdName = pArgList[1];
         if (szBrdName.compare(0, 3, "mcx", 3) == 0)
         {
            eType = ECNF_TYPE_MCX;
         }
         break;

      default:
         cout << "Invalid number of arguments provided. defaulting to cnfB1." << endl << endl;
         break;
   }

   cout << "Board Name is: " << szBrdName.c_str() << endl << endl;

   /*************************************************************************
    * SETUP SRL MODE OF FUNCTIONALITY.
    *************************************************************************/
   int nSRLMode = SR_POLLMODE;
   if (sr_setparm(SRL_DEVICE, SR_MODEID, &nSRLMode) == -1)
   {
      cout << "Error setting SRL mode !!" << endl;
      return 0;
   }

   SRL_DEVICE_HANDLE BrdDevice;
SRL_DEVICE_HANDLE CnfDevice;
   SRL_DEVICE_HANDLE PtyDevice;

/*************************************************************************
 * OPEN A BOARD DEVICE
 *
 * NOTE: THIS CALL IS EXPECTED TO FAIL DUE TO BAD PARAMETERS. TEST TO SEE IF
 *       ERROR HANDLING IS WORKING CORRECTLY. PASSING INVALID DEVICE NAME.
 *************************************************************************/
   if ((BrdDevice = cnf_Open(NULL, NULL, NULL)) == CNF_ERROR)
   {
      ///
      // Good, we were expecting this to happen. Lets get the error information
         cout << "cnf_Open failure!! : Expected failure due to the following" << endl;
         ProcessErrorInformation();
   }

   /*************************************************************************
 * OPEN A BOARD DEVICE
 *
 * NOTE: THIS CALL IS EXPECTED TO FAIL DUE TO BAD PARAMETERS. TEST TO SEE IF
 *       ERROR HANDLING IS WORKING CORRECTLY. PASSING INVALID DEVICE NAME.
 *************************************************************************/
   if ((BrdDevice = cnf_Open("blah_blah", NULL, NULL)) == CNF_ERROR)
   {
      ///
      // Good, we were expecting this to happen. Lets get the error information
         cout << "cnf_Open failure!! : Expected failure due to the following" << endl;
         ProcessErrorInformation();
   }
/*************************************************************************
 * OPEN A BOARD DEVICE.
 *************************************************************************/
   if ((BrdDevice = cnf_Open(szBrdName.c_str(), NULL, NULL)) == CNF_ERROR)
   {
      cout << "cnf_Open failed !!" << endl;
ProcessErrorInformation();
   }
   else
   {
```

```
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_OpenBoard_Event();
      }
   }

   /*****************************************************************************
 * GET THE DEVICE COUNTS FOR THE BOARD DEVICE.
 *****************************************************************************/
   if ((cnf_GetDeviceCount(BrdDevice, NULL)) == CNF_ERROR)
   {
      cout << "cnf_GetDeviceCount failed !!" << endl;
ProcessErrorInformation();
      return 0;
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_GetDeviceCount_Event();
      }
   }

   /*****************************************************************************
    * RESET DEVICES ON THE BOARD DEVICE.
    *****************************************************************************/
   if ((cnf_ResetDevices(BrdDevice, NULL, NULL)) == CNF_ERROR)
   {
      cout << "cnf_ResetDevices failed !!" << endl;
ProcessErrorInformation();
      return 0;
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_ResetDevices_Event();
      }
   }

   /*****************************************************************************
    * GET THE DEVICE COUNTS FOR THE BOARD DEVICE.
    *****************************************************************************/
   if ((cnf_GetDeviceCount(BrdDevice, NULL)) == CNF_ERROR)
   {
      cout << "cnf_GetDeviceCount failed !!" << endl;
ProcessErrorInformation();
      return 0;
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
```

```
      else
      {
         Process_GetDeviceCount_Event();
      }
   }

   /*****************************************************************************
    * GET THE DTMF CONTROL INFORMATION FOR THE BOARD DEVICE.
    *****************************************************************************/
   if ((cnf_GetDTMFControl(BrdDevice, NULL)) == CNF_ERROR)
   {
      cout << "cnf_GetDTMFControl failed !!" << endl;
ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_GetDTMFControl_Event();
      }
   }

   /*****************************************************************************
    * SET THE DTMF CONTROL INFORMATION FOR THE BOARD DEVICE.
    *****************************************************************************/
   CNF_DTMF_CONTROL_INFO DTMFControlInfo;
   DTMFControlInfo.unVersion         = CNF_DTMF_CONTROL_INFO_VERSION_0;
   DTMFControlInfo.eDTMFControlState = ECNF_ATTR_STATE_ENABLED;
   DTMFControlInfo.eVolumeUpDigit    = ECNF_DTMF_DIGIT_POUND;
   DTMFControlInfo.eVolumeDownDigit  = ECNF_DTMF_DIGIT_STAR;
   DTMFControlInfo.eVolumeResetDigit = ECNF_DTMF_DIGIT_5;

   if ((cnf_SetDTMFControl(BrdDevice, &DTMFControlInfo, NULL)) == CNF_ERROR)
   {
      cout << "cnf_SetDTMFControl failed !!" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_SetDTMFControl_Event();
      }
   }

   /*****************************************************************************
    * GET THE DTMF CONTROL INFORMATION FOR THE BOARD DEVICE.
    *****************************************************************************/
   if ((cnf_GetDTMFControl(BrdDevice, NULL)) == CNF_ERROR)
   {
      cout << "cnf_GetDTMFControl failed !!" << endl;
      ProcessErrorInformation();

   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
```

```
      }
      else
      {
         Process_GetDTMFControl_Event();
      }
   }

   /*****************************************************************************
    * ENABLE BOARD DEVICE EVENTS.
    *****************************************************************************/
   unsigned int BrdEventList[10];
   BrdEventList[0] = ECNF_BRD_EVT_CONF_OPENED;
   BrdEventList[1] = ECNF_BRD_EVT_CONF_CLOSED;
   BrdEventList[2] = ECNF_BRD_EVT_ACTIVE_TALKER;
   BrdEventList[3] = ECNF_BRD_EVT_PARTY_ADDED;
   BrdEventList[4] = ECNF_BRD_EVT_PARTY_REMOVED;

   CNF_EVENT_INFO BrdEventInfo;
   BrdEventInfo.unEventCount = 5;
   BrdEventInfo.punEventList = BrdEventList;

   if (cnf_EnableEvents(BrdDevice, &BrdEventInfo, (void *)1) == CNF_ERROR)
   {
      cout << "cnf_EnableEvents failed !!" << endl;
      ProcessErrorInformation();
      return 0;
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_EnableEvents_Event();
      }
   }

   /*****************************************************************************
    * GET ATTRIBUTES ON A BOARD DEVICE. FAILURE CASE.
    *****************************************************************************/
   CNF_ATTR BrdAttrList[MAX_CNF_BRD_ATTR];
   CNF_ATTR_INFO BrdAttrInfo;

   BrdAttrList[0].unAttribute = ECNF_CONF_ATTR_DTMF_MASK;
   BrdAttrInfo.unAttrCount = 1;
   BrdAttrInfo.pAttrList = BrdAttrList;

   if (cnf_GetAttributes(BrdDevice, &BrdAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(BrdDevice) << " failed!! - Expected error
due to invalid attribute" << endl;
      ProcessErrorInformation();
   }

   /*****************************************************************************
    * GET ATTRIBUTES ON A BOARD DEVICE.
    *****************************************************************************/
   int nBrdAttr = CNF_BRD_ATTR_BASE;
   {
      for (int i = 0; i < MAX_CNF_BRD_ATTR; i++, nBrdAttr++)
      {
         BrdAttrList[i].unAttribute = nBrdAttr;
      }
   }
```

```
   BrdAttrInfo.unAttrCount = MAX_CNF_BRD_ATTR;
   BrdAttrInfo.pAttrList = BrdAttrList;

   if (cnf_GetAttributes(BrdDevice, &BrdAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(BrdDevice) << " failed!!" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_GetAttributes_Event();
      }
   }

   /*************************************************************************
    * SET ATTRIBUTES ON A BOARD DEVICE.
    *************************************************************************/

   BrdAttrList[0].unAttribute = ECNF_BRD_ATTR_ACTIVE_TALKER;
   BrdAttrList[0].unValue = ECNF_ATTR_STATE_ENABLED;
   BrdAttrList[1].unAttribute = ECNF_BRD_ATTR_NOTIFY_INTERVAL;
   BrdAttrList[1].unValue = 2000; // 2 Second interval for active talker events.
   BrdAttrInfo.unAttrCount = 2;
   BrdAttrInfo.pAttrList = BrdAttrList;

   if (cnf_SetAttributes(BrdDevice, &BrdAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_SetAttributes( ) on " << ATDV_NAMEP(BrdDevice) << " failed!!" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_SetAttributes_Event();
      }
   }

   /*************************************************************************
    * GET ATTRIBUTES ON A BOARD DEVICE.
    *************************************************************************/
   nBrdAttr = CNF_BRD_ATTR_BASE;
   {
      for (int i = 0; i < MAX_CNF_BRD_ATTR; i++, nBrdAttr++)
      {
         BrdAttrList[i].unAttribute = nBrdAttr;
      }
   }

   BrdAttrInfo.unAttrCount = MAX_CNF_BRD_ATTR;
   BrdAttrInfo.pAttrList = BrdAttrList;

   if (cnf_GetAttributes(BrdDevice, &BrdAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(BrdDevice) << " failed!!" << endl;
      ProcessErrorInformation();
//return 0;
```

```
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_GetAttributes_Event();
      }
   }

   /*****************************************************************************
    * OPEN A CONFERENCE DEVICE.
    *****************************************************************************/
if ((CnfDevice = cnf_OpenConference(BrdDevice, NULL, NULL, NULL)) == CNF_ERROR)

{
cout << "cnf_OpenConference failed !!" << endl;
ProcessErrorInformation();
}
   else
   {
      for (int i = 0; i < 2; i++)
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "cnf_OpenConference on " << ATDV_NAMEP(BrdDevice) << " FAILED..." << endl;
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
         }
         else
         {
            Process_OpenConference_Event();
         }
      }
   }

   /*****************************************************************************
    * IF THIS IS A MEDIA CONFERENCE LETS SET THE VIDEO LAYOUT.
    *****************************************************************************/
   if (eType == ECNF_TYPE_MCX)
   {
      if (cnf_GetVideoLayout(CnfDevice, NULL) == CNF_ERROR)
      {
         cout << "cnf_GetVideoLayout failed !!" << endl;
         ProcessErrorInformation();
      }
      else
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
         }
         else
         {
            Process_GetVideoLayout_Event();
         }
      }

      LB_FRAME_HANDLE LayoutHandle = lb_CreateLayoutTemplate(eLB_LAYOUT_TYPE_4_1);
      if (LayoutHandle == MTK_ERROR)
      {
         cout << "lb_CreateLayoutTemplate failed !!" << endl;
      }
      else
      {
```

```
            cout << "Created 4 region layout..." << endl;
         }

         CNF_VIDEO_LAYOUT_INFO LayoutInfo;
         LayoutInfo.eLayoutSize = eLB_LAYOUT_SIZE_CIF;
         LayoutInfo.LayoutHandle = LayoutHandle;

         if (cnf_SetVideoLayout(CnfDevice, &LayoutInfo, NULL) == CNF_ERROR)
         {
            cout << "cnf_SetVideoLayout failed !!" << endl;
            ProcessErrorInformation();
         }
         else
         {
            if (sr_waitevt(10000) == -1)
            {
               cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
            }
            else
            {
               Process_SetVideoLayout_Event();
            }
         }
      }

      /*****************************************************************************
       * ENABLE CONFERENCE DEVICE EVENTS.
       *****************************************************************************/
      unsigned int CnfEventList[10];
      CnfEventList[0] = ECNF_CONF_EVT_PARTY_ADDED;
      CnfEventList[1] = ECNF_CONF_EVT_PARTY_REMOVED;
      CnfEventList[2] = ECNF_CONF_EVT_ACTIVE_TALKER;

      CNF_EVENT_INFO CnfEventInfo;
      CnfEventInfo.unEventCount = 3;
      CnfEventInfo.punEventList = CnfEventList;

      if (cnf_EnableEvents(CnfDevice, &CnfEventInfo, (void *)1) == CNF_ERROR)
      {
         cout << "cnf_EnableEvents failed !!" << endl;
   ProcessErrorInformation();
      }
      else
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
         }
         else
         {
            Process_EnableEvents_Event();
         }
      }

      /*****************************************************************************
       * GET CONFERENCE DEVICE ATTRIBUTES. FAILURE CASE.
       *****************************************************************************/
      CNF_ATTR CnfAttrList[MAX_CNF_CONF_ATTR];
      CNF_ATTR_INFO CnfAttrInfo;

      CnfAttrList[0].unAttribute = ECNF_BRD_ATTR_NOTIFY_INTERVAL;
      CnfAttrInfo.unAttrCount = 1;
      CnfAttrInfo.pAttrList = CnfAttrList;

      if (cnf_GetAttributes(CnfDevice, &CnfAttrInfo, NULL) == CNF_ERROR)
      {
         cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(CnfDevice) << " failed!! - Expected error
```

```
due to invalid attribute" << endl;
      ProcessErrorInformation();
   }

   /*****************************************************************************
    * GET CONFERENCE DEVICE ATTRIBUTES.
    *****************************************************************************/
   int nCnfAttr = CNF_CONF_ATTR_BASE;
   for (int i = 0; i < MAX_CNF_CONF_ATTR; i++, nCnfAttr++)
   {
      CnfAttrList[i].unAttribute = nCnfAttr;
   }

   CnfAttrInfo.unAttrCount = MAX_CNF_CONF_ATTR;
   CnfAttrInfo.pAttrList = CnfAttrList;

   if (cnf_GetAttributes(CnfDevice, &CnfAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(CnfDevice) << " failed!!" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_GetAttributes_Event();
      }
   }

   /*****************************************************************************
    * SET CONFERENCE DEVICE ATTRIBUTES.
    *****************************************************************************/
   CnfAttrList[0].unAttribute = ECNF_CONF_ATTR_TONE_CLAMPING;
CnfAttrList[0].unValue = ECNF_ATTR_STATE_ENABLED;
   CnfAttrList[1].unAttribute = ECNF_CONF_ATTR_DTMF_MASK;
   CnfAttrList[1].unValue = ECNF_DTMF_MASK_OP_SET | ECNF_DTMF_DIGIT_1 | ECNF_DTMF_DIGIT_2 |
ECNF_DTMF_DIGIT_3 | ECNF_DTMF_DIGIT_4;
CnfAttrInfo.unAttrCount = 2;
CnfAttrInfo.pAttrList = CnfAttrList;

   ///
   // Lets set conference device attributes.
   if (cnf_SetAttributes(CnfDevice, &CnfAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_SetAttributes( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_SetAttributes_Event();
      }
   }

   /*****************************************************************************
    * GET CONFERENCE DEVICE ATTRIBUTES.
    *****************************************************************************/
   nCnfAttr = CNF_CONF_ATTR_BASE;
```

```
   {
      for (int i = 0; i < MAX_CNF_CONF_ATTR; i++, nCnfAttr++)
      {
         CnfAttrList[i].unAttribute = nCnfAttr;
      }
   }

   CnfAttrInfo.unAttrCount = MAX_CNF_CONF_ATTR;
   CnfAttrInfo.pAttrList = CnfAttrList;

   if (cnf_GetAttributes(CnfDevice, &CnfAttrInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(CnfDevice) << " failed!!" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_GetAttributes_Event();
      }
   }

   /*************************************************************************
    * GET LIST OF PARTY'S ADDED TO A CONFERENCE.
    *************************************************************************/
   if (cnf_GetPartyList(CnfDevice, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetPartyList( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_GetPartyList_Event();
      }
   }

   /*************************************************************************
    * OPEN A PARTY DEVICE.
    *************************************************************************/
if ((PtyDevice = cnf_OpenParty(BrdDevice, NULL, NULL, NULL)) == CNF_ERROR)
{
cout << "cnf_OpenParty( ) - failed" << endl;
ProcessErrorInformation();
}
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_OpenParty_Event();
      }
   }
```

```
/*****************************************************************************
 * GET PARTY DEVICE ATTRIBUTES.
 *****************************************************************************/
CNF_ATTR PtyAttrList[MAX_CNF_PTY_ATTR];
int nPtyAttr = CNF_PARTY_ATTR_BASE;

{
   for (int i = 0; i < MAX_CNF_PTY_ATTR; i++, nPtyAttr++)
   {
      PtyAttrList[i].unAttribute = nPtyAttr;
   }
}

CNF_ATTR_INFO PtyAttrInfo;
PtyAttrInfo.unAttrCount = MAX_CNF_PTY_ATTR;
PtyAttrInfo.pAttrList = PtyAttrList;

if (cnf_GetAttributes(PtyDevice, &PtyAttrInfo, NULL) == CNF_ERROR)

{
   cout << "cnf_GetAttributes( ) - failed" << endl;
   ProcessErrorInformation();
}
else
{
   if (sr_waitevt(10000) == -1)
   {
      cout << "sr_waitevt failed - " << ATDV_ERRMSGP(PtyDevice) << endl;
   }
   else
   {

      Process_GetAttributes_Event();
   }
}

/*****************************************************************************
 * SET PARTY DEVICE ATTRIBUTES.
 *****************************************************************************/
PtyAttrList[0].unAttribute = ECNF_PARTY_ATTR_TARIFF_TONE;
PtyAttrList[0].unValue = ECNF_ATTR_STATE_ENABLED;
PtyAttrList[1].unAttribute = ECNF_PARTY_ATTR_COACH;
PtyAttrList[1].unValue = ECNF_ATTR_STATE_ENABLED;

PtyAttrInfo.unAttrCount = 2;
PtyAttrInfo.pAttrList = PtyAttrList;

if (cnf_SetAttributes(PtyDevice, &PtyAttrInfo, NULL) == CNF_ERROR)
{
   cout << "cnf_SetAttributes( ) - failed" << endl;
   ProcessErrorInformation();
}
else
{
   if (sr_waitevt(10000) == -1)
   {
      cout << "sr_waitevt failed - " << ATDV_ERRMSGP(PtyDevice) << endl;
   }
   else
   {
      Process_SetAttributes_Event();
   }
}

/*****************************************************************************
 * ADD A PARTY TO A CONFERENCE.
 *****************************************************************************/
```

```
   CNF_PARTY_INFO PtyInfo;
   PtyInfo.unPartyCount = 1;
   PtyInfo.pPartyList = new SRL_DEVICE_HANDLE[1];
   PtyInfo.pPartyList[0] = PtyDevice;

   if (cnf_AddParty(CnfDevice, &PtyInfo, (void *)&CnfDevice) == CNF_ERROR)
   {
      cout << "cnf_AddParty( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      for (int i = 0; i < 3; i++)
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed" << endl;
         }
         else
         {
            Process_AddParty_Event();
         }
      }
   }

   /*****************************************************************************
    * OPEN MULTIPLE PARTY DEVICES.
 *****************************************************************************/
   const unsigned int unPtyCount = 5;
   SRL_DEVICE_HANDLE * pPtyDeviceList = new SRL_DEVICE_HANDLE[unPtyCount];
{
      for (unsigned int i = 0; i < unPtyCount; i++)
      {
      if ((pPtyDeviceList[i] = cnf_OpenParty(BrdDevice, NULL, NULL, NULL)) == CNF_ERROR)
      {
      cout << "cnf_OpenParty( ) - failed" << endl;
      ProcessErrorInformation();
      }
         else
         {
            if (sr_waitevt(10000) == -1)
            {
               cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
            }
            else
            {
               Process_OpenParty_Event();
            }
         }
      }
   }

   bool bMultiPartyAdded = false;

   /*****************************************************************************
    * ADD MULTIPLE PARTY'S TO A CONFERENCE.
    *****************************************************************************/
   PtyInfo.unPartyCount = unPtyCount;
   PtyInfo.pPartyList = pPtyDeviceList;

   if (cnf_AddParty(CnfDevice, &PtyInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_AddParty( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
```

```
      bMultiPartyAdded = true;
      int nPtyEvtCount = unPtyCount * 2 + 1;
      for (int i = 0; i < nPtyEvtCount; i++)
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
         }
         else
         {
            Process_AddParty_Event();
         }
      }
   }

   /*****************************************************************************
    * GET LIST OF PARTYIES ADDED TO A CONFERENCE.
    *****************************************************************************/
   if (cnf_GetPartyList(CnfDevice, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetPartyList( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_GetPartyList_Event();
      }
   }

   if (eType == ECNF_TYPE_MCX)
   {
      /*****************************************************************************
       * SET LIST OF VISIBLE PARTY'S.
       *****************************************************************************/

      LB_FRAME_HANDLE RegionHandleList[10];
      size_t RegionHandleListSize = 10;
      if (lb_GetRegionList(g_LayoutHandle, RegionHandleList, &RegionHandleListSize) ==
MTK_SUCCESS)
      {
         // We expect to get 4 regions in the list. Lets check...
         if (RegionHandleListSize != 4)
         {
            cout << "Received invalid region handle list size..." << endl;
         }
         else
         {
            CNF_VISIBLE_PARTY_INFO VisiblePartyInfoList[4];

            for (int i = 0; i < RegionHandleListSize; i++)
            {
               VisiblePartyInfoList[i].PartyHandle = pPtyDeviceList[i];
               VisiblePartyInfoList[i].RegionHandle = RegionHandleList[i];
            }

            CNF_VISIBLE_PARTY_LIST VisiblePartyInfo;
            VisiblePartyInfo.unCount = 4;
            VisiblePartyInfo.pPartyList = VisiblePartyInfoList;

            if (cnf_SetVisiblePartyList(CnfDevice, &VisiblePartyInfo, NULL) == CNF_ERROR)
            {
```

```
            cout << "cnf_SetVisiblePartyList( ) - failed" << endl;
            ProcessErrorInformation();
         }
         else
         {
            if (sr_waitevt(10000) == -1)
            {
               cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
            }
            else
            {
               Process_SetVisiblePartyList_Event();
            }
         }
      }
   }
   else
   {
      cout << "lb_GetRegionList( ) - failed" << endl;
   }

   /*****************************************************************************
    * GET LIST OF VISIBLE PARTY'S.
    *****************************************************************************/
   if (cnf_GetVisiblePartyList(CnfDevice, NULL) == CNF_ERROR)
   {
      cout << "cnf_GetVisiblePartyList( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_GetVisiblePartyList_Event();
      }
   }
}

/*****************************************************************************
 * GET PARTY DEVICE ATTRIBUTES.
 *****************************************************************************/

nPtyAttr = CNF_PARTY_ATTR_BASE;
{
   for (int i = 0; i < MAX_CNF_PTY_ATTR; i++, nPtyAttr++)
   {
      PtyAttrList[i].unAttribute = nPtyAttr;
   }
}

PtyAttrInfo.unAttrCount = MAX_CNF_PTY_ATTR;
PtyAttrInfo.pAttrList = PtyAttrList;

if (cnf_GetAttributes(PtyDevice, &PtyAttrInfo, NULL) == CNF_ERROR)

{
   cout << "cnf_GetAttributes( ) - failed" << endl;
   ProcessErrorInformation();
}
else
{
   if (sr_waitevt(10000) == -1)
   {
```

```
      cout << "sr_waitevt failed - " << ATDV_ERRMSGP(PtyDevice) << endl;
   }
   else
   {
      Process_GetAttributes_Event();
   }
}
/*****************************************************************************
 * SET PARTY DEVICE ATTRIBUTES.
 *****************************************************************************/

PtyAttrList[0].unAttribute = ECNF_PARTY_ATTR_TARIFF_TONE;
PtyAttrList[0].unValue = ECNF_ATTR_STATE_DISABLED;
PtyAttrList[1].unAttribute = ECNF_PARTY_ATTR_COACH;
PtyAttrList[1].unValue = ECNF_ATTR_STATE_DISABLED;

PtyAttrInfo.unAttrCount = 2;
PtyAttrInfo.pAttrList = PtyAttrList;

if (cnf_SetAttributes(PtyDevice, &PtyAttrInfo, NULL) == CNF_ERROR)
{
   cout << "cnf_SetAttributes( ) - failed" << endl;
   ProcessErrorInformation();
}
else
{
   if (sr_waitevt(10000) == -1)
   {
      cout << "sr_waitevt failed - " << ATDV_ERRMSGP(PtyDevice) << endl;
   }
   else
   {
      Process_SetAttributes_Event();
   }
}

/*****************************************************************************
 * GET PARTY DEVICE ATTRIBUTES.
 *****************************************************************************/

nPtyAttr = CNF_PARTY_ATTR_BASE;

{
   for (int i = 0; i < MAX_CNF_PTY_ATTR; i++, nPtyAttr++)
   {
      PtyAttrList[i].unAttribute = nPtyAttr;
   }
}

PtyAttrInfo.unAttrCount = MAX_CNF_PTY_ATTR;
PtyAttrInfo.pAttrList = PtyAttrList;

if (cnf_GetAttributes(PtyDevice, &PtyAttrInfo, NULL) == CNF_ERROR)
{
   cout << "cnf_GetAttributes( ) - failed" << endl;
   ProcessErrorInformation();
}
else
{
   if (sr_waitevt(10000) == -1)
   {
      cout << "sr_waitevt failed - " << ATDV_ERRMSGP(PtyDevice) << endl;
   }
   else
   {
      Process_GetAttributes_Event();
   }
```

```
   }

   /*****************************************************************************
    * REMOVE PARTY FROM A CONFERENCE.
    *
    * NOTE: SINCE WE HAVE ENABLED THE PARTY REMOVED EVENT ON BOTH THE BOARD AND
    *       CONFERENCE DEVICES, WE SHOULD EXPECT TO GET THE CNFEV_PARTY_REMOVED
    *       NOTIFICATION EVENT ON BOTH THE BOARD AND CONFERENCE DEVICE HANDLES,
    *       IN ADDITION TO THE CNFEV_REMOVE_PARTY TERMINATION EVENT.
 *****************************************************************************/
if (cnf_RemoveParty(CnfDevice, &PtyInfo, NULL) == CNF_ERROR)
   {
      cout << "cnf_RemoveParty( ) - failed" << endl;
      ProcessErrorInformation();
   }
   else
   {
      for (int i = 0; i < 3; i++)
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
         }
         else
         {
            Process_RemoveParty_Event();
         }
      }
   }

   /*****************************************************************************
    * CLOSE MULTIPLE PARTY DEVICE.
    *****************************************************************************/
{
      for (unsigned int i = 0; i < unPtyCount; i++)
      {
      if (cnf_CloseParty(pPtyDeviceList[i], NULL) == CNF_ERROR)
      {
      cout << "cnf_CloseParty( ) - failed" << endl << endl;
      ProcessErrorInformation();
      }
         else
         {
            cout << "cnf_CloseParty( ) - successful" << endl << endl;
            if (bMultiPartyAdded == true)
            {
               for (int i = 0; i < 2; i++)
               {
                  if (sr_waitevt(10000) == -1)
                  {
                     cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
                  }
                  else
                  {
                     Process_RemoveParty_Event();
                  }
               }
            }
         }
      }
   }

   /*****************************************************************************
    * DISABLE CONFERENCE DEVICE EVENTS.
```

```
 ****************************************************************************/

   if (cnf_DisableEvents(CnfDevice, &CnfEventInfo, (void *)1) == CNF_ERROR)
   {
      cout << "cnf_DisableEvents failed !!" << endl;
ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl;
      }
      else
      {
         Process_DisableEvents_Event();
      }
   }

   /*****************************************************************************
 * CLOSE A CONFERENCE DEVICE.
 *****************************************************************************/
   if (cnf_CloseConference(CnfDevice, NULL) == CNF_ERROR)
{
      cout << "cnf_CloseConference( ) for " << ATDV_NAMEP(CnfDevice) << " FAILED" << endl;
      cout << "\tError - " << ATDV_LASTERR(CnfDevice) << endl;
}
   else
   {
      for (int i = 0; i < 1; i++)
      {
         if (sr_waitevt(10000) == -1)
         {
            cout << "sr_waitevt failed - " << ATDV_ERRMSGP(CnfDevice) << endl << endl;
         }
         else
         {
            Process_Board_Event();
         }
      }


      cout << "cnf_CloseConference( ) - successful" << endl << endl;
   }

   /*****************************************************************************
    * CLOSE A PARTY DEVICE.
    *****************************************************************************/
   if (cnf_CloseParty(PtyDevice, NULL) == CNF_ERROR)
{
cout << "cnf_CloseParty failed !!" << endl << endl;
ProcessErrorInformation();
}
   else
   {
      cout << "cnf_CloseParty( ) - successful !!" << endl << endl;
   }

   /*****************************************************************************
    * DISABLE BOARD DEVICE EVENTS.
    *****************************************************************************/
   unsigned int BrdDisableEventList[10];
   BrdDisableEventList[0] = ECNF_BRD_EVT_CONF_OPENED;
   BrdDisableEventList[1] = ECNF_BRD_EVT_CONF_CLOSED;
   BrdDisableEventList[2] = ECNF_BRD_EVT_ACTIVE_TALKER;
   BrdDisableEventList[3] = ECNF_BRD_EVT_PARTY_ADDED;
   BrdDisableEventList[4] = ECNF_BRD_EVT_PARTY_REMOVED;
```

```
   BrdDisableEventList[5] = ECNF_CONF_ATTR_TONE_CLAMPING;

   CNF_EVENT_INFO BrdDisableEventInfo;
   BrdDisableEventInfo.unEventCount = 5;
   BrdDisableEventInfo.punEventList = &BrdDisableEventList[0];

   if (cnf_DisableEvents(BrdDevice, &BrdDisableEventInfo, (void *)1) == CNF_ERROR)
   {
      cout << "cnf_DisableEvents failed !!" << endl;
ProcessErrorInformation();
   }
   else
   {
      if (sr_waitevt(10000) == -1)
      {
         cout << "sr_waitevt failed - " << ATDV_ERRMSGP(BrdDevice) << endl;
      }
      else
      {
         Process_DisableEvents_Event();
      }
   }

   /*****************************************************************************
    * CLOSE THE BOARD DEVICE.
    *****************************************************************************/
   if (cnf_Close(BrdDevice, NULL) == CNF_ERROR)
   {
      cout << "cnf_Close failed !!" << endl << endl;
ProcessErrorInformation();

   }
   else
   {
      cout << "cnf_Close( ) - Successful" << endl << endl;
   }

   return 0;
}


/**
 * @fn Process_DisableEvents_Event
 */
void Process_DisableEvents_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_EVENT_INFO pInfo = (PCNF_EVENT_INFO) Data.pEventData;

   switch (Data.EventType)
   {
      case CNFEV_DISABLE_EVENT:
      {
         cout << "cnf_DisableEvents on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
 endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t      Event Count: " << pInfo->unEventCount << endl;
         for (int i = 0; i < pInfo->unEventCount; i++)
         {
            cout << "\t            Event: " << pInfo->punEventList[i] << endl;
         }
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;
```

```
      case CNFEV_DISABLE_EVENT_FAIL:
      {
         cout << "cnf_DisableEvents( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
 endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;

         cout << "\t      Event Count: " << pInfo->unEventCount << endl;
         for (int i = 0; i < pInfo->unEventCount; i++)
         {
            cout << "\t            Event: " << pInfo->punEventList[i] << endl;
         }
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;


      default:
      {
         cout << "Process_DisableEvents_Event - UNEXPECTED EVENT" << endl;
         cout << "\t            Event: " << Data.EventType << endl;
         cout << "\t     Event Device: " << Data.EventDevice << endl;
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;
   };
}


/**
 * @fn ProcessEnableEventsEvent
 */
void Process_EnableEvents_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_EVENT_INFO pInfo = (PCNF_EVENT_INFO) Data.pEventData;

   if (Data.EventType == CNFEV_ENABLE_EVENT)
   {
      cout << "cnf_EnableEvents on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" << endl;
      cout << "\tReceived following event information:" << endl;
      cout << "\t            Event: " << Data.EventType << endl;
      cout << "\t       Event Data: " << Data.pEventData << endl;
      if (pInfo)
      {
         cout << "\t      Event Count: " << pInfo->unEventCount << endl;
         for (int i = 0; i < pInfo->unEventCount; i++)
         {
            cout << "\t            Event: " << pInfo->punEventList[i] << endl;
         }
      }
      cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
      cout << "\t     Event Device: " << Data.EventDevice << endl;
      cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
   }
   else
   {
      cout << "cnf_EnableEvents on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" << endl;
      cout << "\tEvent: " << Data.EventType << endl;
      cout << "\t      Event Count: " << pInfo->unEventCount << endl;
      for (int i = 0; i < pInfo->unEventCount; i++)
      {
         cout << "\t            Event: " << pInfo->punEventList[i] << endl;
      }
      cout << endl;
```

```
   }
}

/**
 * @fn srl_GetMetaEvent
 */
void srl_GetMetaEvent(PSRL_METAEVENT a_pMetaEvent)
{
   a_pMetaEvent->EventType       = sr_getevttype();
a_pMetaEvent->EventDevice     = sr_getevtdev();
a_pMetaEvent->EventDataLength = sr_getevtlen();
a_pMetaEvent->pEventData      = sr_getevtdatap();
a_pMetaEvent->pEventUserInfo  = sr_getUserContext();
}


/**
 * @fn ProcessErrorInfo
 */
void ProcessErrorInformation()
{
PCNF_ERROR_INFO pErrorInfo = new CNF_ERROR_INFO;
   if (cnf_GetErrorInfo(pErrorInfo) == CNF_ERROR)
   {
      cout << "cnf_GetErrorInfo() FAILED!!" << endl;
   }
   else
   {
      cout << "\t     Error Code: " << pErrorInfo->unErrorCode << endl;
      cout << "\t   Error String: " << pErrorInfo->szErrorString << endl;
      cout << "\tAdditional Info: " << pErrorInfo->szAdditionalInfo << endl << endl;
   }
}


/**
 * @fn Process_AddParty_Event
 */
void Process_AddParty_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch (Data.EventType)
   {
      case CNFEV_ADD_PARTY:
      {
         PCNF_PARTY_INFO pInfo = (PCNF_PARTY_INFO) Data.pEventData;
         cout << "cnf_AddParty( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" << endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t     Party Handle: " << pInfo->pPartyList[i] << endl;
         }
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_PARTY_ADDED:
      {
         PCNF_PARTY_ADDED_EVENT_INFO pInfo = (PCNF_PARTY_ADDED_EVENT_INFO) Data.pEventData;
         cout << "Received PARTY ADDED notification event..." << endl;
         cout << "\tConference Handle: " << pInfo->ConfHandle << endl;
         cout << "\t  Conference Name: " << pInfo->szConfName << endl;
         cout << "\t     Party Handle: " << pInfo->PartyHandle << endl;
         cout << "\t       Party Name: " << pInfo->szPartyName << endl;
```

```
         cout << "\t     Event Device: " << Data.EventDevice << endl << endl;
      }
      break;

      case CNFEV_ADD_PARTY_FAIL:
      {
         PCNF_PARTY_INFO pInfo = (PCNF_PARTY_INFO) Data.pEventData;
         cout << "cnf_AddParty( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" << endl;
         cout << "\tError - " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t     Party Handle: " << pInfo->pPartyList[i] << endl;
         }
         cout << endl;
      }
      break;

      default:
      {
         cout << "Process_AddParty_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   }
}

void Process_Board_Event()
{
   SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   if (Data.EventType == CNFEV_CONF_CLOSED)
   {
      PCNF_CONF_CLOSED_EVENT_INFO pInfo = (PCNF_CONF_CLOSED_EVENT_INFO) Data.pEventData;
      cout << "Received CONFERENCE CLOSED notification event..." << endl;
      cout << "\t  Conference Name: " << pInfo->szConfName << endl;
      cout << "\t     Event Device: " << Data.EventDevice << endl << endl;
   }
   else
   {
      //ProcessRemovePartyEvent();
   }
}


void Process_GetActiveTalkerList_Event()
{
   SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_ACTIVE_TALKER_INFO pInfo = (PCNF_ACTIVE_TALKER_INFO) Data.pEventData;

   if (Data.EventType == CNFEV_GET_ACTIVE_TALKER)
   {
      cout << "cnf_GetActiveTalkerList( ) - Successful" << endl;
      cout << "\tReceived following event information:" << endl;
      cout << "\t            Event: " << Data.EventType << endl;
      if (pInfo)
      {
         cout << "\t       Event Data: " << pInfo << endl;
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t    Party Info: Party[" << i << "] - Handle[" << pInfo->pPartyList[i] <<
"] - Device Name[" << ATDV_NAMEP(pInfo->pPartyList[i]) << "]" << endl;
```

```
         }
      }
      else
      {
         cout << "\t INVALID PINFO POINTER..." << endl;
      }
      cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
      cout << "\t     Event Device: " << Data.EventDevice << endl;
      cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
   }
   else
   {
      cout << "cnf_GetActiveTalkerList( ) - Failed" << endl;
      cout << "\tEvent: " << Data.EventType << endl << endl;
   }
}


/**
 * @fn Process_GetAttributes_Event
 */
void Process_GetAttributes_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_ATTR_INFO pInfo = (PCNF_ATTR_INFO) Data.pEventData;

switch (Data.EventType)
{
case CNFEV_GET_ATTRIBUTE:
{
cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL " << endl;
      cout << "\tReceived following event information:" << endl;
      if (pInfo)
      {
         cout << "\t  Attribute Count: " << pInfo->unAttrCount << endl;
         for (int i = 0; i < pInfo->unAttrCount; i++)
         {
            cout << "\t   Attribute Info: Attribute[" << pInfo->pAttrList[i].unAttribute << "]
Value[0x" << hex << pInfo->pAttrList[i].unValue << dec << "]" << endl;
         }
      }
      else
      {
         cout << "\t INVALID DATA POINTER..." << endl;
      }
      cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
      cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
}
break;

case CNFEV_GET_ATTRIBUTE_FAIL:
{
cout << "cnf_GetAttributes( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" << endl;
         cout << "\tError - " << ATDV_LASTERR(Data.EventDevice) << endl;

      if (pInfo)
      {
         cout << "\t  Attribute Count: " << pInfo->unAttrCount << endl;
         for (int i = 0; i < pInfo->unAttrCount; i++)
         {
            cout << "\t   Attribute Info: Attribute[" << pInfo->pAttrList[i].unAttribute << "]
Value[" << pInfo->pAttrList[i].unValue << "]" << endl;
         }
      }
      else
```

```
      {
         cout << "\t INVALID DATA POINTER..." << endl;
      }
}
break;

default:
{
cout << "Process_GetAttributes_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
}
break;
}
}


/**
 * @fn Process_GetDeviceCount_Event
 */
void Process_GetDeviceCount_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_DEVICE_COUNT_INFO pInfo = (PCNF_DEVICE_COUNT_INFO) Data.pEventData;

   switch (Data.EventType)
   {
      case CNFEV_GET_DEVICE_COUNT:
      {
         cout << "cnf_GetDeviceCount( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL "
<< endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t             Event Data: " << Data.pEventData << endl;
         cout << "\t     Free Party Devices: " << pInfo->unFreePartyCount << endl;
         cout << "\tFree Conference Devices: " << pInfo->unFreeConfCount << endl;
         cout << "\t      Max Party Devices: " << pInfo->unMaxPartyCount << endl;
         cout << "\t Max Conference Devices: " << pInfo->unMaxConfCount << endl;
         cout << "\t      Event Data Length: " << Data.EventDataLength << endl;
         cout << "\t           Event Device: " << Data.EventDevice << endl;
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_GET_DEVICE_COUNT_FAIL:
      {
         cout << "cnf_GetDeviceCount( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         cout << "\tError - " << ATDV_LASTERR(Data.EventDevice) << endl;
      }
      break;

      default:
      {
         cout << "Process_GetDeviceCount_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   }
}


/**
 * @fn Process_GetDTMFControl_Event
 */
```

```
void Process_GetDTMFControl_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_DTMF_CONTROL_INFO pInfo = (PCNF_DTMF_CONTROL_INFO) Data.pEventData;

   switch (Data.EventType)
   {
      case CNFEV_GET_DTMF_CONTROL:
         cout << "cnf_GetDTMFControl( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t                  Event: " << Data.EventType << endl;
         cout << "\t             Event Data: " << Data.pEventData << endl;
         cout << "\t     DTMF Control State: " << pInfo->eDTMFControlState << endl;
         cout << "\t        Volume Up Digit: " << pInfo->eVolumeUpDigit << endl;
         cout << "\t      Volume Down Digit: " << pInfo->eVolumeDownDigit << endl;
         cout << "\t     Volume Reset Digit: " << pInfo->eVolumeResetDigit << endl;
         cout << "\t      Event Data Length: " << Data.EventDataLength << endl;
         cout << "\t           Event Device: " << Data.EventDevice << endl;
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;

      case CNFEV_GET_DTMF_CONTROL_FAIL:
         cout << "cnf_GetDTMFControl( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:
         cout << "Process_GetDTMFControl_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
         break;
   }
}


/**
 * @fn Process_GetPartyList_Event
 */
void Process_GetPartyList_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);
   PCNF_PARTY_INFO pInfo = (PCNF_PARTY_INFO) Data.pEventData;

   if (Data.EventType == CNFEV_GET_PARTY_LIST)
   {
      cout << "cnf_GetPartyList( ) - Successful" << endl;
      cout << "\tReceived following event information:" << endl;
      cout << "\t            Event: " << Data.EventType << endl;
      if (pInfo)
      {
         cout << "\t       Event Data: " << pInfo << endl;
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t    Party Info: Party[" << i << "] - Handle[" << pInfo->pPartyList[i] <<
"] - Device Name[" << ATDV_NAMEP(pInfo->pPartyList[i]) << "]" << endl;
         }
      }
      else
      {
         cout << "\t INVALID PINFO POINTER..." << endl;
```

```
      }
      cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
      cout << "\t     Event Device: " << Data.EventDevice << endl;
      cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
   }
   else
   {
      cout << "cnf_GetPartyList( ) - Failed" << endl;
      cout << "\tEvent: " << Data.EventType << endl;
   }
}


/**
 * @fn ProcessMetaEvent
 */
void ProcessMetaEvent(char * a_szString)
{
SRL_METAEVENT MetaData;
srl_GetMetaEvent(&MetaData);
   cout << a_szString << endl;
cout << "\tReceived following event information:" << endl;
cout << "\t            Event: " << MetaData.EventType << endl;
cout << "\t       Event Data: " << MetaData.pEventData << endl;
cout << "\tEvent Data Length: " << MetaData.EventDataLength << endl;
cout << "\t     Event Device: " << MetaData.EventDevice << endl;
cout << "\t  Event User Info: " << MetaData.pEventUserInfo << endl << endl;
}

/**
 * @fn Process_OpenBoard_Event
 */
void Process_OpenBoard_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch (Data.EventType)
   {
      case CNFEV_OPEN:
      {
         cout << "cnf_Open( ) - Successful" << endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t       Event Data: " << Data.pEventData << endl;
         cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
         cout << "\t     Event Device: " << Data.EventDevice << endl;
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      default:
      {
         cout << "cnf_Open( ) - Failed" << endl;
         cout << "\tEvent: " << Data.EventType << endl;
      }
      break;
   };
}

void Process_OpenConference_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch (Data.EventType)
   {
      case CNFEV_OPEN_CONF:
```

```
      {
         PCNF_OPEN_CONF_RESULT pInfo = (PCNF_OPEN_CONF_RESULT) Data.pEventData;
         cout << "cnf_OpenConference( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL "
<< endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\tConference Device: " << pInfo->ConfHandle << endl;
         cout << "\t  Conference Name: " << pInfo->szConfName << endl;
         cout << "\t     Event Device: " << Data.EventDevice << endl;
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_CONF_OPENED:
      {
         PCNF_CONF_OPENED_EVENT_INFO pInfo = (PCNF_CONF_OPENED_EVENT_INFO) Data.pEventData;
         cout << "Received CONFERENCE OPENED notification event..." << endl;
         cout << "\tConference Handle: " << pInfo->ConfHandle << endl;
         cout << "\t  Conference Name: " << pInfo->szConfName << endl;
         cout << "\t     Event Device: " << Data.EventDevice << endl << endl;
      }
      break;

      case CNFEV_OPEN_CONF_FAIL:
      {
         PCNF_OPEN_CONF_RESULT pInfo = (PCNF_OPEN_CONF_RESULT) Data.pEventData;
         cout << "cnf_OpenConference( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED " <<
endl;
         cout << "\tConference Device: " << pInfo->ConfHandle << endl << endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         cnf_CloseConference(pInfo->ConfHandle, NULL);
      }
      break;

      default:
      {
         cout << "Process_OpenConference_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   };
}



void Process_OpenParty_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch(Data.EventType)
   {
      case CNFEV_OPEN_PARTY:
      {
         PCNF_OPEN_PARTY_RESULT pInfo = (PCNF_OPEN_PARTY_RESULT) Data.pEventData;
         cout << "cnf_OpenParty( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL " <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t   Party Device: " << pInfo->PartyHandle << endl;
         cout << "\t     Party Name: " << pInfo->szPartyName << endl;
         cout << "\t   Event Device: " << Data.EventDevice << endl;
         cout << "\tEvent User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_OPEN_PARTY_FAIL:
```

```
      {
         PCNF_OPEN_PARTY_RESULT pInfo = (PCNF_OPEN_PARTY_RESULT) Data.pEventData;
         cout << "cnf_OpenParty( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED " << endl;
         cout << "\tParty Device: " << pInfo->PartyHandle << endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         cnf_CloseParty(pInfo->PartyHandle, NULL);
      }
      break;

      default:
      {
         cout << "Process_OpenParty_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   };
}


/**
 * @fn Process_RemoveParty_Event
 */
void Process_RemoveParty_Event()
{
SRL_METAEVENT Data;

srl_GetMetaEvent(&Data);

   switch (Data.EventType)
   {
      case CNFEV_REMOVE_PARTY:
      {
         PCNF_PARTY_INFO pInfo = (PCNF_PARTY_INFO) Data.pEventData;
         cout << "cnf_RemoveParty( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t     Party Handle: " << pInfo->pPartyList[i] << endl;
         }
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;

      }

      break;


      case CNFEV_PARTY_REMOVED:
      {
         PCNF_PARTY_REMOVED_EVENT_INFO pInfo = (PCNF_PARTY_REMOVED_EVENT_INFO) Data.pEventData;
         cout << "Received PARTY REMOVED notification event..." << endl;
         cout << "\tConference Handle: " << pInfo->ConfHandle << endl;
         cout << "\t  Conference Name: " << pInfo->szConfName << endl;
         cout << "\t     Party Handle: " << pInfo->PartyHandle << endl;
         cout << "\t       Party Name: " << pInfo->szPartyName << endl;
         cout << "\t     Event Device: " << Data.EventDevice << endl << endl;
      }
      break;

      default:
      {
         PCNF_PARTY_INFO pInfo = (PCNF_PARTY_INFO) Data.pEventData;
         cout << "cnf_RemoveParty( ) - Failed" << endl;
         cout << "\tEvent: " << Data.EventType << endl;
```

```
         cout << "\t      Party Count: " << pInfo->unPartyCount << endl;
         for (int i = 0; i < pInfo->unPartyCount; i++)
         {
            cout << "\t     Party Handle: " << pInfo->pPartyList[i] << endl;
         }
         cout << endl;
      }
      break;
   }
}


/**
 * @fn Process_ResetDevices_Event
 */
void Process_ResetDevices_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch (Data.EventType)
   {
      case CNFEV_RESET_DEVICES:
      {
         cout << "cnf_ResetDevices( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL " <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t       Event Data: " << Data.pEventData << endl;
         cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_RESET_DEVICES_FAIL:
      {
         cout << "cnf_ResetDevices( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED " <<
endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
      }
      break;

      default:
      {
         cout << "Process_ResetDevices_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   };
}


/**
 * @fn Process_SetAttributes_Event
 */
void Process_SetAttributes_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);


   PCNF_ATTR_INFO pInfo = (PCNF_ATTR_INFO) Data.pEventData;

   switch (Data.EventType)
   {
      case CNFEV_SET_ATTRIBUTE:
```

```
      {
         cout << "cnf_SetAttributes( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         if (pInfo)
         {
            cout << "\t  Attribute Count: " << pInfo->unAttrCount << endl;
            PCNF_ATTR pAttrList = pInfo->pAttrList;
            if (pAttrList)
            {
               for (int i = 0; i < pInfo->unAttrCount; i++)
               {
                  cout << "\t   Attribute Info: Attribute[" << pAttrList[i].unAttribute << "]
Value[0x" << hex << pAttrList[i].unValue << dec << "]" << endl;
               }
            }
            else
            {
               cout << "\t INVALID ATTRIBUTE LIST POINTER..." << endl;
            }

         }
         else
         {
            cout << "\t INVALID PINFO POINTER..." << endl;
         }
         cout << "\tEvent Data Length: " << Data.EventDataLength << endl;
         cout << "\t  Event User Info: " << Data.pEventUserInfo << endl << endl;
      }
      break;

      case CNFEV_SET_ATTRIBUTE_FAIL:
      {
         cout << "cnf_SetAttributes( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         if (pInfo)
         {
            cout << "\t  Attribute Count: " << pInfo->unAttrCount << endl;
            PCNF_ATTR pAttrList = pInfo->pAttrList;

            if (pAttrList)
            {
               for (int i = 0; i < pInfo->unAttrCount; i++)
               {
                  cout << "\t   Attribute Info: Attribute[" << pAttrList[i].unAttribute << "]
Value[" << pAttrList[i].unValue << "]" << endl;
               }
            }
            else
            {
               cout << "\t INVALID ATTRIBUTE LIST POINTER..." << endl;
            }
         }
         else
         {
            cout << "\t INVALID PINFO POINTER..." << endl;
         }
      }
      break;

      default:
      {
```

```
         cout << "Process_SetAttributes_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
      }
      break;
   };
}


/**
 * @fn Process_SetDTMFControl_Event
 */
void Process_SetDTMFControl_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   switch(Data.EventType)
   {
      case CNFEV_SET_DTMF_CONTROL:
         cout << "cnf_SetDTMFControl( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t             Event Data: " << Data.pEventData << endl;
         cout << "\t      Event Data Length: " << Data.EventDataLength << endl;
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;

      case CNFEV_SET_DTMF_CONTROL_FAIL:
         cout << "cnf_SetDTMFControl( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:

         cout << "Process_SetDTMFControl_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
         break;
   }
}


/**
 * @fn Process_GetVideoLayout_Event
 */
void Process_GetVideoLayout_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_VIDEO_LAYOUT_INFO pInfo = (PCNF_VIDEO_LAYOUT_INFO) Data.pEventData;

   switch(Data.EventType)
   {
      case CNFEV_GET_VIDEO_LAYOUT:
         cout << "cnf_GetVideoLayout( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         if (pInfo)
         {
            cout << "\t          Layout Handle: " << pInfo->LayoutHandle << endl;
            cout << "\t            Layout Size: " << pInfo->eLayoutSize << endl;
            g_LayoutHandle = pInfo->LayoutHandle;
            eLB_LAYOUT_TYPE eType;
```

```
            if (lb_GetType(pInfo->LayoutHandle, &eType) == MTK_SUCCESS)
            {
                cout << "\t            Layout Type: " << eType << endl;
            }
         }
         else
         {
            cout << "\tInvalid pINFO pointer." << endl;
         }
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;


      case CNFEV_GET_VIDEO_LAYOUT_FAIL:
         cout << "cnf_GetVideoLayout( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:
         cout << "Process_GetVideoLayout_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
         break;
   }
}


/**
 * @fn Process_SetVideoLayout_Event
 */
void Process_SetVideoLayout_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_VIDEO_LAYOUT_INFO pInfo = (PCNF_VIDEO_LAYOUT_INFO) Data.pEventData;

   switch(Data.EventType)
   {
      case CNFEV_SET_VIDEO_LAYOUT:
         cout << "cnf_SetVideoLayout( ) on " << ATDV_NAMEP(Data.EventDevice) << " SUCCESSFUL" <<
endl;
         cout << "\tReceived following event information:" << endl;
         cout << "\t          Layout Handle: " << pInfo->LayoutHandle << endl;
         cout << "\t            Layout Size: " << pInfo->eLayoutSize << endl;
         g_LayoutHandle = pInfo->LayoutHandle;
         eLB_LAYOUT_TYPE eType;
         if (lb_GetType(pInfo->LayoutHandle, &eType) == MTK_SUCCESS)
         {
            cout << "\t            Layout Type: " << eType << endl;
         }
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;

      case CNFEV_SET_VIDEO_LAYOUT_FAIL:
         cout << "cnf_SetVideoLayout( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED" <<
endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:
         cout << "Process_SetVideoLayout_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
```

```
         break;
   }
}


/**
 * @fn Process_GetVisiblePartyList_Event
 */
void Process_GetVisiblePartyList_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_VISIBLE_PARTY_LIST pInfo = (PCNF_VISIBLE_PARTY_LIST) Data.pEventData;

   switch(Data.EventType)
   {
      case CNFEV_GET_VISIBLE_PARTY_LIST:
         cout << "cnf_GetVisiblePartyList( ) on " << ATDV_NAMEP(Data.EventDevice) << "
SUCCESSFUL" << endl;
         cout << "\tReceived following event information:" << endl;
         if (pInfo)
         {
            for (unsigned int i = 0; i < pInfo->unCount; i++)
            {
               PCNF_VISIBLE_PARTY_INFO pVPI = &(pInfo->pPartyList[i]);
               cout << "\tVisiblePartyList[" << i << "] --- Party Handle: " << pVPI->PartyHandle
<< " Region Handle: " << pVPI->RegionHandle << endl;
            }
         }
         else
         {
            cout << "Received invalid data pointer..." << endl;
         }
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;

      case CNFEV_GET_VISIBLE_PARTY_LIST_FAIL:
         cout << "cnf_GetVisiblePartyList( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED"
<< endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:
         cout << "Process_GetVisiblePartyList_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
         break;
   }
}


/**
 * @fn Process_SetVisiblePartyList_Event
 */
void Process_SetVisiblePartyList_Event()
{
SRL_METAEVENT Data;
srl_GetMetaEvent(&Data);

   PCNF_VISIBLE_PARTY_LIST pInfo = (PCNF_VISIBLE_PARTY_LIST) Data.pEventData;

   switch(Data.EventType)
   {
      case CNFEV_SET_VISIBLE_PARTY_LIST:
         cout << "cnf_SetVisiblePartyList( ) on " << ATDV_NAMEP(Data.EventDevice) << "
```

```
SUCCESSFUL" << endl;
         cout << "\tReceived following event information:" << endl;
         if (pInfo)
         {
            for (unsigned int i = 0; i < pInfo->unCount; i++)
            {
               PCNF_VISIBLE_PARTY_INFO pVPI = &(pInfo->pPartyList[i]);
               cout << "\tVisiblePartyList[" << i << "] --- Party Handle: " << pVPI->PartyHandle
<< " Region Handle: " << pVPI->RegionHandle << endl;
            }
         }
         else
         {
            cout << "Received invalid data pointer..." << endl;
         }
         cout << "\t        Event User Info: " << Data.pEventUserInfo << endl << endl;
         break;

      case CNFEV_SET_VISIBLE_PARTY_LIST_FAIL:
         cout << "cnf_SetVisiblePartyList( ) on " << ATDV_NAMEP(Data.EventDevice) << " FAILED"
<< endl;
         cout << "ATDV_LASTERR : " << ATDV_LASTERR(Data.EventDevice) << endl;
         cout << "ATDV_ERRMSGP : " << ATDV_ERRMSGP(Data.EventDevice) << endl;
         break;

      default:
         cout << "Process_SetVisiblePartyList_Event() - Unexpected event" << endl;
         cout << "\t       Event: " << Data.EventType << endl;
         cout << "\tEvent Device: " << Data.EventDevice << endl << endl;
         break;
   }
}
```

Figure 2. Conferencing (CNF) Example Code Output

```
Conferencing (CNF) Example Code
=======================================

Board Name is: cnfB1

cnf_Open failure!! : Expected failure due to the following Error Code: 4  Error String: Invalid
parameter in function call

Additional Info: Invalid parameter - a_szBrdName is NULL

cnf_Open failure!! : Expected failure due to the following Error Code: 3Error String: Invalid
device name provided by user

Additional Info: Invalid device name [blah_blah] specified

cnf_Open( ) - Successful

Received following event information:
       Event Data: 0
       Event Data Length: 10
       Event Device: 1
       Event User Info: 0

cnf_GetDeviceCount( ) on cnfB1 SUCCESSFUL
Received following event information:
             Event Data: 0x8723e68
     Free Party Devices: 60
Free Conference Devices: 30
      Max Party Devices: 60
 Max Conference Devices: 30
```

```
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_ResetDevices( ) on cnfB1 SUCCESSFUL
Received following event information:
       Event Data: 0
Event Data Length: 10
  Event User Info: 0

cnf_GetDeviceCount( ) on cnfB1 SUCCESSFUL
Received following event information:
             Event Data: 0x873a1f8
     Free Party Devices: 60
Free Conference Devices: 30
      Max Party Devices: 60
 Max Conference Devices: 30
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_GetDTMFControl( ) on cnfB1 SUCCESSFUL
Received following event information:
                  Event: 49164
             Event Data: 0x873a188
     DTMF Control State: 1
        Volume Up Digit: 2048
      Volume Down Digit: 1024
     Volume Reset Digit: 16
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_SetDTMFControl( ) on cnfB1 SUCCESSFUL
Received following event information:
             Event Data: 0
      Event Data Length: 10
        Event User Info: 0

cnf_GetDTMFControl( ) on cnfB1 SUCCESSFUL
Received following event information:
                  Event: 49164
             Event Data: 0x873a188
     DTMF Control State: 1
        Volume Up Digit: 2048
      Volume Down Digit: 1024
     Volume Reset Digit: 16
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_EnableEvents on cnfB1 SUCCESSFUL
Received following event information:
            Event: 49162
       Event Data: 0x873a288
      Event Count: 5
            Event: 301
            Event: 302
            Event: 305
            Event: 303
            Event: 304
Event Data Length: 32
     Event Device: 1
  Event User Info: 0x1

cnf_GetAttributes( ) on cnfB1 failed!! -
```

```
Expected error due to invalid attribute Error Code: 5
   Error String: Invalid attribute provided by user
Additional Info: Attribute[102] not a valid device attribute

cnf_GetAttributes( ) on cnfB1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[1]


Value[0x1]
   Attribute Info: Attribute[2]

Value[0x1]
   Attribute Info: Attribute[3]

Value[0x7d0]
Event Data Length: 48
  Event User Info: 0

cnf_SetAttributes( ) on cnfB1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[1]

Value[0x1]
   Attribute Info: Attribute[3]

Value[0x7d0]
Event Data Length: 36
  Event User Info: 0

cnf_GetAttributes( ) on cnfB1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[1]

Value[0x1]
   Attribute Info: Attribute[2]

Value[0x1]
   Attribute Info: Attribute[3]

Value[0x7d0]

Event Data Length: 48
  Event User Info: 0

Received CONFERENCE OPENED notification event...
Conference Handle: 2
  Conference Name: cnfB1C1
     Event Device: 1

cnf_OpenConference( ) on cnfB1 SUCCESSFUL
Received following event information:
Conference Device: 2
  Conference Name: cnfB1C1
     Event Device: 1
  Event User Info: 0

cnf_EnableEvents on cnfB1C1 SUCCESSFUL
Received following event information:
            Event: 49162
       Event Data: 0x873bc00
      Event Count: 3
            Event: 401
            Event: 402
```

```
           Event: 404
Event Data Length: 24
     Event Device: 2
  Event User Info: 0x1

cnf_GetAttributes( ) on cnfB1C1 failed!! -

Expected error due to invalid attribute
     Error Code: 5
   Error String: Invalid attribute provided by user
Additional Info: Attribute[3] not a valid device attribute

cnf_GetAttributes( ) on cnfB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[101]

Value[0x1]
   Attribute Info: Attribute[102]

Value[0x0]
   Attribute Info: Attribute[103]

Value[0x0]
Event Data Length: 48
  Event User Info: 0

cnf_SetAttributes( ) on cnfB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[101]


Value[0x1]
   Attribute Info: Attribute[102]

Value[0x40000f]
Event Data Length: 36
  Event User Info: 0

cnf_GetAttributes( ) on cnfB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[101]

Value[0x1]
   Attribute Info: Attribute[102]

Value[0x1007]
   Attribute Info: Attribute[103]

Value[0x0]
Event Data Length: 48
  Event User Info: 0
sr_waitevt failed - No error

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 3
     Party Name: cnfB1P1
   Event Device: 1
Event User Info: 0

cnf_GetAttributes( ) on cnfB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]
```

```
Value[0x0]
   Attribute Info: Attribute[202]

Value[0x0]
   Attribute Info: Attribute[203]

Value[0x0]
   Attribute Info: Attribute[204]

Value[0x0]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0x0]
   Attribute Info: Attribute[207]

Value[0x0]
Event Data Length: 96
  Event User Info: 0

cnf_SetAttributes( ) on cnfB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[201]

Value[0x1]
   Attribute Info: Attribute[202]

Value[0x1]
Event Data Length: 36
  Event User Info: 0

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: cnfB1C1
     Party Handle: 3
       Party Name: cnfB1P1
     Event Device: 1

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: cnfB1C1
     Party Handle: 3
       Party Name: cnfB1P1
     Event Device: 2

cnf_AddParty( ) on cnfB1C1 SUCCESSFUL
Received following event information:
      Party Count: 1
     Party Handle: 3
  Event User Info: 0xbff2e6f8

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 4
     Party Name: cnfB1P2
   Event Device: 1
Event User Info: 0

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 5
     Party Name: cnfB1P3
   Event Device: 1
```

```
Event User Info: 0

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 6
     Party Name: cnfB1P4
   Event Device: 1
Event User Info: 0

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 7
     Party Name: cnfB1P5
   Event Device: 1
Event User Info: 0

cnf_OpenParty( ) on cnfB1 SUCCESSFUL
Received following event information:
   Party Device: 8
     Party Name: cnfB1P6
   Event Device: 1
Event User Info: 0

cnf_AddParty( ) - failed
     Error Code: 15
   Error String: Functionality currently not supported
Additional Info: Cannot add more than 1 party

cnf_GetPartyList( ) - Successful
Received following event information:
            Event: 49167
       Event Data: 0x8761bb8
      Party Count: 1
    Party Info: Party[0] - Handle[3]
- Device Name[cnfB1P1]
Event Data Length: 16
     Event Device: 2
  Event User Info: 0

cnf_GetAttributes( ) on cnfB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]

Value[0x1]
   Attribute Info: Attribute[202]

Value[0x1]
   Attribute Info: Attribute[203]

Value[0x0]
   Attribute Info: Attribute[204]

Value[0x0]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0x0]
   Attribute Info: Attribute[207]

Value[0x1]
Event Data Length: 96
  Event User Info: 0

cnf_SetAttributes( ) on cnfB1P1 SUCCESSFUL
```

```
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[201]

Value[0x0]
   Attribute Info: Attribute[202]

Value[0x0]
Event Data Length: 36
  Event User Info: 0

cnf_GetAttributes( ) on cnfB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]

Value[0x0]
   Attribute Info: Attribute[202]

Value[0x0]
   Attribute Info: Attribute[203]

Value[0x0]
   Attribute Info: Attribute[204]

Value[0x0]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0x0]
   Attribute Info: Attribute[207]

Value[0x1]
Event Data Length: 96
  Event User Info: 0

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: cnfB1C1
     Party Handle: 3
       Party Name: cnfB1P1
     Event Device: 1

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: cnfB1C1
     Party Handle: 3
       Party Name: cnfB1P1
     Event Device: 2

cnf_RemoveParty( ) on cnfB1C1 SUCCESSFUL
Received following event information:
      Party Count: 1
     Party Handle: 3
  Event User Info: 0

cnf_CloseParty( ) - successful

cnf_CloseParty( ) - successful

cnf_CloseParty( ) - successful

cnf_CloseParty( ) - successful

cnf_CloseParty( ) - successful
```

```
cnf_DisableEvents on cnfB1C1 SUCCESSFUL
Received following event information:
      Event Count: 3
            Event: 401
            Event: 402
            Event: 404
  Event User Info: 0x1

Received CONFERENCE CLOSED notification event...

  Conference Name: cnfB1C1
     Event Device: 1

cnf_CloseConference( ) - successful
cnf_CloseParty( ) - successful !!

cnf_DisableEvents on cnfB1 SUCCESSFUL
Received following event information:
      Event Count: 5
            Event: 301
            Event: 302
            Event: 305
            Event: 303
            Event: 304
  Event User Info: 0x1

cnf_Close( ) - Successful
```

Figure 3. Conferencing (MCX) Example Code Output

```
Conferencing (MCX) Example Code
=======================================

Board Name is: mcxB1

cnf_Open failure!! : Expected failure due to the following
     Error Code: 4
   Error String: Invalid parameter


in function call
Additional Info: Invalid parameter - a_szBrdName is NULL

cnf_Open failure!! : Expected failure due to the following
     Error Code: 3
   Error String: Invalid device name provided by user
Additional Info: Invalid device name [blah_blah] specified

cnf_Open( ) - Successful
Received following event information:
       Event Data: 0
Event Data Length: 10
     Event Device: 1
  Event User Info: 0

cnf_GetDeviceCount( ) on mcxB1 SUCCESSFUL
Received following event information:
             Event Data: 0x96a1630
     Free Party Devices: 60
Free Conference Devices: 30
      Max Party Devices: 60
 Max Conference Devices: 30
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0
```

```
cnf_ResetDevices( ) on mcxB1 SUCCESSFUL
Received following event information:
       Event Data: 0
Event Data Length: 10
  Event User Info: 0

cnf_GetDeviceCount( ) on mcxB1 SUCCESSFUL
Received following event information:
             Event Data: 0x96a1630
     Free Party Devices: 61
Free Conference Devices: 30
      Max Party Devices: 60
 Max Conference Devices: 30
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_GetDTMFControl( ) on mcxB1 SUCCESSFUL
Received following event information:
                  Event: 49164
             Event Data: 0x969f7e8
     DTMF Control State: 1
        Volume Up Digit: 2048
      Volume Down Digit: 1024
     Volume Reset Digit: 16
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_SetDTMFControl( ) on mcxB1 SUCCESSFUL
Received following event information:
             Event Data: 0
      Event Data Length: 10
        Event User Info: 0

cnf_GetDTMFControl( ) on mcxB1 SUCCESSFUL
Received following event information:
                  Event: 49164
             Event Data: 0x969f7e8
     DTMF Control State: 1
        Volume Up Digit: 2048
      Volume Down Digit: 1024
     Volume Reset Digit: 16
      Event Data Length: 20
           Event Device: 1
        Event User Info: 0

cnf_EnableEvents on mcxB1 SUCCESSFUL
Received following event information:
           Event: 49162
       Event Data: 0x96b8a90
      Event Count: 5
            Event: 301
            Event: 302
            Event: 305
            Event: 303
            Event: 304
Event Data Length: 32
     Event Device: 1
  Event User Info: 0x1

cnf_GetAttributes( ) on mcxB1 failed!! -

Expected error due to invalid attribute
     Error Code: 5
   Error String: Invalid attribute provided by user
```

```
Additional Info: Attribute[102] not a valid device attribute

cnf_GetAttributes( ) on mcxB1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[1]

Value[0x1]
   Attribute Info: Attribute[2]

Value[0x0]
   Attribute Info: Attribute[3]

Value[0x1d01]
Event Data Length: 48
  Event User Info: 0

cnf_SetAttributes( ) on mcxB1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[1]

Value[0x1]
   Attribute Info: Attribute[3]

Value[0x7d0]
Event Data Length: 36
  Event User Info: 0

cnf_GetAttributes( ) on mcxB1 SUCCESSFUL

Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[1]

Value[0x1]
   Attribute Info: Attribute[2]

Value[0x0]
   Attribute Info: Attribute[3]

Value[0x1d01]
Event Data Length: 48
  Event User Info: 0

cnf_OpenConference( ) on mcxB1 SUCCESSFUL
Received following event information:
Conference Device: 2
  Conference Name: mcxB1C1
     Event Device: 1
  Event User Info: 0

Received CONFERENCE OPENED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Event Device: 1

cnf_GetVideoLayout( ) on mcxB1C1 SUCCESSFUL
Received following event information:
          Layout Handle: 256
            Layout Size: 1
            Layout Type: 0
        Event User Info: 0

Created 4 region layout...
cnf_SetVideoLayout( ) on mcxB1C1 SUCCESSFUL
Received following event information:
```

```
          Layout Handle: 256
            Layout Size: 2
            Layout Type: 401
        Event User Info: 0

cnf_EnableEvents on mcxB1C1 SUCCESSFUL
Received following event information:
            Event: 49162
       Event Data: 0x96b9830
      Event Count: 3
            Event: 401
            Event: 402
            Event: 404
Event Data Length: 24
     Event Device: 2
  Event User Info: 0x1

cnf_GetAttributes( ) on mcxB1C1 failed!! -
Expected error due to invalid attribute
     Error Code: 5
   Error String: Invalid attribute provided by user
Additional Info: Attribute[3] not a valid device attribute

cnf_GetAttributes( ) on mcxB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[101]


Value[0x1]
   Attribute Info: Attribute[102]


Value[0x3]
   Attribute Info: Attribute[103]

Value[0x3e8]
Event Data Length: 48
  Event User Info: 0

cnf_SetAttributes( ) on mcxB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[101]

Value[0x1]
   Attribute Info: Attribute[102]

Value[0x40000f]
Event Data Length: 36
  Event User Info: 0

cnf_GetAttributes( ) on mcxB1C1 SUCCESSFUL
Received following event information:
  Attribute Count: 3
   Attribute Info: Attribute[101]

Value[0x1]
   Attribute Info: Attribute[102]

Value[0x3]
   Attribute Info: Attribute[103]

Value[0x3e8]
Event Data Length: 48
  Event User Info: 0
```

```
cnf_GetPartyList( ) - Successful
Received following event information:
            Event: 49167
       Event Data: 0x96b9610
      Party Count: 0
Event Data Length: 12
     Event Device: 2
  Event User Info: 0

cnf_OpenParty( ) on mcxB1P1 SUCCESSFUL
Received following event information:
   Party Device: 3
     Party Name: mcxB1P1
   Event Device: 3
Event User Info: 0

cnf_GetAttributes( ) on mcxB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]

Value[0x0]
   Attribute Info: Attribute[202]

Value[0x7fffac]
   Attribute Info: Attribute[203]

Value[0x420]
   Attribute Info: Attribute[204]

Value[0x1]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0xccccc]
   Attribute Info: Attribute[207]

Value[0x420]
Event Data Length: 96
  Event User Info: 0

cnf_SetAttributes( ) on mcxB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[201]

Value[0x1]
   Attribute Info: Attribute[202]

Value[0x1]
Event Data Length: 36
  Event User Info: 0

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 3
       Party Name: mcxB1P1
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 3
       Party Name: mcxB1P1
```

```
     Event Device: 1

cnf_AddParty( ) on mcxB1C1 SUCCESSFUL
Received following event information:
      Party Count: 1
     Party Handle: 3
  Event User Info: 0xbff60468

cnf_OpenParty( ) on mcxB1P2 SUCCESSFUL
Received following event information:
   Party Device: 5
     Party Name: mcxB1P2
   Event Device: 5
Event User Info: 0

cnf_OpenParty( ) on mcxB1P3 SUCCESSFUL
Received following event information:
   Party Device: 7
     Party Name: mcxB1P3
   Event Device: 7
Event User Info: 0

cnf_OpenParty( ) on mcxB1P4 SUCCESSFUL
Received following event information:
   Party Device: 9
     Party Name: mcxB1P4
   Event Device: 9
Event User Info: 0

cnf_OpenParty( ) on mcxB1P5 SUCCESSFUL
Received following event information:
   Party Device: 11
     Party Name: mcxB1P5
   Event Device: 11
Event User Info: 0

cnf_OpenParty( ) on mcxB1P6 SUCCESSFUL
Received following event information:
   Party Device: 13
     Party Name: mcxB1P6
   Event Device: 13
Event User Info: 0

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 5
       Party Name: mcxB1P2
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 7
       Party Name: mcxB1P3
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 9
       Party Name: mcxB1P4
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
```

```
     Party Handle: 11
       Party Name: mcxB1P5
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 13
       Party Name: mcxB1P6
     Event Device: 2

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 5
       Party Name: mcxB1P2
     Event Device: 1

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 7
       Party Name: mcxB1P3
     Event Device: 1

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 9
       Party Name: mcxB1P4
     Event Device: 1

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 11
       Party Name: mcxB1P5
     Event Device: 1

Received PARTY ADDED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 13
       Party Name: mcxB1P6
     Event Device: 1

cnf_AddParty( ) on mcxB1C1 SUCCESSFUL
Received following event information:
      Party Count: 5
     Party Handle: 5
     Party Handle: 7
     Party Handle: 9
     Party Handle: 11
     Party Handle: 13
  Event User Info: 0

cnf_GetPartyList( ) - Successful
Received following event information:
            Event: 49167
       Event Data: 0x96e1f68
      Party Count: 6
    Party Info: Party[0] - Handle[3]
- Device Name[mcxB1P1]
    Party Info: Party[1] - Handle[5]
- Device Name[mcxB1P2]
    Party Info: Party[2] - Handle[7]
```

```
- Device Name[mcxB1P3]
    Party Info: Party[3] - Handle[9]

- Device Name[mcxB1P4]
    Party Info: Party[4] -
Handle[11] - Device Name[mcxB1P5]
    Party Info: Party[5] -
Handle[13] - Device Name[mcxB1P6]
Event Data Length: 36
     Event Device: 2
  Event User Info: 0

cnf_SetVisiblePartyList( ) on mcxB1C1 SUCCESSFUL
Received following event information:
VisiblePartyList[0] --- Party Handle: 5 Region Handle: 257
VisiblePartyList[1] --- Party Handle: 7 Region Handle: 258
VisiblePartyList[2] --- Party Handle: 9 Region Handle: 259
VisiblePartyList[3] --- Party Handle: 11 Region Handle: 260
        Event User Info: 0

cnf_GetVisiblePartyList( ) on mcxB1C1 SUCCESSFUL
Received following event information:
VisiblePartyList[0] --- Party Handle: 5 Region Handle: 257
VisiblePartyList[1] --- Party Handle: 7 Region Handle: 258
VisiblePartyList[2] --- Party Handle: 9 Region Handle: 259
VisiblePartyList[3] --- Party Handle: 11 Region Handle: 260
        Event User Info: 0

cnf_GetAttributes( ) on mcxB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]

Value[0x1]
   Attribute Info: Attribute[202]

Value[0x7fffac]
   Attribute Info: Attribute[203]

Value[0x420]
   Attribute Info: Attribute[204]

Value[0x1]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0xccccc]
   Attribute Info: Attribute[207]

Value[0x420]
Event Data Length: 96
  Event User Info: 0

cnf_SetAttributes( ) on mcxB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 2
   Attribute Info: Attribute[201]

Value[0x0]
   Attribute Info: Attribute[202]

Value[0x0]
Event Data Length: 36
  Event User Info: 0
```

```
cnf_GetAttributes( ) on mcxB1P1 SUCCESSFUL
Received following event information:
  Attribute Count: 7
   Attribute Info: Attribute[201]

Value[0x0]
   Attribute Info: Attribute[202]

Value[0x7fffac]
   Attribute Info: Attribute[203]

Value[0x420]
   Attribute Info: Attribute[204]

Value[0x1]
   Attribute Info: Attribute[205]

Value[0x0]
   Attribute Info: Attribute[206]

Value[0xccccc]
   Attribute Info: Attribute[207]

Value[0x420]
Event Data Length: 96
  Event User Info: 0

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 3
       Party Name: mcxB1P1
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 3
       Party Name: mcxB1P1
     Event Device: 1

cnf_RemoveParty( ) on mcxB1C1 SUCCESSFUL
Received following event information:
      Party Count: 1
     Party Handle: 3
  Event User Info: 0

cnf_CloseParty( ) - successful
Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 5
       Party Name: mcxB1P2
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 5
       Party Name: mcxB1P2
     Event Device: 1

cnf_CloseParty( ) - successful
Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 7
```

```
       Party Name: mcxB1P3
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 7
       Party Name: mcxB1P3
     Event Device: 1

cnf_CloseParty( ) - successful
Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 9
       Party Name: mcxB1P4
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 9
       Party Name: mcxB1P4
     Event Device: 1

cnf_CloseParty( ) - successful
Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 11
       Party Name: mcxB1P5
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 11
       Party Name: mcxB1P5
     Event Device: 1

cnf_CloseParty( ) - successful
Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 13
       Party Name: mcxB1P6
     Event Device: 2

Received PARTY REMOVED notification event...
Conference Handle: 2
  Conference Name: mcxB1C1
     Party Handle: 13
       Party Name: mcxB1P6
     Event Device: 1

cnf_DisableEvents on mcxB1C1 SUCCESSFUL
Received following event information:
      Event Count: 3
            Event: 401
            Event: 402
            Event: 404
   Event User Info: 0x1

Received CONFERENCE CLOSED notification event...
  Conference Name: mcxB1C1
     Event Device: 1
```

```
cnf_CloseConference( ) - successful

cnf_CloseParty( ) - successful !!

cnf_DisableEvents on mcxB1 SUCCESSFUL
Received following event information:
      Event Count: 5
            Event: 301
            Event: 302
            Event: 305
            Event: 303
            Event: 304
  Event User Info: 0x1

cnf_Close( ) - Successful
```

Glossary 13

active talker: A participant in a conference who is providing “non-silence” energy.

automatic gain control (AGC): An electronic circuit used to maintain the audio signal volume at a constant level. AGC maintains nearly constant gain during voice signals, thereby avoiding distortion, and optimizes the perceptual quality of voice signals by using a new method to process silence intervals (background noise).

asynchronous function: A function that allows program execution to continue without waiting for a task to complete. To implement an asynchronous function, an application-defined event handler must be enabled to trap and process the completed event. Contrast with synchronous function.

bit mask: A pattern which selects or ignores specific bits in a bit-mapped control or status field.

bitmap: An entity of data (byte or word) in which individual bits contain independent control or status information.

board device: A board-level object that maps to a virtual board.

buffer: A block of memory or temporary storage device that holds data until it can be processed. It is used to compensate for the difference in the rate of the flow of information (or time occurrence of events) when transmitting data from one device to another.

bus: An electronic path that allows communication between multiple points or devices in a system.

busy device: A device that has one of the following characteristics: is stopped, being configured, has a multitasking or non-multitasking function active on it, or I/O function active on it.

channel device: A channel-level object that can be manipulated by a physical library, such as an individual telephone line connection. A channel is also a subdevice of a board.

CO (Central Office): A local phone network exchange, the telephone company facility where subscriber lines are linked, through switches, to other subscriber lines (including local and long distance lines). The term “Central Office” is used in North America. The rest of the world calls it “PTT”, for Post, Telephone, and Telegraph.

coach: A participant in a conference that can be heard by pupils only. A mentoring relationship exists between a coach and a pupil.

conferee: Participant in a conference call. Synonym of party.

conference: Ability for three or more participants in a call to communicate with one another in the same call.

conferencing: Ability to perform a conference.

conference bridging: Ability for all participants in two or more established conferences to speak to and/or listen to one another.

configuration file: An unformatted ASCII file that stores device initialization information for an application.

configuration manager: A utility with a graphical user interface (GUI) that enables you to add new boards to your system, start and stop system service, and work with board configuration data. Also known as DCM.

CT Bus: Computer Telephony bus. A time division multiplexing communications bus that provides 4096 time slots for transmission of digital information between CT Bus products. See TDM bus.

data structure: Programming term for a data element consisting of fields, where each field may have a different type definition and length. A group of data structure elements usually share a common purpose or functionality.

device: A computer peripheral or component controlled through a software device driver. A Dialog® voice and/or network interface expansion board is considered a physical board containing one or more logical board devices, and each channel or time slot on the board is a device.

device channel: A voice data path that processes one incoming or outgoing call at a time (equivalent to the terminal equipment terminating a phone line).

device driver: Software that acts as an interface between an application and hardware devices.

device handle: Numerical reference to a device, obtained when a device is opened using **xx_open( )**, where *xx* is the prefix defining the device to be opened. The device handle is used for all operations on that device.

device name: Literal reference to a device, used to gain access to the device via an **xx_open( )** function, where *xx* is the prefix defining the device to be opened.

DM3: Refers to Dialogic® mediastream processing architecture, which is open, layered, and flexible, encompassing hardware as well as software components. A whole set of products from Dialogic are built on DM3 architecture.

driver: A software module which provides a defined interface between a program and the firmware interface.

DTMF (Dual-Tone Multifrequency): Push-button or touch-tone dialing based on transmitting a high- and a low-frequency tone to identify each digit on a telephone keypad.

E1: A CEPT digital telephony format devised by the CCITT, used in Europe and other countries around the world. A digital transmission channel that carries data at the rate of 2.048 Mbps (DS-1 level). CEPT stands for the Conference of European Postal and Telecommunication Administrations. Contrast with T1.

extended attribute functions: A class of functions that take one input parameter and return device-specific information. For instance, a voice device's extended attribute function returns information specific to the voice devices. Extended attribute function names are case-sensitive and must be in capital letters. See also standard runtime library (SRL).

firmware: A set of program instructions that reside on an expansion board.

idle device: A device that has no functions active on it.

party: A participant in a conference. Synonym of conferee.

pupil: A participant in a conference that has a mentoring relationship with a coach.

resource: Functionality (for example, conferencing) that can be assigned to a call. Resources are *shared* when functionality is selectively assigned to a call and may be shared among multiple calls. Resources are *dedicated* when functionality is fixed to the one call.

RFU: Reserved for future use.

route: Assign a resource to a time slot.

SRL: See **Standard Runtime Library**.

standard attribute functions: Class of functions that take one input parameter (a valid device handle) and return generic information about the device. For instance, standard attribute functions return IRQ and error information for all device types. Standard attribute function names are case-sensitive and must be in capital letters. Standard attribute functions for all Dialogic® devices are contained in the SRL. See standard runtime library (SRL).

standard runtime library (SRL): A Dialogic® software resource containing event management and standard attribute functions and data structures used by all Dialogic® devices, but which return data unique to the device.

synchronous function: Blocks program execution until a value is returned by the device. Also called a blocking function. Contrast with asynchronous function.

T1: A digital line transmitting at 1.544 Mbps over 2 pairs of twisted wires. Designed to handle a minimum of 24 voice conversations or channels, each conversation digitized at 64 Kbps. T1 is a digital transmission standard in North America. Contrast with E1.

TDM (Time Division Multiplexing): A technique for transmitting multiple voice, data, or video signals simultaneously over the same transmission medium. TDM is a digital technique that interleaves groups of bits from each signal, one after another. Each group is assigned its own "time slot" and can be identified and extracted at the receiving end. See also time slot.

TDM bus: Time division multiplexing bus. A resource sharing bus such as the SCbus or CT Bus that allows information to be transmitted and received among resources over multiple data lines.

termination condition: An event or condition which, when present, causes a process to stop.

termination event: An event that is generated when an asynchronous function terminates. See also asynchronous function.

thread (Windows®): The executable instructions stored in the address space of a process that the operating system actually executes. All processes have at least one thread, but no thread belongs to more than one process. A multithreaded process has more than one thread that are executed seemingly simultaneously. When the last thread finishes its task, then the process terminates. The main thread is also referred to as a primary thread; both main and primary thread refer to the first thread started in a process. A thread of execution is just a synonym for thread.

tone clamping: (DTMF tone clamping) Mutes DTMF tones heard in a conference. If a confereee's phone generates a tone, the DTMF signal will not interfere with the conference. Applies to transmitted audio into the conference and does not affect DTMF function.

time division multiplexing (TDM): See TDM (Time Division Multiplexing).

time slot: The smallest, switchable data unit on a TDM bus. A time slot consists of 8 consecutive bits of data. One time slot is equivalent to a data path with a bandwidth of 64 kbps. In a digital telephony environment, a normally continuous and individual communication (for example, someone speaking on a telephone) is (1) digitized, (2) broken up into pieces consisting of a fixed number of bits, (3) combined with pieces of other individual communications in a regularly repeating, timed sequence (multiplexed), and (4) transmitted serially over a single telephone line. The process happens at such a fast rate that, once the pieces are sorted out and put back together again at the receiving end, the speech is normal and continuous. Each individual, pieced-together communication is called a time slot.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W