Call Logging API Software Reference

for Windows

05-1591-002

Publication Date: November, 2003

Intel Converged Communications, Inc.
1515 Route 10
Parsippany NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website:
http://developer.intel.com/design/telecom/support/

For **Products and Services Information**, visit the Intel Communications Systems Products website:
http://www.intel.com/design/network/products/telecom/

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page:
http://www.intel.com/buy/wtb/wtb1028.htm

Table of Contents

List of Tables

List of Figures

Revision History

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

Document No.	Publication Date	Description of Revisions
05-1591-002	November 2003	**Chapter 1**: updated all sections and added sections to reflect newly supported analog HiZ boards. Revised Figure 1 and Figure 2. **Chapter 2**: added new chapter and new content on Call Logging HiZDemo and SnifferMFC demos. **cl_GetOrdinalChannel( )**: new function. **cl_Open( )**: updated to reflect newly supported analog HiZ boards. **cl_PeekOrdinalChannel( )**: new function. **cl_StartTrace( )**: previously documented in System Release 5.1 for Windows Release Update. **cl_StopTrace( )**: previously documented in System Release 5.1 for Windows Release Update.
05-1591-001	August 2001	Initial version of document.

Preface

Intended Audience

This guide is for application developers who wish to use the Intel® Call Logging application programming interface (API) to build call monitoring or call recording applications.

Organization of this Guide

This guide is organized as follows:

- **Chapter 1** provides an overview of the functionality of the Call Logging API, including a typical hardware configuration, interfaces with other libraries, system operation, and call logging events.
- **Chapter 2** provides information on Call Logging demos.
- **Chapter 3** provides an overview of the Call Logging API functions.
- **Chapter 4** provides detailed descriptions of the Call Logging API functions.
- **Appendix A** provides sample code for developing a call logging application.

A **Glossary** and an **Index** are also included.

How to Use This Guide

This guide provides detailed information about Call Logging API functions, parameters, and events. Other APIs, such as the R4 Voice library, are used to develop call logging applications. Please refer to the appropriate API documentation for information about other API functions.

Documentation Conventions

The following conventions are used in this document:

- Function Names - are shown in bold with the name of the function followed by parentheses, for example, **cl_Close( )**.
- Function Parameters - are shown in bold, for example, **linedev**.
- Events - are shown in uppercase, for example, CLEV_MESSAGE.
- Data Structures - are shown in uppercase, for example, CL_EVENTDATA.
- Error Codes - are shown in uppercase, for example, ECL_OUT_OF_MEMORY.
- Result Values - are shown in uppercase, for example, ECL_CONNECT_MESSAGE.

Related Documentation

Refer to the following documents in addition to the *Call Logging API Software Reference* when developing a Call Logging application:

- *Global Call API Library Reference*
- *Global Call API Programming Guide*
- the Global Call Technology User's Guide for the protocol you are using
- *Voice API Library Reference*
- *Voice API Programming Guide*
- *Standard Runtime Library API Library Reference*
- *Standard Runtime Library API Programming Guide*

1. Product Description

The Call Logging API is the software companion of the HiZ family of products, which provide high impedance interfaces for non-intrusive line monitoring. This chapter describes the Call Logging API under the following topics:

- 1.1. Call Logging API Overview
- 1.2. Digital High Impedance (HiZ) Hardware Configuration
- 1.3. Analog High Impedance (HiZ) Hardware Configuration
- 1.4. Supported Configurations
- 1.5. Supported Protocols
- 1.6. External Interfaces
- 1.7. Call Logging System Operation
- 1.8. Call Logging Scenarios
- 1.9. Device Enumeration

1.1. Call Logging API Overview

The Call Logging API enables the development of applications to monitor the traffic on analog or digital lines between the network side and the user side. The network side refers to the central office (CO) public switched telephone network (PSTN), and the user side refers to the customer premises equipment (CPE) private branch exchange (PBX).

In addition, the Call Logging API together with the Voice library can be used to build call recording applications to record conversations on analog or digital lines that connect the user side and the network side.

The Call Logging API handles transactions and reports transaction events for an analog or digital line so that call monitoring or call recording applications can be developed seamlessly for either line type.

On digital HiZ products, the Call Logging API can be used to create call monitoring applications that monitor Layer 1 (L1) alarms, Layer 2 (L2) events, and Layer 3 (L3) messages. More specifically, the Call Logging API offers the following features for the development of call monitoring applications:

- handling L1 alarms, such as "Loss of Sync", L2 events, including lost frames, and L3 messages, such as state changes, and reporting the alarms, events, as well as messages to the application
- gathering L3 messages and handling transaction state transitions, such as dialing, connected, or disconnected, according to the L3 messages received
- querying previously received L3 messages when a transaction event is triggered
- retrieving protocol-specific information from L3 messages

On analog HiZ products, all pertinent signaling and supervision events are available to call logging applications.

1.2. Digital High Impedance (HiZ) Hardware Configuration

As shown in Figure 1, a digital line connecting the CPE PBX (user side) to a CO PSTN (network side) can be monitored using two HiZ connectors on a HiZ board. One of these HiZ connectors receives the voice and signaling data transmitted by the CO PSTN, while the other receives the data transmitted by the CPE PBX. This typical HiZ configuration constitutes the hardware part of a call logging system. The Call Logging API simplifies the implementation of the software in such a system.

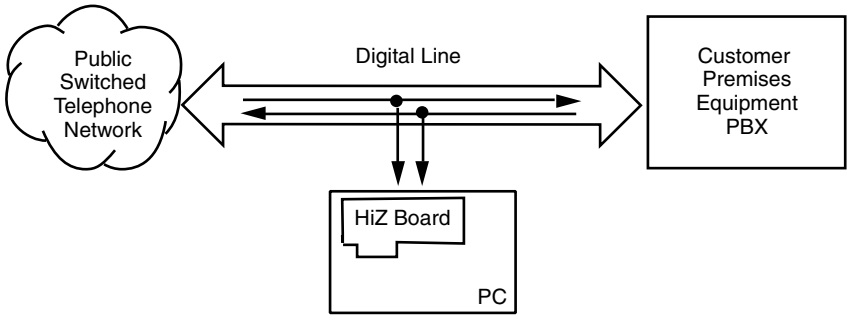


Figure 1. Typical Digital High Impedance (HiZ) Configuration

For more detailed information about connecting HiZ cable assemblies, see the *Quick Install Card* that is included with the board.

1.3. Analog High Impedance (HiZ) Hardware Configuration

Unlike the digital HiZ hardware configuration, a call logging system for analog HiZ boards uses one connector for each channel being monitored. The **cl_Open( )** function has been extended to support the specification of several analog HiZ devices in one function call. The lines connected to these HiZ devices can subsequently be monitored using a single call logging device handle.

As shown in *Figure 2*, an analog CO PSTN line connecting to a customer premises PBX/KTS or analog phone can be monitored using a single connector on an analog HiZ board. The analog HiZ board receives the ringing, loop signaling, caller ID tones, and voice passing between the network and the phone user. With an analog HiZ board such as the DMV160LPHIZ, up to 16 analog lines can be monitored.

In an environment without a PBX/KTS, the HiZ board taps into the analog line between the CO PSTN and the phone user.

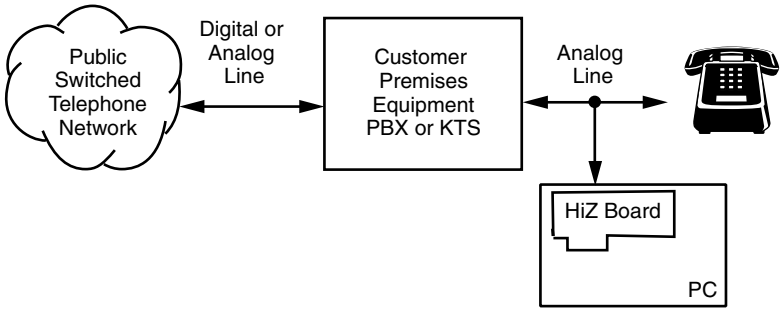


Figure 2. Typical Analog High Impedance (HiZ) Configuration

1.4. Supported Configurations

The Call Logging API supports high impedance boards, including the HiZ family of products. Refer to the *Release Guide* accompanying the software release you installed for a list of supported board models.

1.5. Supported Protocols

The Call Logging API provides a protocol-independent API and can be used with analog HiZ boards, or with digital HiZ boards on lines using a common channel signaling (CCS) protocol. Refer to the *Release Guide* accompanying the software release for a list of supported protocols.

1.6. External Interfaces

Figure 3 illustrates how the Call Logging API interacts with other R4 libraries and the application.

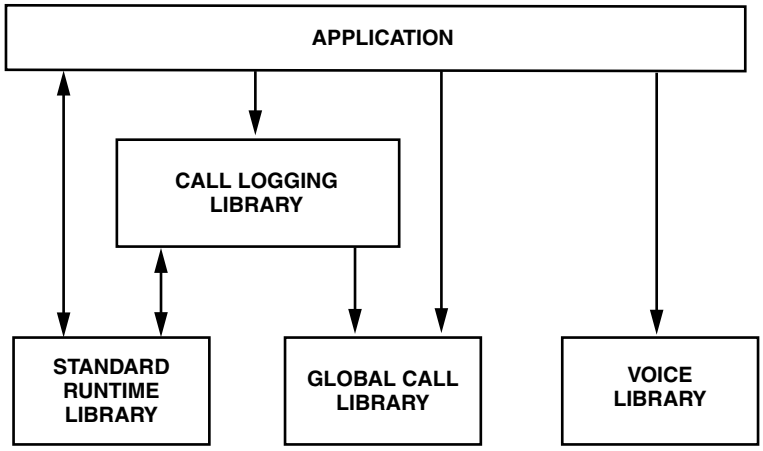


Figure 3. Call Logging API Interfaces

A description of how the other libraries interact with the Call Logging API follows.

1.6.1. Standard Runtime Library

The Call Logging API registers the call logging device with the Standard Runtime Library (SRL). Call logging events are posted to the SRL, which then delivers these events to the application. You must either call the **sr_enbhdlr( )** function to install an event handler or call the **sr_waitevt( )** function to retrieve and process the call logging events posted by the Call Logging API. See *section 1.8. Call Logging Scenarios* for more information on using SRL library functions in your application. For more information about SRL functions, see the *Standard Runtime Library API Library Reference*.

1.6.2. Global Call Library

The Call Logging API uses the Global Call **gc_GetFrame( )** function or **gc_Extension( )** function to collect signaling data.

In all applications on analog HiZ boards, you must either call the **gc_OpenEx( )** function (with network interface and voice resource) or the **gc_AttachResource( )**

function to associate the network interface devices with voice resource devices. For more information about Global Call API functions, see the *Global Call API Library Reference*.

In call monitoring applications on digital HiZ boards, there is no need to use **gc_OpenEx( )** or **gc_AttachResource( )** because all of the relevant information is provided by the Call Logging API.

However, in call recording applications on digital HiZ boards, you need to know the time slot on which the voice data is being transmitted. Therefore, you must open the devices by calling either the **gc_OpenEx( )** function (with network interface and voice resource) or the **gc_AttachResource( )** function to associate the network interface devices with voice resource devices.

See *section 1.8. Call Logging Scenarios* for more information on using Global Call library functions in your application.

1.6.3. Voice Library

When the Call Logging API is used for call recording applications, the Voice **dx_mreciottdata( )** function (for digital HiZ) or **dx_reciottdata( )** function (for analog HiZ) is called to perform the transaction recording.

The RM_NOTIFY flag in the mode parameter of various Voice record functions is used to instruct these functions to generate a record notification beep tone.

The **dx_SetRecordNotifyBeepTone( )** function specifies the template of the cadenced tone to be used as the record notification beep tone during subsequent calls to the Voice record functions. This function overwrites the default template used on DM3 boards. If no template is specified, the default beep tone has these specifications: 1400 Hz, -18 dB, 420 msecs on, 15 secs off.

See *section 1.8. Call Logging Scenarios* for more information on using Voice library functions in your application. For details on Voice API functions, see the *Voice API Library Reference*.

1.7. Call Logging System Operation

The primary function of a call logging system is to observe the activity on analog or digital lines. When a call is established on a monitored line, the call logging system receives the voice and signaling data from both the outbound and inbound parties. In order to indicate this dual source of data, the Call Logging API refers to calls as call logging **transactions**.

The Call Logging API analyzes the signaling data and manages the call logging transactions according to **semantics rules**, which consist of:

- a list of semantics **states**, such as dialing (or ringing), alerting (or seizing the line), connected or disconnected, that a call logging transaction can use to represent the current status of the monitored call. The number, index, and names of the semantics states are retrieved using the **cl_GetSemanticsState** functions.
- a list of semantics **variables**, such as calling party number, called party number, interface identifier, digital bearer channel (B channel) or analog channel number, and ordinal channel number. The semantics variables are assigned from the contents of the signaling data and can be queried by the application. The variables can be queried on an event basis when an event is received or on a polling basis:
    - Event basis – use the "Get" functions **cl_GetCalled( )**, **cl_GetCalling( )**, **cl_GetChannel( )**, **cl_GetOrdinalChannel( )**, and **cl_GetVariable( )**.
    - Polling basis – use the "Peek" functions **cl_PeekCalled( )**, **cl_PeekCalling( )**, **cl_PeekChannel( )**, **cl_PeekOrdinalChannel( )**, and **cl_PeekVariable( )**.
- a list of the specific **events** that the Call Logging API must monitor to identify signaling data (analog HiZ) or the first and the last L3 messages (digital HiZ) related to a call logging transaction. The list of events is also used to determine when the monitored call is connected and later on disconnected. This list allows the Call Logging API to report those key events to the application. Information about specific events is retrieved using the **cl_GetTransaction( )** and **cl_GetTransactionDetails( )** functions along with other call logging functions depending on the information required. (See *section 1.8.3. Event Handling* for a sample scenario.)

For more information on Call Logging API functions, see *Chapter 4. Call Logging Function Reference*.

1.7.1. Generating Call Logging Events

Because of the high impedance nature of its configuration, the call logging system is only an observer; it never has to make outbound calls or answer inbound calls. Instead, the call logging system receives unsolicited events, such as analog line events, digital L1 alarms, and digital L2 frames that may contain L3 messages.

When an analog line event is received, the Call Logging API:

1. identifies the line event and the channel on which it occurred.
2. converts the line event into a call logging message.
3. changes the state of the related call logging transaction according to the semantics rules.
4. sends an unsolicited call logging event to the application.

When a digital L2 frame containing a digital L3 message is received, the Call Logging API:

1. extracts the L3 message from the L2 frame.
2. decodes the L3 message.
3. changes the state of the related call logging transaction according to the semantics rules.
4. sends an unsolicited call logging event to the application.

Table 1. Call Logging Events describes the call logging events that the Call Logging API can generate:

Table 1. Call Logging Events

Event	Description
CLEV_MESSAGE	An analog line event is received, or the monitored L2 frame contains an L3 message about a call logging transaction.

Event	Description
CLEV_ALARM	An analog line event was received, or an L1 alarm was detected.
CLEV_ERROR	An error occurred.

1.7.2. Retrieving Event Data

Information about events received by the application is contained in an event data block. This information includes the time the initial unsolicited event was observed by the Call Logging API. You obtain the event data block by calling the **sr_getevtdatap( )** function while processing a call logging event.

You obtain the type of event by calling the **sr_getevttype( )** function. As described in Table 1, the type of event indicates whether a message was received (CLEV_MESSAGE), an alarm was detected (CLEV_ALARM), or an error occurred (CLEV_ERROR).

CL_EVENTDATA Data Structure

The event data block associated with call logging events is based on the CL_EVENTDATA data structure:

```
{
      int         iResult;
      time_t      timeEvent;
} CL_EVENTDATA;
```

Table 2. CL_EVENTDATA Data Structure Fields describes the fields in the CL_EVENTDATA data structure.

Table 2. CL_EVENTDATA Data Structure Fields

Field	Meaning/Values
iResult	A bitset of result codes and error codes. The field can contain several of the following symbolic values: **CLEV_MESSAGE events:** • ECL_CONNECT_MESSAGE

Field	Meaning/Values
	• ECL_DISCONNECT_MESSAGE • ECL_FIRST_MESSAGE • ECL_LAST_MESSAGE • ECL_NOERR – The value of ECL_NOERR is 0. This value means that no error was detected and because no bit is set, the message received is not the first or last message, not a connect or disconnect message, and that this message has not triggered a semantics state change. You can ignore it unless you want to log every message. Note that digital L3 messages and analog line events are reported to the application as call logging messages by means of CLEV_MESSAGE. • ECL_STATE_HAS_CHANGED • ECL_WRONG_FIRST_MESSAGE – This bit is set (together with the ECL_FIRST_MESSAGE bit) to indicate that the received message is the first message received about the call logging transaction, but this message is unexpected according to the semantics rules. This situation happens when there are already calls in progress when the call logging application starts and the received message is about one of these other calls. This situation can also occur if the expected first message was missed or incorrectly received by the call logging system because of bad cables, poor connections, or glitches on the line. **CLEV_ERROR events:** • ECL_L2FRAMES_WERE_LOST • ECL_L2LAYER_WAS_RESTARTED

Field	Meaning/Values
	• ECL_OUT_OF_MEMORY – This bit means there is no more memory left to store transactions. You can get this error event if the application does not use **cl_ReleaseTransaction( )**. • ECL_UNRECOGNIZED_L2FRAME • ECL_UNRECOGNIZED_L3MESSAGE
timeEvent	The time when the analog line event was received, the digital L2 frame was monitored on the line (that is, when the message was received or the error occurred), or the digital L1 alarm event was detected.

1.8. Call Logging Scenarios

This section provides scenarios for typical call logging applications. The scenarios include Call Logging API functions and functions from other libraries, such as the Global Call API, the SRL and the Voice API. For more information about Call Logging API functions, see *Chapter 4. Call Logging Function Reference*. For more on functions from other APIs, refer to the *Global Call API Library Reference* or the *Standard Runtime Library API Library Reference,* as appropriate.

Refer also to *Appendix A* for sample code demonstrating the use of various call logging functions and other API functions in a network monitoring application.

NOTE: Because the Call Logging API is not multithread safe and call logging functions must be called in the same thread, asynchronous is the only programming model to use for call logging applications.

1.8.1. Application Start-Up

Table 3. Application Start-Up Scenario provides the start-up scenario for a typical call logging application. For information on gc_ functions, see the *Global Call API Library Reference*. For information on sr_ functions, see the *Standard Runtime Library API Library Reference*.

Table 3. Application Start-Up Scenario

Function	Description
gc_Start( )	Starts the call logging application using the Global Call API. **gc_Start( )** must be called before **cl_Open( )** once per process.
gc_OpenEx( )	In a loop, opens a Global Call device. The function also returns a unique line device ID to identify the physical device or devices that carry the call. On analog HiZ boards for all applications, you must attach a voice resource device to each network interface device used to monitor analog lines. To do so, call **gc_OpenEx( )** and specify the network interface device name with the ":N_" key and the voice resource device name with the ":V" key. On digital HiZ boards, use this function for call recording applications. There is no need to use this function for call monitoring applications.
gc_GetVoiceH( )	In a loop, if you have called **gc_OpenEx( )**, **gc_GetVoiceH( )** retrieves the voice resource device handle that can be subsequently used to record conversations.
sr_enbhdlr( )	Optional. If desired, enables the event handler for the voice device opened using **gc_OpenEx( )**.

Function	Description
gc_GetXmitSlot( )	In a loop, if you have called **gc_OpenEx( )**, **gc_GetXmitSlot( )** retrieves the transmit time slot number of the network interface devices. These transmit time slot numbers can subsequently be used to record conversations using **dx_mreciottdata( )** (digital HiZ) or **dx_reciottdata( )** (analog HiZ).
cl_Open( )	Opens the call logging device, loads the semantics rules, and returns the call logging device handle. Several analog HiZ devices are gathered and specified in this single device handle. Thus, you will likely call **gc_OpenEx( )** several times in your application and **cl_Open( )** only once.
sr_enbhdlr( )	Enables the call logging event handler for the call logging device.
cl_GetSemanticsStateCount( )	Optional. Gets the number of semantics states.
cl_GetSemanticsStateName( )	Optional in a loop. Gets the names of the semantics states.

1.8.2. Application Termination

Table 4. Application Termination Scenario provides the termination scenario for a typical call logging application.

Table 4. Application Termination Scenario

Function	Description
sr_dishdlr( )	Disables the call logging event handler.
cl_Close( )	Closes the call logging device.

Function	Description
sr_dishdlr( )	If desired, disables the event handler for the voice device at application termination time.
gc_Close( )	In a loop. Call logging applications based on analog HiZ boards and call recording applications should close the Global Call devices at application termination time. This function also closes any voice devices that may have been opened using **gc_OpenEx( )**.
gc_Stop( )	Stops the Global Call application.

1.8.3. Event Handling

Table 5. Event Handling Scenario provides an event handling scenario of a typical call logging application. *Table 6. Event Handling: CLEV_MESSAGE Scenario* provides an event handling scenario in which a CLEV_MESSAGE event is received and call recording takes place.

Table 5. Event Handling Scenario

Function	Description
sr_getevtdev( )	Gets the call logging device handle associated with the current event.
sr_getevttype( )	Identifies the kind of call logging event: CLEV_MESSAGE, CLEV_ALARM or CLEV_ERROR.
sr_getevtdatap( )	Obtains the call logging event data block, CL_EVENTDATA (see *section 1.7.2. Retrieving Event Data*).
cl_GetUsrAttr( )	Gets the user-defined attribute associated with the call logging device.

Table 6. Event Handling: CLEV_MESSAGE Scenario

Function	Description
cl_GetTransaction( ) or **cl_GetTransactionDetails( )**	Gets the call logging transaction ID and other details.
cl_SetTransactionUsrAttr( )	If the ECL_FIRST_MESSAGE bit is set in the iResult field of the call logging event data block*, use this function to associate the user-defined attribute with the transaction.
cl_GetTransactionUsrAttr( )	If the ECL_FIRST_MESSAGE bit is not set in the iResult field of the call logging event data block*, use this function to retrieve the user-defined attribute associated with the transaction.
cl_GetMessage( ) or **cl_GetMessageDetails( )**	If needed, gets the message ID and other details, such as the source of the message, the name of the message, or the human-readable decoded text version of the L3 message (available for digital HiZ only).
cl_GetCalling( )	If needed, gets the calling party number.
cl_GetCalled( )	If needed, gets the called party number.
cl_GetChannel( ) or **cl_GetOrdinalChannel( )**	If needed, gets the digital bearer channel (B channel) or analog channel number, or preferably the ordinal channel number.

Function	Description
dx_mreciottdata( ) or **dx_reciottdata( )**	If the ECL_CONNECT_MESSAGE bit is set in the iResult field of the call logging event data block*, use one of these functions to start call recording. On analog lines, use **dx_reciottdata( )**. On digital lines, use **dx_mreciottdata( )**. To generate record notification beep tone while recording, bitwise-or the RM_NOTIFY value in the mode parameter of the **dx_reciottdata( )** function.
dx_stopch( )	If the ECL_DISCONNECT_MESSAGE bit is set in the iResult field of the call logging event data block*, use this function to complete call recording.
cl_ReleaseTransaction( )	If the ECL_LAST_MESSAGE bit is set in the iResult field of the call logging event data block*, use this function to release the call logging transaction.
* See *section 1.7.2. Retrieving Event Data* for more on the call logging event data block.	

1.9. Device Enumeration

The analog loop start interfaces on the analog HiZ board are treated as a separate device from the voice resource. The **dti** devices represent the loop start interfaces, and the **dxxx** devices represent the voice resources.

The following scenario assumes that the analog HiZ board is the only board in the system.

For the analog HiZ board, such as the DMV160LPHIZ board, device enumeration follows the rules listed below:

- The 16 loop start analog interfaces are enumerated as:

dtiB1T1 to dtiB4T4

Each virtual board has four time slots.

- The 16 voice resource devices are enumerated as:

  dxxxB1C1 to dxxxB4C4

NOTE: In Windows, in a system with DM3 and Springware boards, all DM3 board devices are numbered in sequential order after Springware devices are numbered.

2. Call Logging Demos

This chapter describes the Call Logging demos and provides instructions for running these demos.

2.1. HiZDemo Application

The HiZDemo application is described in the following topics:

2.1.1. HiZDemo Description

The HiZDemo application is a text-based (command line) demo that illustrates call monitoring and call recording functionality on analog HiZ boards and on digital HiZ boards. This demo can be a useful tool for discovering all devices in the system, both HiZ devices and non-HiZ devices.

Using this demo, you can do the following:

- discover all devices available in the system
- select analog HiZ and/or digital HiZ devices to monitor and record calls

- monitor calls (monitoring activity is not displayed on the screen but occurs in the background)
- record calls and save the recordings
- generate record notification beep tone on capable devices

2.1.2. HiZDemo Requirements

The following hardware, software, and equipment requirements must be met before running the HiZDemo application:

- A supported Intel® analog and/or digital HiZ board has been installed in your system.
- For analog HiZ boards, a telephone is connected to a Central Office (CO) or PBX, and this connection is tapped. For an example of hardware configuration, see *Figure 2. Typical Analog High Impedance (HiZ) Configuration*.
- For digital HiZ boards, a T-1 or E-1 connection between a network side and a user side is available, and this connection is tapped. For an example of hardware configuration, see *Figure 1. Typical Digital High Impedance (HiZ) Configuration*.
- The Intel® Dialogic® system release software has been installed and the system requirements for this system release have been met. For more information, see the System Requirements section in the Release Guide for the system release you are using.

2.1.3. Starting the HiZDemo

To start the HiZDemo application, follow these instructions:

1. Open a command prompt window and go to the directory where the demo is located.
2. At the command prompt, type:

```
hizdemo
```

Alternatively, you can double-click on *hizdemo.exe* from Windows Explorer.

The HiZDemo main menu is then displayed.

2.1.4. HiZDemo Menus

This section provides a reference to the high-level HiZDemo menus. Note that every menu has an option to cancel (select 0) and return to a previous menu or if you are at the main menu to exit the application. For information on running the demo, see *section 2.1.5. Running the HiZDemo on Analog HiZ Boards* and *section 2.1.6. Running the HiZDemo on Digital HiZ Boards*.

HiZDemo Main Menu

The HiZDemo main menu has the following menu options:

```
HiZDemo main menu
 1 : Discover and select devices
 2 : Start monitoring and recording calls
 0 : Exit HiZDemo application
```

These choices are described as follows:

- **Discover and select devices:** Select this menu option to discover all devices available on the system. The Devices menu is then displayed.
- **Start monitoring and recording calls:** Select this menu option after you have discovered and selected devices.
- **Exit HiZDemo application:** Select this menu option when you are ready to exit the demo application.

Devices Menu

The Devices menu has the following menu options:

```
Devices menu
 1 : Discover devices available on this system
 2 : Select devices used to monitor and record calls
 3 : Display the current list of selected devices
 0 : Return to HiZDemo main menu
```

These choices are described as follows:

- **Discover devices available on this system:** This menu option discovers all devices available on the system, including analog HiZ boards, digital HiZ boards, and non-HiZ boards. This option retrieves the number of network interface devices and the number of voice devices. It also checks the properties of each device. If a device is a HiZ device, the board, its network interfaces and voice resources will be available for selection in subsequent menus.

  The demo displays messages on the screen about the devices that have been discovered and usable devices. A usable device means that it can be used for call logging purposes. The discovery process distinguishes between HiZ devices and non-HiZ devices, allowing you to run the demo in a chassis with many kinds of boards installed. Voice resources without recording capability are also sorted out as they cannot be used for call recording.
- **Select devices used to monitor and record calls:** Select this menu option after you have discovered devices. The Select Devices menu is displayed.
- **Display the current list of selected devices:** This menu option displays the list of devices that you have selected. The selected devices are listed using the syntax for the **pszDeviceName** parameter of the **cl_Open( )** function.
- **Return to HiZDemo main menu:** Select this menu option to return to the main menu.

Select Devices Menu

The Select Devices menu has the following menu options:

```
Select Devices menu
 1 : Select Analog devices
 2 : Select Digital devices
 0 : Cancel
```

These choices are described as follows:

- **Select Analog devices:** Select this menu option when running the demo on an analog HiZ board.

- **Select Digital devices:** Select this menu option when running the demo on a digital HiZ board.

Select Analog Devices Menu

The Select Analog Devices menu has the following menu options:

```
Select Analog Devices menu
 1 : Select all network interface devices from a range of boards
 2 : Select all network interface devices from a single board
 3 : Select a range of network interface devices
 4 : Select a single network interface device
 5 : Select the voice resources for call recording
 6 : Display the currently selected devices and resources
 7 : Validate the selected devices and resources
 0 : Cancel
```

These choices are described as follows:

- **Select all network interface devices from a range of boards:** Select this menu option to perform call recording on all network interface devices from a range of boards. After you select this option, a series of prompts will appear asking you to:

  Select first analog board in range
  Select analog board range

- **Select all network interface devices from a single board:** Select this menu option to perform call recording on all network interface devices from a single board. After you select this option, a prompt will appear asking you to:

  Select analog board

- **Select a range of network interface devices:** Select this menu option to perform call recording on a range of network interface devices. After you select this option, a series of prompts will appear asking you to:

  Select first analog device in range
  Select analog device range

- **Select a single network interface device:** Select this menu option to perform call recording on a single network interface device. After you select this

option, a prompt will appear asking you to:

Select analog device

- **Select the voice resources for call recording:** Select this menu option to choose the voice resources to be used for call recording. After you select this option, a message informs you that voice resources will be reserved for the selected network interfaces in sequential order, starting from the selected one.
- **Display the currently selected devices and resources:** This menu option displays the HiZ devices and voice resources that you have selected in previous menus.
- **Validate the selected devices and resources:** This menu option checks that you have selected at least one HiZ network interface device and a voice resource. If validation fails, a message is displayed indicating what's missing.
- **Cancel:** This menu option cancels the current selection and returns you to the previous menu.

Select Digital Devices Menu

The Select Digital Devices menu has the following menu options:

```
Select Digital Devices menu
 1 : Select the Protocol
 2 : Select the Network side network interface board
 3 : Select the User side network interface board
 4 : Select the voice resources for call recording
 5 : Display the currently selected protocol, boards and resources
 6 : Validate the selected protocol, boards and resources
 0 : Cancel
```

These choices are described as follows:

- **Select the Protocol:** This menu option selects the protocol that the tapped digital line is using. After you select this option, the Select Digital Protocol menu is displayed.
- **Select the Network side network interface board:** This menu option selects the network interface board for the network side (PSTN). After you select this option, a list of HiZ network interface boards is displayed.

- **Select the User side network interface board:** This menu option selects the network interface board for the user side (CPE). After you select this option, a list of HiZ network interface boards is displayed.
- **Select the voice resources for call recording:** Select this menu option to choose the voice resources to be used for call recording. After you select this option, a message informs you that voice resources will be allocated sequentially, starting from the selected one.
- **Display the currently selected protocol, boards and resources:** This menu option displays the protocol, HiZ boards, and voice resources that you have selected in previous menus.
- **Validate the selected protocol, boards and resources:** This menu option checks that you have selected a protocol, two HiZ network interface boards, and a voice resource. If validation fails, a message is displayed indicating what's missing.
- **Cancel:** This menu option cancels the current selection and returns you to the previous menu.

Select Digital Protocol Menu

The Select Digital Protocol menu has the following menu options:

```
Select Digital Protocol menu
 1 : ISDN
 2 : NET5
 3 : QSIGE1
 4 : 4ESS
 5 : 5ESS
 6 : DMS
 7 : NI2
 8 : NTT
 9 : QSIGT1
 0 : Cancel
```

Choose the appropriate protocol that is being used on the tapped digital line.

2.1.5. Running the HiZDemo on Analog HiZ Boards

The following steps describe one way to run the HiZDemo application on an analog HiZ board.

1. After starting the HiZDemo application, the HiZDemo main menu is displayed. Enter 1 to discover and select devices.
2. The Devices menu is displayed. Enter 1 to discover devices available on this system. This option retrieves and lists the number of network interface devices and the number of voice devices on the system. It also checks the properties of each device. If a device is a HiZ device, the board, its network interfaces and voice resources will be available for selection in subsequent menus. For information on device enumeration on the analog HiZ board, see section *1.9. Device Enumeration*.
3. The Devices menu is displayed. Enter 2 to select devices used to monitor and record calls.
4. The Select Devices menu is displayed. Enter 1 to select analog devices.
5. The Select Analog Devices menu is displayed. In this example, enter 4 to select a single network interface device.

   Note that from the Select Analog Devices menu, you can select all network interface devices from a range of boards, or all network interface devices from a single board, or a range of network interface devices. In this example, select a single network interface device.
6. The Select Analog Device prompt is displayed. This prompt lists the analog HiZ devices that were discovered in step 2. Make your selection, such as 1 for dtiB1T1.
7. The Select Analog Devices menu is displayed. Enter 5 to select the voice resources for call recording.
8. A message informs you that voice resources will be allocated sequentially, starting from the selected one. Select the first voice resource for call recording, such as dxxxB1C1.
9. The Select Analog Devices menu is displayed. Enter 6 to display the currently selected devices and resources for call recording.

   A message informs you of the selected devices and resources.

10. The Select Analog Devices menu is displayed. Enter 7 to validate the selected devices and resources. Note that you must perform the validation step before you can start monitoring and recording calls.

    If validation fails, a message is displayed indicating what's missing; for example, no voice resource was selected. Correct the error until validation is successful. If validation is successful, the current selection process is completed and the Devices menu is displayed.

11. After validation is successful, the Devices menu is displayed. Enter 0 to return to the HiZDemo main menu.

12. The HiZDemo main menu is displayed. Enter 2 to start monitoring and recording calls.

13. You are asked if you want record notification beep tone to be generated on capable devices. If yes, enter 1. If no, enter 0.

14. Several messages are displayed informing you of the status of the demo run, such as:

    ```
    Preparing to monitor and record calls…
    Starting Global Call…
    Opening channels and call logging devices…

    Currently monitoring and recording calls
    ```

    From this point on, new calls observed on the tapped line will be recorded. Note that calls already in progress won't be recorded, because the HiZDemo must first detect that a connection was made. Also note that nothing happens on the screen while calls are being monitored and recorded. Record files are silently created in the demo directory and can be noticed by means of another command prompt window or the Windows Explorer.

15. After call monitoring and recording has begun, you are given an opportunity to stop the process. When you are ready, enter 0 to stop the process.

16. After you have stopped the process, the HiZDemo main menu is displayed. Enter 0 to exit the HiZDemo application. You can review the files that contain the recordings. Look for *cxxryyyy.wav* in the demo directory where *xx* represents the ordinal channel number and *yyyy* represents the sequence number.

2.1.6. Running the HiZDemo on Digital HiZ Boards

The following steps describe one way to run the HiZDemo application on a digital HiZ board.

1. After starting the HiZDemo application, the HiZDemo main menu is displayed. Enter 1 to discover and select devices.
2. The Devices menu is displayed. Enter 1 to discover devices available on this system. This option retrieves and lists the number of network interface devices and the number of voice devices on the system. It also checks the properties of each device. If a device is a HiZ device, the board, its network interfaces and voice resources will be available for selection in subsequent menus.
3. The Devices menu is displayed. Enter 2 to select devices used to monitor and record calls.
4. The Select Devices menu is displayed. Enter 2 to select digital devices.
5. The Select Digital Devices menu is displayed. Enter 1 to select the protocol in use with this board.
6. The Select Digital Protocol menu is displayed. This prompt lists the supported protocols. Make your selection, such as 1 for ISDN.
7. The Select Digital Devices menu is displayed. Enter 2 to specify the network interface board that receives the voice and signaling data transmitted by the network side (CO PSTN).
8. The Select Network side network interface board prompt is displayed. Make your selection from the list of network interface boards.
9. The Select Digital Devices menu is displayed. Enter 3 to specify the network interface board that receives the voice and signaling data transmitted by the user side (CPE PBX).
10. The Select User side network interface board prompt is displayed. Make your selection from the list of network interface boards.
11. The Select Digital Devices menu is displayed. Enter 4 to select the voice resources for call recording.

12. A message informs you that voice resources will be allocated sequentially, starting from the selected one. Select the first voice resource for call recording such as dxxxB1C1.
13. The Select Digital Devices menu is displayed. Enter 5 to display the currently selected protocol, boards, and resources for call recording.

    A message informs you of the selected protocol, boards, and resources.
14. The Select Digital Devices menu is displayed. Enter 6 to validate the selected protocol, boards, and resources. Note that you must perform the validation step before you can start recording calls.

    If validation fails, a message is displayed indicating what's missing; for example, no voice resource was selected. Correct the error until validation is successful. If validation is successful, the current selection process is completed and the Devices menu is displayed.
15. After validation is successful, the Devices menu is displayed. Enter 0 to return to the HiZDemo main menu.
16. The HiZDemo main menu is displayed. Enter 2 to start monitoring and recording calls.
17. You are asked if you want record notification beep tone to be generated on capable devices. If yes, enter 1. If no, enter 0. Note that digital HiZ devices are currently not able to transmit the record notification beep tone to the monitored line.
18. Several messages are displayed informing you of the status of the demo run, such as:

    ```
    Preparing to monitor and record calls…
    Starting Global Call…
    Opening channels and call logging devices…

    Currently monitoring and recording calls
    ```

    From this point on, new calls observed on the tapped line will be recorded. Note that calls already in progress won't be recorded, because the HiZDemo must first detect that a connection was made. Also note that nothing happens on the screen while calls are being monitored and recorded. Record files are silently created in the demo directory and can be noticed by means of another command prompt window or the Windows Explorer.

19. After call monitoring and recording has begun, you are given an opportunity to stop the process. When you are ready, enter 0 to stop the process.
20. After you have stopped the process, the HiZDemo main menu is displayed. Enter 0 to exit the HiZDemo application. You can review the files that contain the recordings. Look for *cxxryyyy.wav* in the demo directory where *xx* represents the ordinal channel number and *yyyy* represents the sequence number.

2.1.7. Functions Used by the HiZDemo

Table 7 provides a list of the Call Logging API functions that are called by the HiZDemo application. The function name and source file name are also specified for easier reference. The files are located in *\program files\dialogic\demos\hiz\hizdemo*.

Table 7. Call Logging Functions Used by HiZDemo

HiZDemo Function name and Source File Name	Call Logging API Function
StartMonitoring( ) in monitor.c	**cl_Open( )**
StopMonitoring( ) in monitor.c	**cl_Close( )**
CallLoggingEventHandler( ) in monitor.c	**cl_GetUsrAttr( )** **cl_GetOrdinalChannel( )** **cl_GetTransaction( )** **cl_ReleaseTransaction( )**

2.1.8. Files Used by the HiZDemo

Table 8 lists the files used by the HiZDemo application. The files are located in *\program files\dialogic\demos\hiz\hizdemo*.

Table 8. Files Used by HiZDemo

File Name	Description
c*xx*r*yyyy*.wav	recording output in wave file format generated after the demo is run where *xx* represents the ordinal channel number (hexadecimal) and *yyyy* represents the sequence number (hexadecimal)
channel.c	HiZDemo channel handling source code
channel.h	HiZDemo channel handling header file
device.c	HiZDemo device discovery and select source code
device.h	HiZDemo device discovery and select header file
HiZdemo.c	HiZDemo main source code
HiZdemo.dsp	HiZDemo project file
HiZdemo.dsw	HiZDemo work space
HiZdemo.exe	HiZDemo executable file
HiZdemo.h	HiZDemo main header file
menu.c	HiZDemo menus and prompts source code
menu.h	HiZDemo menus and prompts header file
monitor.c	HiZDemo call monitoring and recording source code
monitor.h	HiZDemo call monitoring and recording header file

2.2. SnifferMFC Demo

The SnifferMFC demo application is described in the following topics:

- 2.2.1. SnifferMFC Demo Description
- 2.2.2. SnifferMFC Demo Requirements
- 2.2.3. Starting the SnifferMFC Demo

2.2.1. SnifferMFC Demo Description

The SnifferMFC demo application is built with a graphical user interface (GUI) based on the Microsoft Foundation Classes (MFC). The SnifferMFC demo illustrates call monitoring and call logging functionality on analog HiZ boards and on digital HiZ boards. This demo is a useful debugging tool as every event observed on the line is reported on the screen.

Using the SnifferMFC demo, you can do the following:

- select analog HiZ or digital HiZ devices to monitor and log calls
- monitor calls
- view the results of call monitoring on the screen
- on digital HiZ boards, generate an ISDN trace file that can be used in conjunction with **cl_DecodeTrace( )**
- on digital HiZ boards, decode ISDN trace files previously generated on the board

NOTES: **1.** Unlike the HiZDemo, the SnifferMFC demo does not discover all devices available on the system and distinguish HiZ devices from non-HiZ devices. You must gather HiZ device information before you run the SnifferMFC demo.

2. Do not run the SnifferMFC demo for long periods of time, because the information displayed requires more and more memory.

2.2.2. SnifferMFC Demo Requirements

The following hardware and software requirements must be met before running the SnifferMFC demo application:

- A supported Intel® analog and/or digital HiZ board has been installed in your system.
- For analog HiZ boards, a telephone connected to a Central Office (CO) or PBX is available, and this connection is tapped.
- For digital HiZ boards, a T-1 or E-1 connection between a network side and a user side is available, and this connection is tapped.
- The Intel® Dialogic® system release software has been installed and the system requirements for this system release have been met. For more information, see the System Requirements section in the Release Guide for the system release you are using.

2.2.3. Starting the SnifferMFC Demo

To start the SnifferMFC application, follow these instructions:

1. Open Windows Explorer and go to the directory where the demo is located.
2. Double-click on *sniffermfc.exe*.

The SnifferMFC main window is displayed.

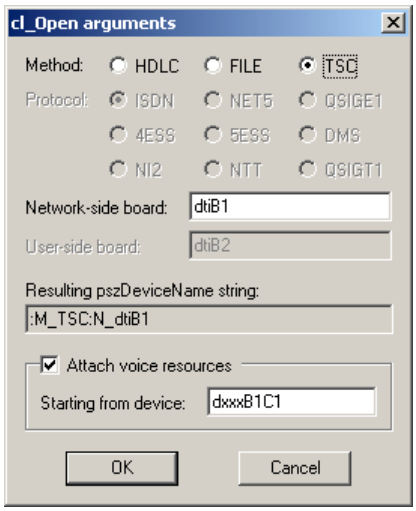
2.2.4. Running the SnifferMFC Demo on Analog HiZ Boards

The following steps describe one way to run the SnifferMFC demo application on an analog HiZ board.

1. After starting the application, the main menu is displayed.
   Select Sniffer > Open.
2. The cl_Open arguments window is displayed.
   - Click TSC for method.
   - Enter the network-side board such as dtiB1. To monitor calls on a range of boards, use a dash; for example, specify dtiB1-dtiB4.

You will notice the resulting pszDeviceName string used by the **cl_Open( )** function.

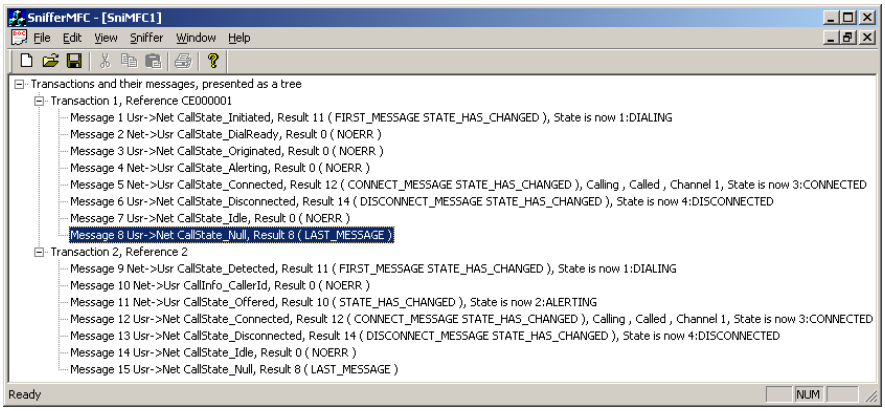
- When TSC is selected as the method, the Attach voice resources check box is automatically selected.
- Enter the first voice resource for call logging such as dxxxB1C1.
- Click OK.



3. A dialog box appears asking if you want to see the list of semantics states. Click Yes or No as desired.

   If you click Yes, the list of semantics states is displayed. Click OK to continue.
   If you click No, continue to step 4.

4. The SnifferMFC main window is displayed and call logging transactions are presented on the screen.



To stop the SnifferMFC demo, select Sniffer > Close, then File > Exit.

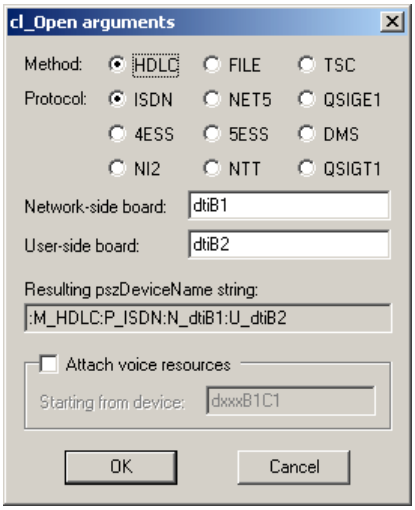
2.2.5. Running the SnifferMFC Demo on Digital HiZ Boards

The following steps describe one way to run the SnifferMFC demo application on a digital HiZ board.

1. After starting the application, the main menu is displayed.
   Select Sniffer > Open.
2. The cl_Open arguments window is displayed.
3. Choose the HDLC method or the FILE method.

   HDLC method:

   - Click HDLC for method.
   - Click the appropriate protocol such as ISDN.
   - Enter the network-side board such as dtiB1.
   - Enter the user-side board such as dtiB2.
   - You will notice the resulting pszDeviceName string used by the **cl_Open( )** function.
   - Click OK.



OR

FILE method:

- Click FILE to decode ISDN trace files previously generated on digital HiZ boards.
- Click the appropriate protocol.
- You will notice the resulting pszDeviceName string used by the **cl_Open( )** function.
- The network-side board, the user-side board, and the voice resources values are not used.
- Click OK.

4. The SnifferMFC main window is displayed and call logging transactions are presented on the screen.

To stop the demo, select Sniffer > Close, then File > Exit.

2.2.6. Functions Used by the SnifferMFC Demo

Table 9 provides a list of the Call Logging API functions that are called by the SnifferMFC demo application. The function name and source file name are also specified for easier reference.

Table 9. Call Logging Functions Used by the SnifferMFC Demo

SnifferMFC Function name and Source File Name	**Call Logging API Function**
CSnifferMFCDoc::OnSnifferOpen( ) in SnifferMFCDoc.cpp	**cl_GetSemanticsStateCount( )** **cl_GetSemanticsStateName( )** **cl_Open( )**
CSnifferMFCOpenDlg::UpdateDeviceName( ) in SnifferMFCOpenDlg.cpp	construction of the **pszDeviceName** parameter of **cl_Open( )**
CSnifferMFCDoc::AttachVoiceResourcesTo NetworkChannels( ) in SnifferMFCDoc.cpp	takes care of attaching voice resource devices to network interface devices
CSnifferMFCDoc::OnSnifferClose( ) in SnifferMFCDoc.cpp	**cl_Close( )**

SnifferMFC Function name and Source File Name	**Call Logging API Function**
CSnifferMFCDoc::EventHandler( ) in SnifferMFCDoc.cpp	**cl_GetCalled( )** **cl_GetCalling( )** **cl_GetChannel( )** **cl_GetMessage( )** **cl_GetMessageDetails( )** **cl_GetTransaction( )** **cl_GetTransactionDetails( )** **cl_GetUsrAttr( )** **cl_GetVariable( )** **cl_ReleaseTransaction( )**
CSnifferMFCDoc::OnSnifferDecodetrace( ) in SnifferMFCDoc.cpp	**cl_DecodeTrace( )**
CSnifferMFCDoc::OnSnifferStarttrace( ) in SnifferMFCDoc.cpp	**cl_StartTrace( )**
CSnifferMFCDoc::OnSnifferStoptrace( ) in SnifferMFCDoc.cpp	**cl_StopTrace( )**

2.2.7. Files Used by the SnifferMFC Demo

Table 10 lists the files used by the SnifferMFC demo. The files are located in *\program files\dialogic\demos\hiz\sniffermfc*.

Table 10. Files Used by SnifferMFC Demo

File Name	Description
ChildFrm.cpp	SnifferMFC child window handling source code
ChildFrm.h	SnifferMFC child window handling header file
ISDN.log	SnifferMFC sample ISDN trace file
MainFrm.cpp	SnifferMFC main window handling source code
MainFrm.h	SnifferMFC main window handling header file
resource.h	SnifferMFC resource definition header file
SnifferMFC.clw	SnifferMFC ClassWizard file
SnifferMFC.cpp	SnifferMFC main source code
SnifferMFC.dsp	SnifferMFC project file
SnifferMFC.dsw	SnifferMFC work space
SnifferMFC.exe	SnifferMFC sample program binary
SnifferMFC.h	SnifferMFC main header file
SnifferMFC.ico	SnifferMFC main icon
SnifferMFC.rc	SnifferMFC resource script
SnifferMFC.rc2	SnifferMFC secondary resource script
SnifferMFCDoc.cpp	SnifferMFC document handling header file
SnifferMFCDoc.h	SnifferMFC document handling header file
SnifferMFCDoc.ico	SnifferMFC document icon
SnifferMFCOpenDlg.cpp	SnifferMFC open device dialog handling source code
SnifferMFCOpenDlg.h	SnifferMFC open device dialog handling header file
SnifferMFCView.cpp	SnifferMFC view handling source code

File Name	Description
SnifferMFCView.h	SnifferMFC view handling header file
StdAfx.cpp	SnifferMFC precompilation source code
StdAfx.h	SnifferMFC precompilation header file
Toolbar.bmp	SnifferMFC toolbar bitmap

3. Call Logging Function Overview

This chapter provides an overview of the Call Logging API functions that are used to develop and run call monitoring and call recording applications and a section on error handling.

3.1. Call Logging Function Categories

The call logging functions can be divided into three categories:

- Device-based functions – functions that affect the status of a device or that set or get information related to a particular device
- Transaction-based functions – functions that set or get information related to a particular call logging transaction
- Event-based functions – functions that get information that has been sent to and stored in the event data block after a CLEV_MESSAGE event was generated.

Table 11, Table 12, and Table 13 categorize the functions accordingly and provide brief descriptions of each of the call logging functions. Some functions that are specific to common channel signaling (CCS) protocols are not supported on analog HiZ boards. For detailed descriptions of the functions, see *Chapter 4. Call Logging Function Reference*.

Table 11. Device-based Call Logging Functions

Function	Description
cl_Close( )	closes a previously opened call logging device
cl_DecodeTrace( )	decodes a previously recorded L2 frame trace file and posts the call logging events to the SRL
cl_GetSemanticsStateCount( )	gets the number of semantics states

Function	Description
cl_GetSemanticsStateIndex( )	gets the index of a semantics state from its name
cl_GetSemanticsStateName( )	gets the name of a semantics state from its index
cl_GetUsrAttr( )	gets the user-defined attribute for a call logging device
cl_Open( )	opens a call logging device
cl_SetUsrAttr( )	sets the user-defined attribute for a call logging device
cl_StartTrace( )	starts recording an L2 frames trace file
cl_StopTrace( )	stops recording an L2 frames trace file

Table 12. Transaction-based Call Logging Functions

Function	Description
cl_GetTransaction( )	gets the call logging transaction ID
cl_GetTransactionDetails( )	gets the call logging transaction ID and other details
cl_GetTransactionUsrAttr( )	gets the user-defined attribute for a call logging transaction
cl_PeekCalled( )	gets the called party number, at function call time
cl_PeekCalling( )	gets the calling party number, at function call time
cl_PeekChannel( )	gets the channel number, at function call time
cl_PeekOrdinalChannel( )	gets the ordinal channel number, at function call time

Function	Description
cl_PeekVariable( )	gets the value of a semantics-defined variable, at function call time
cl_ReleaseTransaction( )	releases the call logging transaction
cl_SetTransactionUsrAttr( )	sets the user-defined attribute for a call logging transaction

Table 13. Event-based Call Logging Functions

Function	Description
cl_GetCalled( )	gets the called party number, at event time
cl_GetCalling( )	gets the calling party number, at event time
cl_GetChannel( )	gets the channel number, at event time
cl_GetOrdinalChannel( )	gets the ordinal channel number, at event time
cl_GetMessage( )	gets the message ID
cl_GetMessageDetails( )	gets the message ID and other details
cl_GetTransaction( )	gets the call logging transaction ID
cl_GetTransactionDetails( )	gets the call logging transaction ID and other details
cl_GetVariable( )	gets the value of a semantics-defined variable, at event time

3.2. Error Handling

Call Logging functions return a negative value (test for <0) for failure and in most cases you use the SRL Standard Attribute function **ATDV_LASTERR( )** to obtain the error code. The **cl_Open( )** function cannot use this error handling method, however, because if it fails it does not return a device handle, which is a

required parameter for **ATDV_LASTERR( ).** Instead, **cl_Open( )** error codes are returned in the **errno** global variable.

Table 14 shows the list of possible error codes that can be returned. The Errors section of each function description lists the error codes that apply to that particular function.

Table 14. Call Logging Function Errors

Error Code Value	Description
ECL_NOMEM	out of memory
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDPARAMETER	invalid parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_UNSUPPORTED	function not supported
ECL_FILEOPEN	fopen failed
ECL_FILECLOSE	fclose failed
ECL_FILEREAD	fread failed
ECL_FILEWRITE	fwrite failed
ECL_TRACESTARTED	trace already started
ECL_TRACENOTSTARTED	trace not started
ECL_INVALIDDEVICE	invalid device handle
ECL_INTERNAL	internal Call Logging error; cause unknown
ECL_GCOPENEX_NETWORK *	gc_OpenEx( ) failed on the network side
ECL_GCOPENEX_USER *	gc_OpenEx( ) failed on the user side

ECL_DTOPEN_BOARD	dt_open( ) failed
ECL_ATDV_SUBDEVS_BOARD	ATDV_SUBDEVS( ) failed
ECL_HIZOPEN_CHANNEL	hiz_open( ) failed
*The following additional flag is set in these error code values to indicate that the error occurred while the Call Logging API was calling a Global Call function:	
ECL_FLAG_INSIDE_GC	use gc_ErrorValue( ) for an additional error description
The error codes returned by the Call Logging API may be explicit enough, but for more information about the Global Call gc_ functions and error code values they return, see the *Global Call API Library Reference*.	

Some Call Logging functions will return -2 if they fail because the transaction was already released. **ATDV_LASTERR( )** returns ECL_TRANSACTIONRELEASED in these cases.

ECL_INVALIDPARAMETER is returned when a parameter is fully or partially invalid. Examples of situations in which this error code can be returned are:

- a field in the pszDeviceName parameter of **cl_Open( )** is incorrect
- the state index specified by the iSemanticsStateIndex parameter of **cl_GetSemanticsStateName( )** is out of bounds
- the variable specified by the pszVariableName parameter of **cl_GetVariable( )** or **cl_PeekVariable( )** does not exist

ECL_INVALIDCONTEXT is returned if the Call Logging function can only be called while processing a CLEV_MESSAGE event and the function is called at another time or with a different device than the one for which the event was posted.

ECL_UNSUPPORTED is returned if the Call Logging function is not supported under the current conditions. For example, **cl_DecodeTrace( )** can only be called when the FILE method was specified in the pszDeviceName parameter of **cl_Open( ).**

ECL_FILEOPEN and ECL_FILEREAD are returned when fopen and fread fail. This can happen when **cl_DecodeTrace( )** is called if the pszFileName parameter specifies a file that does not exist or a file that does not have the required format.

4. Call Logging Function Reference

This chapter provides a detailed description of each Call Logging function included in the *cllib.h* file; functions are presented in alphabetical order.

This chapter also includes the following information:

- function documentation – the function description format
- general function syntax – the programming convention format

4.1. Function Documentation

The Call Logging API functions are listed alphabetically in the remainder of this chapter. The format for each function description is as follows:

- **Function header** – Lists the function name and briefly states the purpose of the function.
    - **Name** – Defines the function name and function syntax using standard C language syntax.
    - **Inputs** – Lists all input parameters using standard C language syntax.
    - **Returns** – Lists all returns of the function.
    - **Includes** – Lists all include files required by the function.
    - **Mode** – Asynchronous or synchronous.
    - **Platform** – Indicates which platforms (DM3 or Springware) are supported for each function. The term "DM3 boards" refers to products that are based on the Intel® DM3 mediastream architecture. The term "Springware boards" refers to boards based on earlier-generation architecture.

      Note: Although the Call Logging API is supported on DM3 boards only, this information is provided for consistency across all libraries.
- **Description paragraph** – Provides a description of function operation, including parameter descriptions.

- **Termination Event paragraph** – Describes the event(s) returned to indicate the termination of the function. Termination events are returned by asynchronous functions only.
- **Cautions paragraph** – Provides warnings and reminders.
- **Example paragraph** – Provides C language coding example(s) showing how the function can be used in application code.
- **Errors paragraph** – Lists specific error codes for each function.
- **See Also paragraph** – Provides a list of related functions.

4.2. General Function Syntax

The Call Logging functions use the following format:

cl_function(reference, parameter1, parameter2, …, parameterN)

where:

- **cl_function** – is the function name.
- **reference** – is an input field that directs the function to a specific call logging device or call logging transaction.
- **parameters** – are input or output fields.

Name:	int cl_Close(hDevice)	
Inputs:	long hDevice	• Call Logging device handle
Returns:	0 on success -1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

Description

The **cl_Close( )** function closes a previously opened call logging device that was opened using the **cl_Open( )** function.

Parameter	Description
hDevice	The device handle of the call logging device to be closed.

Termination Events

None

Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

Example

```
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

typedef struct
{
      const char* pszProtocol;
      const char* pszNetworkDeviceName;
      const char* pszUserDeviceName;
} DEVICEUSRATTR;

extern long EventHandler(unsigned long hEvent);
```

```
extern DEVICEUSRATTR* GetDeviceUsrAttr(long hDevice);

void ExitApplication(void)
{
      DEVICEUSRATTR* pDeviceUsrAttr;

      if (g_hDevice != EV_ANYDEV)
      {
            if (sr_dishdlr(g_hDevice, EV_ANYEVT, EventHandler) != 0)
            {
                  printf("ExitApplication - sr_dishdlr() failed\n");
            }

            pDeviceUsrAttr = GetDeviceUsrAttr(g_hDevice);
            free(pDeviceUsrAttr);

            if (cl_Close(g_hDevice) != 0)
            {
                  printf("ExitApplication - cl_Close() failed\n");
            }

            g_hDevice = EV_ANYDEV;

            if (gc_Stop() != GC_SUCCESS)
            {
                  printf("ExitApplication - gc_Stop() failed\n");
            }
      }
}
```

Errors

None

See Also

- **cl_GetUsrAttr( )**
- **cl_SetUsrAttr( )**

Name:	int cl_DecodeTrace(hDevice, pszFileName)	
Inputs:	long hDevice	• Call Logging device handle
	const char* pszFileName	• pointer to ASCIIZ string
Returns:	0 on success	
	-1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_DecodeTrace( )** function is not supported on analog HiZ boards.

The **cl_DecodeTrace( )** function decodes a previously recorded L2 frames trace file and posts the call logging events to the SRL. The events are posted as if the L2 frames recorded in the trace file were actually monitored on the line when they were occurring. Since trace files do not contain time information, the call logging events are generated at a high rate.

The **cl_DecodeTrace( )** function is used primarily for testing. The SnifferMFC demo provides an ISDN log file that can be used in conjunction with **cl_DecodeTrace( ).**

Parameter	Description
hDevice	The device handle of the call logging device.
pszFileName	A pointer to the ASCIIZ string that specifies the path and name of the recorded L2 frames trace file.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

- This function can be called only for call logging devices for which the FILE method was specified in the **pszDeviceName** parameter when the device was opened. See the **cl_Open( )** function description for more information.

Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void DecodeTraceFile(const char* pszTraceFileName)
{
      if (g_hDevice != EV_ANYDEV)
      {
            if (cl_DecodeTrace(g_hDevice, pszTraceFileName) != 0)
            {
                  printf("DecodeTraceFile - cl_DecodeTrace() failed\n");
            }
      }
}
```

Errors

If the function returns a value <0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_FILEOPEN	fopen failed
ECL_FILEREAD	fread failed
ECL_UNSUPPORTED	function not supported
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_Open( )**

Name:	int cl_GetCalled(hDevice, pclEventData, pszCalled, iCalledSize)	
Inputs:	long hDevice	• Call Logging device handle
	CL_EVENTDATA* pclEventData	• pointer to the call logging event data block
	char* pszCalled	• pointer to the buffer for the called party number
	int iCalledSize	• size of the buffer for the called party number
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetCalled( )** function gets the called party number, at event time. The value returned is that of the semantics-defined CALLED variable. The function returns the called party number as the number would have appeared at the time the CLEV_MESSAGE event was generated.

Parameter	Description
hDevice	The device handle of the call logging device.
pclEventData	A pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *1.7.2. Retrieving Event Data* for more on the event data block.
pszCalled	The pointer to the buffer into which the called party number is returned as an ASCIIZ string. If the called party number is not available, the function will return with an empty string.
iCalledSize	The size of the buffer into which the called party number is returned, where the maximum size includes the terminating NUL of the ASCIIZ string.

■ Termination Events

None

Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

Example

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

void GetCalled_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      char szCalled[32];

      iRet = cl_GetCalled(hDevice, pclEventData, szCalled, sizeof(szCalled));
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetCalled_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetCalled_WithinEventHandler - cl_GetCalled() failed\n");
            }
            return;
      }

      printf("Called party number is: \"%s\"\n", szCalled);
}
```

Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory

ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_ReleaseTransaction( )**
- **cl_GetVariable( )**
- **cl_PeekCalled( )**

Name:	int cl_GetCalling(hDevice, pclEventData, pszCalling, iCallingSize)	
Inputs:	long hDevice	• Call Logging device handle
	CL_EVENTDATA* pclEventData	• pointer to the call logging event data block
	char* pszCalling	• pointer to the buffer for the calling party number
	int iCallingSize	• size of the buffer for the calling party number
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetCalling( )** function gets the calling party number, at event time. The value returned is that of the semantics-defined CALLING variable. The function returns the calling party number as the number would have appeared at the time the CLEV_MESSAGE event was generated.

Parameter	Description
hDevice	The device handle of the call logging device.
pclEventData	A pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *1.7.2. Retrieving Event Data* for more information.
pszCalling	The pointer to the buffer into which the calling party number is returned as an ASCIIZ string. If the calling party number is not available, the function will return with an empty string.
iCallingSize	The size of the buffer into which the calling party number is returned, where the maximum size includes the terminating NUL of the ASCIIZ string.

■ Termination Events

None

Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

Example

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

void GetCalling_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      char szCalling[32];

      iRet = cl_GetCalling(hDevice, pclEventData, szCalling, sizeof(szCalling));
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetCalling_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetCalling_WithinEventHandler - cl_GetCalling() failed\n");
            }
            return;
      }

      printf("Calling party number is: \"%s\"\n", szCalling);
}
```

Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory

ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_ReleaseTransaction( )**
- **cl_GetVariable( )**
- **cl_PeekCalling( )**

Name:	int cl_GetChannel(hDevice, pclEventData, pszChannel, iChannelSize)	
Inputs:	long hDevice	• Call Logging device handle
	CL_EVENTDATA* pclEventData	• pointer to the call logging event data block
	char* pszChannel	• pointer to the buffer for the channel number
	int iChannelSize	• size of the buffer for the channel number
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetChannel( )** function gets the channel number, at event time. The value returned is that of the semantics-defined CHANNEL variable. The function returns the channel number as the number would have appeared at the time the CLEV_MESSAGE event was generated.

Parameter	Description
hDevice	The device handle of the call logging device.
pclEventData	A pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.
pszChannel	The pointer to the buffer into which the channel number is returned as an ASCIIZ string. For analog HiZ boards, this string contains a number between 1 and the number of analog HiZ devices specified in the **pszDeviceName** parameter of the **cl_Open( )** function.
iChannelSize	The size of the buffer into which the channel number is returned, where the maximum size includes the terminating NUL of the ASCIIZ string.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.
- The range of channel numbers depends on the semantics rules and does not necessarily match Intel device name numbering. For example, E-1 ISDN channel numbers range from 1 to 15 and from 17 to 31, while the Intel device names on an E-1 board range from “dtiBxT1” to “dtiBxT30”.

■ Example

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

void GetChannel_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      char szChannel[8];

      iRet = cl_GetChannel(hDevice, pclEventData, szChannel, sizeof(szChannel));
```

```
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetChannel_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetChannel_WithinEventHandler - cl_GetChannel() failed\n");
            }
            return;
      }

      printf("Bearer channel number is: \"%s\"\n", szChannel);
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_ReleaseTransaction( )**
- **cl_GetVariable( )**
- **cl_GetOrdinalChannel( )**
- **cl_PeekChannel( )**
- **cl_PeekOrdinalChannel( )**

Name:	int cl_GetMessage(hDevice, pidMessage, pclEventData)
Inputs:	long hDevice • call logging device handle long* pidMessage • pointer to the message ID CL_EVENTDATA* pclEventData • pointer to the call logging event data block
Returns:	0 on success -1 on failure -2 if call logging transaction already released
Includes:	cllib.h
Mode:	synchronous
Platform:	DM3

■ Description

The **cl_GetMessage( )** function returns the ID of a message for which a CLEV_MESSAGE event was generated. The returned message ID is unique and protocol dependent.

Parameter	Description
hDevice	The device handle of the call logging device.
pidMessage	A pointer to the returned message ID.
pclEventData	A pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

Example

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

void GetMessage_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      long idMessage;

      iRet = cl_GetMessage(hDevice, &idMessage, pclEventData);
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetMessage_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetMessage_WithinEventHandler - cl_GetMessage() failed\n");
            }
            return;
      }

      printf("Message ID=%08X\n", idMessage);
}
```

Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_ReleaseTransaction( )**
- **cl_GetMessageDetails( )**

Name:	int cl_GetMessageDetails(hDevice, pidMessage, pclEventData, piSource, pszName, iNameSize, pszTraceText, iTraceTextSize)	
Inputs:	long hDevice	• call logging device handle
	long* pidMessage	• pointer to message ID
	CL_EVENTDATA* pclEventData	• pointer to call logging event data block
	int* piSource	• pointer to code that identifies sender side of message
	char* pszName	• pointer to buffer for message name
	int iNameSize	• size of buffer for message name
	char* pszTraceText	• pointer to buffer for decoded message text
	int iTraceTextSize	• size of buffer for decoded message text
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetMessageDetails( )** function returns the ID and details of a message for which a CLEV_MESSAGE event was generated. Optional details about the message that can be returned include the message source, the message name, and the human-readable decoded text based on the protocol message. Pass NULL as the related parameter for any details that are not needed.

The returned message ID is unique and protocol-dependent. The value of the returned code is either CL_SOURCE_NETWORK or CL_SOURCE_USER.

Parameter	Description
hDevice	The device handle of the call logging device.
pidMessage	The pointer to the returned message ID.
pclEventData	The pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.
piSource	The pointer to the returned code that identifies the sender side of the message. The value of the returned code is either CL_SOURCE_NETWORK or CL_SOURCE_USER
pszName	The pointer to the buffer into which the name of the message is returned as an ASCIIZ string.
iNameSize	The size of the buffer into which the name of the message is returned, where the maximum size includes the terminating NUL of the ASCIIZ string.
pszTraceText	The pointer to the buffer into which the human-readable decoded text of the message is returned as an ASCIIZ string (digital HiZ boards only).
iTraceTextSize	The size of the buffer into which the human-readable decoded text of the message is returned, where the maximum size includes the terminating NUL of the ASCIIZ string (digital HiZ boards only).

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.
- The names of the messages are protocol dependent.

■ Example

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

const char* ToText_MessageSource(int iMessageSource)
{
      if (iMessageSource == CL_SOURCE_NETWORK)
      {
            return "Network-side";
      }
      else if (iMessageSource == CL_SOURCE_USER)
      {
            return "User-side";
      }
      return "Unknown";
}

void GetMessageDetails_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      long idMessage;
      int iMessageSource;
      char szMessageName[32];
      char szMessageTraceText[4096];

      iRet = cl_GetMessageDetails(hDevice, &idMessage, pclEventData, &iMessageSource,
szMessageName, sizeof(szMessageName), szMessageTraceText, sizeof(szMessageTraceText));
      if (iRet != 0)
      {
            if (iRet == -2)
           {
                  printf("GetMessageDetails_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetMessageDetails_WithinEventHandler - cl_GetMessageDetails()
failed\n");
            }
            return;
      }

      printf("Message ID=%08X \"%s\" sent by %s(%i):\n%s\n", idMessage, szMessageName,
ToText_MessageSource(iMessageSource), iMessageSource, szMessageTraceText);
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter

ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_ReleaseTransaction( )**
- **cl_GetMessage ( )**

Name:	int cl_GetOrdinalChannel(hDevice, pclEventData, piOrdinalChannel)	
Inputs:	long hDevice	• call logging device handle
	CL_EVENTDATA* pclEventData	• pointer to the call logging event data block
	int* piOrdinalChannel	• pointer to the returned ordinal number of the channel
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetOrdinalChannel( )** function gets the ordinal channel number, at event time. The value returned is the ordinal number of the channel among those monitored by the call logging device. The ordinal channel number is an integer between 0 and the number of monitored channels minus one. The function returns the ordinal channel number as it would have appeared at the time the CLEV_MESSAGE event was generated.

If you previously called **cl_GetChannel( )** to access your own channel-related data stored in an array, then the new **cl_GetOrdinalChannel( )** function is a recommended replacement. As it returns an integer between 0 and the number of monitored channels minus one, this function is similar to **cl_GetChannel( )** and should be used in its place. This new function, introduced in System Release 6.0, allows direct access into any array of channel-related structures that you wish to allocate. This is particularly useful in E-1 trunk environments where the value returned by **cl_GetChannel( )** ranges from 1 to 15 and 17 to 31. The new **cl_GetOrdinalChannel( )** function allows you to build a unified method to handle monitored channels, whether they sit on T-1, E-1, or analog lines.

Parameter	Description
hDevice	The device handle of the call logging device.
pclEventData	A pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.
piOrdinalChannel	The pointer to the ordinal number of the channel, returned as an integer between 0 and the number of monitored channels minus one.

■ Termination Events

None.

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

■ Example

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

void GetOrdinalChannel_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
  int iRet;
  int iOrdinalChannel;

  iRet = cl_GetOrdinalChannel(hDevice, pclEventData, &iOrdinalChannel);
  if (iRet != 0)
  {
    if (iRet == -2)
    {
      printf("GetOrdinalChannel_WithinEventHandler - Transaction already released\n");
    }
    else
    {
      printf("GetOrdinalChannel_WithinEventHandler - cl_GetOrdinalChannel() failed\n");
    }
    return;
  }

  printf("Channel index (or ordinal number) is: %i\n", iOrdinalChannel);
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetChannel( )**
- **cl_PeekOrdinalChannel( )**

Name:	int cl_GetSemanticsStateCount(hDevice, piSemanticsStateCount)	
Inputs:	long hDevice	• call logging device handle
	int* piSemanticsStateCount	• pointer to returned number
Returns:	0 on success -1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetSemanticsStateCount( )** function returns the number of semantics states defined in the protocol-specific semantics for a specified call logging device. Semantics states, such as dialing, connected or disconnected, represent the current status of the monitored call.

The list of semantics states (count, names, and indexes) is protocol-dependent. Semantics states are indexed from 0 to the number of semantics states minus one.

Parameter	Description
hDevice	The device handle of the call logging device.
piSemanticsStateCount	The pointer to the returned number of semantics states for this call logging device.

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>
```

```
/* The Call Logging Device Handle */
extern long g_hDevice;

void GetSemanticsStates(void)
{
      int nStates;
      int iState;
      char szState[64];

      if (g_hDevice != EV_ANYDEV)
      {
            if (cl_GetSemanticsStateCount(g_hDevice, &nStates) != 0)
            {
                  printf("GetSemanticsStates - cl_GetSemanticsStateCount() failed\n");
                  return;
            }

            printf("There are %i semantics states:\n", nStates);

            for (iState = 0 ; (iState < nStates) ; ++iState)
            {
                  if (cl_GetSemanticsStateName(g_hDevice, iState, szState,
sizeof(szState)) != 0)
                  {
                        printf("GetSemanticsStates - cl_GetSemanticsStateName()
failed\n");
                        return;
                  }

                  printf("%i: \"%s\"\n", iState, szState);
            }
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetSemanticsStateName( )**

Name:	int cl_GetSemanticsStateName(hDevice, iSemanticsStateIndex, pszSemanticsStateName, iSemanticsStateNameSize)	
Inputs:	long hDevice	• call logging device handle
	int iSemanticsStateIndex	• index of semantics state
	char* pszSemanticsStateName	• pointer to buffer into which semantics state name is returned
	int iSemanticsStateNameSize	• size of buffer into which name of semantics state name is returned
Returns:	0 on success -1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetSemanticsStateName( )** function returns the name of a semantics state according to its index. Semantics states, such as dialing, connected or disconnected, represent the current status of the monitored call. The names of semantics states are returned as ASCIIZ strings.

Semantics states are indexed from 0 to the number of semantics states minus one. The list of semantics states (count, names, and indexes) is protocol dependent.

Parameter	Description
hDevice	The device handle of the call logging device.
iSemanticsStateIndex	The index of the semantics state.
pszSemanticsStateName	The pointer to the buffer into which the name of the indexed semantics state is to be returned. The name is returned as an ASCIIZ string.
iSemanticsStateNameSize	The size of the buffer into which the name of the indexed semantics state is to be returned, where the maximum size includes the terminating NUL of the returned ASCIIZ string.

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void GetSemanticsStates(void)
{
      int nStates;
      int iState;
      char szState[64];

      if (g_hDevice != EV_ANYDEV)
      {
            if (cl_GetSemanticsStateCount(g_hDevice, &nStates) != 0)
            {
                  printf("GetSemanticsStates - cl_GetSemanticsStateCount() failed\n");
                  return;
            }

            printf("There are %i semantics states:\n", nStates);

            for (iState = 0 ; (iState < nStates) ; ++iState)
```

```
            {
                  if (cl_GetSemanticsStateName(g_hDevice, iState, szState,
sizeof(szState)) != 0)
                  {
                        printf("GetSemanticsStates - cl_GetSemanticsStateName()
failed\n");
                        return;
                  }

                  printf("%i: \"%s\"\n", iState, szState);
            }
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	**Returned When**
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDPARAMETER	invalid parameter
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetSemanticsStateCount( )**

Name:	int cl_GetTransaction(hDevice, pidTransaction, pclEventData)	
Inputs:	long hDevice	• call logging device handle
	long* pidTransaction	• pointer to returned transaction ID
	CL_EVENTDATA* pclEventData	• pointer to call logging event data block
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetTransaction( )** function returns the ID of a call logging transaction for which a CLEV_MESSAGE event was generated. The returned call logging transaction ID is unique and protocol dependent.

Parameter	Description
hDevice	The device handle of the call logging device.
pidTransaction	The pointer to the returned call logging transaction ID.
pclEventData	The pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information on the event data block.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>

typedef struct
{
      const char* pszProtocol;
      const char* pszNetworkDeviceName;
      const char* pszUserDeviceName;
} DEVICEUSRATTR;

typedef struct
{
      time_t      timeConnect;
      time_t      timeDisconnect;
} TRANSACTIONUSRATTR;

extern DEVICEUSRATTR* GetDeviceUsrAttr(long hDevice);
extern TRANSACTIONUSRATTR* GetTransactionUsrAttr(long hDevice, long idTransaction);
extern TRANSACTIONUSRATTR* SetTransactionUsrAttr(long hDevice, long idTransaction);

long EventHandler(unsigned long hEvent)
{
      long hDevice;
      long lEvent;
      CL_EVENTDATA* pclEventData;
      int iRet;
      long idTransaction;
      DEVICEUSRATTR* pDeviceUsrAttr;
      TRANSACTIONUSRATTR* pTransactionUsrAttr;

      hDevice = sr_getevtdev(hEvent);
      if (hDevice == -1)
      {
            printf("EventHandler - sr_getevtdev() failed\n");
            return 1;
      }

      lEvent = sr_getevttype(hEvent);
      if (lEvent == -1)
      {
            printf("EventHandler - sr_getevttype() failed\n");
            return 1;
      }

      pclEventData = (CL_EVENTDATA*)sr_getevtdatap(hEvent);
      if (pclEventData == NULL)
      {
            printf("EventHandler - sr_getevtdatap() failed\n");
            return 1;
      }

      pDeviceUsrAttr = GetDeviceUsrAttr(hDevice);

      if (lEvent == CLEV_MESSAGE)
      {
            printf("EventHandler - CLEV_MESSAGE - iResult=%08X\n", pclEventData-
>iResult);

            iRet = cl_GetTransaction(hDevice, &idTransaction, pclEventData);
            if (iRet != 0)
```

```
        {
              if (iRet == -2)
              {
                    printf("EventHandler - Transaction already released\n");
              }
              else
              {
                    printf("EventHandler - cl_GetTransaction() failed\n");
              }
              return 0;
        }

        printf("Transaction ID=%08X\n", idTransaction);

        if ((pclEventData->iResult & ECL_FIRST_MESSAGE) != 0)
        {
              pTransactionUsrAttr = SetTransactionUsrAttr(hDevice, idTransaction);
        }
        else
        {
              pTransactionUsrAttr = GetTransactionUsrAttr(hDevice, idTransaction);
        }

        if (pTransactionUsrAttr != NULL)
        {
              if ((pclEventData->iResult & ECL_CONNECT_MESSAGE) != 0)
              {
                    pTransactionUsrAttr->timeConnect = pclEventData->timeEvent;
              }

              if ((pclEventData->iResult & ECL_DISCONNECT_MESSAGE) != 0)
              {
                    pTransactionUsrAttr->timeDisconnect = pclEventData->timeEvent;
              }
        }

        if ((pclEventData->iResult & ECL_LAST_MESSAGE) != 0)
        {
              free(pTransactionUsrAttr);
              pTransactionUsrAttr = NULL;

              iRet = cl_ReleaseTransaction(hDevice, idTransaction);
              if (iRet != 0)
              {
                    if (iRet == -2)
                    {
                          printf("EventHandler - Transaction already released\n");
                    }
                    else
                    {
                          printf("EventHandler - cl_ReleaseTransaction() failed\n");
                     }
              }
        }

        return 0;
  }
  else if (lEvent == CLEV_ALARM)
  {
        printf("EventHandler - CLEV_ALARM - iResult=%08X\n", pclEventData->iResult);
        return 0;
  }
  else if (lEvent == CLEV_ERROR)
  {
        printf("EventHandler - CLEV_ERROR - iResult=%08X\n", pclEventData->iResult);
```

```
            return 0;
      }

      printf("EventHandler - Unknown event(%08X)\n", lEvent);
      return 1;
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	**Returned When**
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetTransactionDetails( )**
- **cl_ReleaseTransaction( )**
- **cl_SetTransactionUsrAttr( )**
- **cl_GetTransactionUsrAttr( )**

Name:	int cl_GetTransactionDetails(hDevice, pidTransaction, pclEventData, plReference, piSemanticsStateIndex, pszSemanticsStateName, iSemanticsStateNameSize)	
Inputs:	long hDevice	• call logging device handle
	long* pidTransaction	• pointer to returned transaction ID
	CL_EVENTDATA* pclEventData	• pointer to call logging event data block
	long* plReference	• pointer to returned call reference number
	int* piSemanticsStateIndex	• pointer to returned index of current semantics state
	char* pszSemanticsStateName	• pointer to buffer into which name of current semantics state is returned
	int iSemanticsStateNameSize	• size of buffer into which name of current semantics state is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetTransactionDetails( )** function returns the ID and details of a transaction for which a CLEV_MESSAGE event was generated. The details about the specified call logging transaction are optional and can include the call reference number and the name or index of the current semantics state. Pass NULL as the related parameter for any details that are not needed.

The list of semantics states (count, names and indexes) is protocol dependent.

Parameter	Description
hDevice	The device handle of the call logging device.
pidTransaction	The pointer to the returned call logging transaction ID. The call logging transaction ID is unique and protocol dependent.
pclEventData	The pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.
plReference	The pointer to the returned call reference number of the specified call logging transaction. The meaning of the call reference number is protocol specific.
piSemanticsStateIndex	The pointer to the returned index of the current semantics state of the specified call logging transaction. Semantics states are indexed from 0 to the number of semantics states minus one.
pszSemanticsStateName	The pointer to the buffer into which the name of the current semantics state of the specified call logging transaction is returned. The name is returned as an ASCIIZ string.
iSemanticsStateNameSize	The size of the buffer into which the name of the current semantics state of the specified call logging transaction is returned, where maximum size includes the terminating NUL of the ASCIIZ string.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can be called only while processing a CLEV_MESSAGE event.

■ Example

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

void GetTransactionDetails_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      long idTransaction;
      long lReference;
      int iSemanticsState;
      char szSemanticsStateName[32];

      iRet = cl_GetTransactionDetails(hDevice, &idTransaction, pclEventData,
&lReference, &iSemanticsState, szSemanticsStateName, sizeof(szSemanticsStateName));
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetTransactionDetails_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetTransactionDetails_WithinEventHandler -
cl_GetTransactionDetails() failed\n");
            }
            return;
      }

      printf("Transaction ID=%08X, Reference=%08X, State=\"%s\"(%i)\n", idTransaction,
lReference, szSemanticsStateName, iSemanticsState);
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetTransaction( )**
- **cl_ReleaseTransaction( )**
- **cl_SetTransactionUsrAttr( )**
- **cl_GetTransactionUsrAttr( )**
- **cl_GetSemanticsStateCount( )**
- **cl_GetSemanticsStateName( )**

Name:	int cl_GetTransactionUsrAttr(hDevice, idTransaction, ppUsrAttr)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	void** ppUsrAttr	• pointer to returned pointer to user-defined attribute
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetTransactionUsrAttr( )** function returns the user-defined transaction attribute for a specified call logging transaction. The user-defined attributes are set using the **cl_SetTransactionUsrAttr( )** function. If the **cl_SetTransactionUsrAttr( )** function has not been called for the specified call logging transaction, NULL will be returned.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
ppUsrAttr	The pointer to the returned pointer to the user-defined attribute for the specified call logging transaction.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The application is responsible for freeing the memory allocated to store the user-defined attribute.

■ Example

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

typedef struct
{
      time_t      timeConnect;
      time_t      timeDisconnect;
} TRANSACTIONUSRATTR;

TRANSACTIONUSRATTR* GetTransactionUsrAttr(long hDevice, long idTransaction)
{
      TRANSACTIONUSRATTR* pTransactionUsrAttr;
      int iRet;

      iRet = cl_GetTransactionUsrAttr(hDevice, idTransaction,
(void**)&pTransactionUsrAttr);
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetTransactionUsrAttr - Transaction already released\n");
            }
            else
            {
                  printf("GetTransactionUsrAttr - cl_GetTransactionUsrAttr() failed\n");
            }

            return NULL;
      }

      return pTransactionUsrAttr;
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetTransaction( )**
- **cl_GetTransactionDetails( )**
- **cl_ReleaseTransaction( )**
- **cl_SetTransactionUsrAttr( )**

Name:	int cl_GetUsrAttr(hDevice, ppUsrAttr)	
Inputs:	long hDevice	• call logging device handle
	void** ppUsrAttr	• pointer to returned pointer to user-defined attribute
Returns:	0 on success -1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetUsrAttr( )** function returns the user-defined attribute for a call logging device. The user-defined attributes are set using the **cl_SetUsrAttr( )** function. If NULL was specified as the **pUsrAttr** parameter of the **cl_Open( )** function and if the **cl_SetUsrAttr( )** function has not been called for the specified call logging device, NULL will be returned.

Parameter	Description
hDevice	The device handle of the call logging device.
ppUsrAttr	The pointer to the returned pointer to the user-defined attribute for the specified call logging device.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The application is responsible for freeing the memory allocated to store the user-defined attribute.

■ Example

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

typedef struct
```

```
{
      const char* pszProtocol;
      const char* pszNetworkDeviceName;
      const char* pszUserDeviceName;
} DEVICEUSRATTR;

DEVICEUSRATTR* GetDeviceUsrAttr(long hDevice)
{
      DEVICEUSRATTR* pDeviceUsrAttr;

      if (cl_GetUsrAttr(hDevice, (void**)&pDeviceUsrAttr) != 0)
      {
            printf("GetDeviceUsrAttr - cl_GetUsrAttr() failed\n");

            return NULL;
      }

      return pDeviceUsrAttr;
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_Open( )**
- **cl_SetUsrAttr( )**

Name:	int cl_GetVariable(hDevice, pclEventData, pszVariableName, pszVariable, iVariableSize)	
Inputs:	long hDevice	• call logging device handle
	CL_EVENTDATA* pclEventData	• pointer to call logging event data block
	const char* pszVariableName	• pointer to ASCIIZ string that specifies name of variable
	char* pszVariable	• pointer to buffer into which the value of the variable is returned
	int iVariableSize	• size of buffer into which the value of the variable is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_GetVariable( )** function returns the semantics-defined variable as it would have appeared at the time the CLEV_MESSAGE event was generated. The current list of semantics variables common to all protocols is (case-sensitive) CALLED, CALLING, CHANNEL. Additional protocol-specific variable names can be defined by the semantics.

Parameter	Description
hDevice	The device handle of the call logging device.
pclEventData	The pointer to the call logging event data block obtained from **sr_getevtdatap( )** while the function was processing a CLEV_MESSAGE event. See *section 1.7.2. Retrieving Event Data* for more information.
pszVariableName	The pointer to the ASCIIZ string that specifies the name of the semantics-defined variable.
pszVariable	The pointer to the buffer into which the value of the semantics-defined variable is returned. The value is returned as an ASCIIZ string.
iVariableSize	The size of the buffer into which the value of the semantics-defined variable is returned, where maximum size includes the terminating NUL of the returned ASCIIZ string.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- This function can only be called while processing a CLEV_MESSAGE event.

■ Example

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

void GetVariable_WithinEventHandler(long hDevice, CL_EVENTDATA* pclEventData)
{
      int iRet;
      char szCalled[32];
      char szCalling[32];
      char szChannel[8];

      iRet = cl_GetVariable(hDevice, pclEventData, "CALLED", szCalled,
sizeof(szCalled));
      if (iRet != 0)
      {
```

```
            if (iRet == -2)
            {
                  printf("GetVariable_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetVariable_WithinEventHandler - cl_GetVariable() failed\n");
            }
      }
      else
      {
            printf("Called party number is: \"%s\"\n", szCalled);
      }

      iRet = cl_GetVariable(hDevice, pclEventData, "CALLING", szCalling,
sizeof(szCalling));
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetVariable_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetVariable_WithinEventHandler - cl_GetVariable() failed\n");
            }
      }
      else
      {
            printf("Calling party number is: \"%s\"\n", szCalling);
      }

      iRet = cl_GetVariable(hDevice, pclEventData, "CHANNEL", szChannel,
sizeof(szChannel));
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("GetVariable_WithinEventHandler - Transaction already
released\n");
            }
            else
            {
                  printf("GetVariable_WithinEventHandler - cl_GetVariable() failed\n");
            }
      }
      else
      {
            printf("Bearer channel number is: \"%s\"\n", szChannel);
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDCONTEXT	invalid event context
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INVALIDPARAMETER	invalid parameter
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetCalled( )**
- **cl_GetCalling( )**
- **cl_GetChannel( )**
- **cl_PeekVariable( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_Open(phDevice, pszDeviceName, pUsrAttr)	
Inputs:	long* phDevice	• pointer to returned call logging device handle
	const char* pszDeviceName	• pointer to ASCIIZ string defining device to be opened and protocol to be used
	void* pUsrAttr	• pointer to user-defined attribute for device
Returns:	0 on success -1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_Open( )** function opens a call logging device and returns a call logging device handle that will be used to monitor the traffic on the line. On digital HiZ boards, the **pszDeviceName** parameter defines the protocol to be used, the method for retrieving L2 frames, and, if needed, the names of the HiZ devices to be used. On analog HiZ boards, this parameter defines the list of HiZ devices to be used.

Parameter	Description
phDevice	The pointer to the returned call logging device handle.
pszDeviceName	The pointer to the ASCIIZ string that defines the call logging device to be opened and the protocol to be used if applicable. See below for a description of the format for this parameter.
pUsrAttr	The pointer to the user-defined attribute for the specified call logging device.

The format of the **pszDeviceName** parameter is:

<field1><field2>…<fieldN>

These fields may be listed in any order. The format of each of these fields is:

:<key>_<value>

Table 15 lists the valid keys and their acceptable values for digital HiZ boards. All other keys are reserved for future use.

Table 15. pszDeviceName Field Values (Digital HiZ)

Key	Meaning	Acceptable Values (Digital HiZ)
M	Specifies the method used to get L2 frames.	HDLC FILE **
P	Specifies the protocol name.	ISDN * NET5 QSIGE1 4ESS 5ESS DMS NI2 NTT QSIGT1
N	Specifies the name of the HiZ device that is connected to the network side. This key is ignored when the method used to get L2 frames is :M_FILE.	dtiB<n>
U	Specifies the name of the HiZ device that is connected to the user side. This key is ignored when the method used to get L2 frames is :M_FILE	dtiB<n>

* The protocol name ISDN specifically refers to the E-1 Euro-ISDN protocol, also known as NET5.

** Use FILE with the **cl_DecodeTrace( )** for testing purposes. No physical device is needed when the FILE method is used to get L2 frames.

Table 16 lists the valid keys and their acceptable values for analog HiZ boards. All other keys are reserved for future use. In this table, TSC refers to Telephony Service Channel.

Table 16. pszDeviceName Field Values (Analog HiZ)

Key	Meaning	Acceptable Values (Analog HiZ)
M	Specifies the method for collecting signaling data.	TSC
P	(not supported)	(not supported)
N	Specifies the names of analog HiZ devices connected to monitored lines.	dtiB*x*T*y*, dtiB*x*T*y*-dtiB*x*T*z*, dtiB*x*, dtiB*x*-dtiB*w*, or any comma-separated list of such values (see the description following this table for more information)
U	(not supported)	(not supported)

On analog HiZ boards, the "**N**" field accepts a comma-separated list of analog HiZ device names, expressed as one of the following:

- **dtiB*x*T*y***: a single network interface time slot device (one channel)
- **dtiB*x*T*y*-dtiB*x*T*z***: a range of network interface time slot devices on the same network interface logical board (several channels)
- **dtiB*x***: a network interface logical board (all the channels on the logical board)
- **dtiB*x*-dtiB*w***: a range of network interface logical boards (all the channels on the logical boards).

NOTE: The device name is case-sensitive. You must use the format shown: lowercase "dti", uppercase "B", and uppercase "T".

On analog and digital HiZ boards, a call logging device gathers a set of HiZ line devices and monitors the whole set of channels associated with these devices.

■ Termination Events

None

Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The application is responsible for allocating the memory used to store the user-defined attribute.
- Once per process, the **cl_Open( )** function must be preceded by a call to **gc_Start( )**.

Example

```
#include <srllib.h>
#include <gclib.h>
#include <gcerr.h>
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>
#include <memory.h>


/* The Call Logging Device Handle */
long g_hDevice = EV_ANYDEV;

typedef struct
{
  const char* pszProtocol;
  const char* pszNetworkDeviceName;
  const char* pszUserDeviceName;
} DEVICEUSRATTR;

extern long EventHandler(unsigned long hEvent);
extern void ExitApplication(void);

int InitApplication(const char* pszProtocol, const char* pszNetworkDeviceName, const
char* pszUserDeviceName)
{
/*
  To build the pszDeviceName parameter of cl_Open(), parameters are considered as
follows:
  - pszProtocol==NULL and pszNetworkDeviceName==NULL
    => error, pszUserDeviceName ignored
  - pszProtocol==NULL and pszNetworkDeviceName!=NULL
    => Analog devices (:M_TSC), pszUserDeviceName ignored
  - pszProtocol!=NULL and (pszNetworkDeviceName==NULL or pszUserDeviceName==NULL)
    => Digital recorded frames (:M_FILE), pszNetworkDeviceName and pszUserDeviceName
ignored
  - pszProtocol!=NULL and pszNetworkDeviceName!=NULL and pszUserDeviceName!=NULL
    => Digital devices (:M_HDLC)
*/
  DEVICEUSRATTR* pDeviceUsrAttr;
  char szDeviceName[256];

  if (g_hDevice == EV_ANYDEV)
  {
    if ((pszProtocol == NULL) && (pszNetworkDeviceName == NULL))
    {
      printf("InitApplication - Invalid parameters\n");
```

```
      return -1;
    }

    pDeviceUsrAttr = (DEVICEUSRATTR*)malloc(sizeof(DEVICEUSRATTR));
    if (pDeviceUsrAttr == NULL)
    {
      printf("InitApplication - malloc() failed\n");
      return -1;
    }

    memset(pDeviceUsrAttr, 0, sizeof(DEVICEUSRATTR));
    pDeviceUsrAttr->pszProtocol = pszProtocol;
    pDeviceUsrAttr->pszNetworkDeviceName = pszNetworkDeviceName;
    pDeviceUsrAttr->pszUserDeviceName = pszUserDeviceName;

    if (gc_Start(NULL) != GC_SUCCESS)
    {
      printf("InitApplication - gc_Start() failed\n");

      free(pDeviceUsrAttr);
      return -1;
    }

    if (pszProtocol == NULL)
    {
      sprintf(szDeviceName, ":M_TSC:N_%s", pszNetworkDeviceName);
    }
    else if ((pszNetworkDeviceName == NULL) || (pszUserDeviceName == NULL))
    {
      sprintf(szDeviceName, ":P_%s:M_FILE", pszProtocol);
    }
    else
    {
      sprintf(szDeviceName, ":P_%s:M_HDLC:N_%s:U_%s", pszProtocol, pszNetworkDeviceName,
pszUserDeviceName);
    }

    if (cl_Open(&g_hDevice, szDeviceName, pDeviceUsrAttr) != 0)
    {
      printf("InitApplication - cl_Open() failed\n");

      gc_Stop();
      free(pDeviceUsrAttr);
      return -1;
    }

    if (sr_enbhdlr(g_hDevice, EV_ANYEVT, EventHandler) != 0)
    {
      printf("InitApplication - sr_enbhdlr() failed\n");

      cl_Close(g_hDevice);
      g_hDevice = EV_ANYDEV;
      gc_Stop();
      free(pDeviceUsrAttr);
      return -1;
    }
  }

  return 0;
}

int main(int argc, char* argv[])
{
  char szUnusedInput[256];
```

```
/* Typical call for Digital HiZ */
  /*
  if (InitApplication("ISDN", "dtiB1", "dtiB2") != 0)
  */
/* Typical call for Analog HiZ */
  */
  if (InitApplication(NULL, "dtiB1,dtiB2T1-dtiB2T2,dtiB3-dtiB4", NULL) != 0)
  */

  {
    return 1;
  }

  /* Wait until <Return> is hit */
  gets(szUnusedInput);

  ExitApplication();
  return 0;
}
```

■ Errors

The **cl_Open( )** function does not return errors in the standard Call Logging API return code format because the error is a system, parameter, or Global Call error. Also, the standard **ATDV_LASTERR( )** function requires a device handle and if **cl_Open( )** fails, there is none.

If an error occurs during the **cl_Open( )** call, a -1 is returned and the specific error code is returned in the errno global variable. The error codes that can be returned if **cl_Open( )** fails are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_GCOPENEX_NETWORK *	**gc_OpenEx( )** failed on the network side
ECL_GCOPENEX_USER *	**gc_OpenEx( )** failed on the user side
ECL_DTOPEN_BOARD	**dt_open( )** failed on board
ECL_ATDVSUBDEVS_BOARD	**ATDV_SUBDEVS( )** failed on board
ECL_HIZOPEN_CHANNEL	**hiz_open( )** failed on channel
ECL_INVALIDPARAMETER	invalid parameter
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging API error; cause

	unknown
*The following additional flag is set in these error code values to indicate that the error occurred while the Call Logging API was calling a Global Call function:	
ECL_FLAG_INSIDE_GC	use **gc_ErrorValue( )** for an additional error description

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_Close( )**
- **cl_SetUsrAttr( )**

Name:	int cl_PeekCalled(hDevice, idTransaction, pszCalled, iCalledSize)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	char* pszCalled	• pointer to buffer into which called party number is returned
	int iCalledSize	• size of buffer into which called party number is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_PeekCalled( )** function gets the called party number as it was observed at the time the function was called. The **cl_PeekCalled( )** function returns the value of the semantics-defined CALLED variable.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
pszCalled	The pointer to the buffer into which the called party number is returned. The called party number is returned as an ASCIIZ string. If the called party number is not available, the function returns with an empty string.
iCalledSize	The size of the buffer into which the called party number is returned, where maximum size includes the terminating NUL of the returned ASCIIZ string.

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void PeekCalled(long idTransaction)
{
      int iRet;
      char szCalled[32];

      if (g_hDevice != EV_ANYDEV)
      {
            iRet = cl_PeekCalled(g_hDevice, idTransaction, szCalled, sizeof(szCalled));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("PeekCalled - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekCalled - cl_PeekCalled() failed\n");
                  }
                  return;
            }

            printf("Called party number is: \"%s\"\n", szCalled);
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRANSACTIONRELEASED	transaction already released

ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetCalled( )**
- **cl_PeekVariable( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_PeekCalling(hDevice, idTransaction, pszCalling, iCallingSize)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	char* pszCalling	• pointer to buffer into which calling party number is returned
	int iCallingSize	• size of buffer into which calling party number is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_PeekCalling( )** function gets the calling party number as it was observed at the time the function was called. The **cl_PeekCalling( )** function returns the value of the semantics-defined CALLING variable.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
pszCalling	The pointer to the buffer into which the calling party number is returned. The calling party number is returned as an ASCIIZ string. If the calling party number is not available, the function returns with an empty string.
iCallingSize	The size of the buffer into which the calling party number is returned, where maximum size includes the terminating NUL of the returned ASCIIZ string.

■ Termination Events

None

Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void PeekCalling(long idTransaction)
{
      int iRet;
      char szCalling[32];

      if (g_hDevice != EV_ANYDEV)
      {
            iRet = cl_PeekCalling(g_hDevice, idTransaction, szCalling,
sizeof(szCalling));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("PeekCalling - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekCalling - cl_PeekCalling() failed\n");
                  }
                  return;
            }

            printf("Calling party number is: \"%s\"\n", szCalling);
      }
}
```

Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRANSACTIONRELEASED	transaction already released

ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetCalling( )**
- **cl_PeekVariable( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_PeekChannel(hDevice, idTransaction, pszChannel, iChannelSize)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	char* pszChannel	• pointer to buffer into which channel number is returned
	int iChannelSize	• size of buffer into which channel number is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_PeekChannel( )** function gets the channel number as it was observed at the time the function was called. The **cl_PeekChannel( )** function returns the value of the semantics-defined CHANNEL variable.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
pszChannel	The pointer to the buffer into which the channel number is returned. The channel number is returned as an ASCIIZ string. For analog HiZ boards, this string contains a number between 1 and the number of analog HiZ devices specified in the **pszDeviceName** parameter of the **cl_Open( )** function.
iChannelSize	The size of the buffer into which the channel number is returned, where maximum size includes the terminating NUL of the returned ASCIIZ string.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The range of channel numbers depends on the semantics rules and does not necessarily match Intel device name numbering. For example, E-1 ISDN channel numbers range from 1 to 15 and from 17 to 31, while the Intel device names on an E-1 board range from “dtiBxT1” to “dtiBxT30”.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void PeekChannel(long idTransaction)
{
      int iRet;
      char szChannel[8];

      if (g_hDevice != EV_ANYDEV)
      {
            iRet = cl_PeekChannel(g_hDevice, idTransaction, szChannel,
sizeof(szChannel));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("PeekChannel - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekChannel - cl_PeekChannel() failed\n");
                  }
                  return;
            }

            printf("Bearer channel number is: \"%s\"\n", szChannel);
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetChannel( )**
- **cl_GetOrdinalChannel( )**
- **cl_PeekOrdinalChannel( )**
- **cl_PeekVariable( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_PeekOrdinalChannel(hDevice,idTransaction, piOrdinalChannel)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	int* piOrdinalChannel	• pointer to the returned ordinal number of the channel
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_PeekOrdinalChannel( )** function gets the ordinal channel number as it was observed at the time the function was called. The value returned is the ordinal number of the channel among those monitored by the call logging device. The ordinal channel number is an integer between 0 and the number of monitored channels minus one.

If you previously called **cl_PeekChannel( )** to access your own channel-related data stored in an array, then the new **cl_PeekOrdinalChannel( )** function is a recommended replacement. As it returns an integer between 0 and the number of monitored channels minus one, this function is similar to **cl_PeekChannel( )** and should be used in its place. This new function, introduced in System Release 6.0, allows direct access into any array of channel-related structures that you wish to allocate. This is particularly useful in E-1 trunk environments where the value returned by **cl_PeekChannel( )** ranges from 1 to 15 and 17 to 31. The new **cl_PeekOrdinalChannel( )** function allows you to build a unified method to handle monitored channels, whether they sit on T-1, E-1, or analog lines.

Parameter	Description
hDevice	The device handle of the call logging device
idTransaction	The call logging transaction ID
piOrdinalChannel	The pointer to the ordinal number of the channel, returned as an integer between 0 and the number of monitored channels minus one

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void PeekOrdinalChannel(long idTransaction)
{
  int iRet;
  int iOrdinalChannel;

  if (g_hDevice != EV_ANYDEV)
  {
    iRet = cl_PeekOrdinalChannel(g_hDevice, idTransaction, &iOrdinalChannel);
    if (iRet != 0)
    {
      if (iRet == -2)
      {
        printf("PeekOrdinalChannel - Transaction already released\n");
      }
      else
      {
        printf("PeekOrdinalChannel - cl_PeekOrdinalChannel() failed\n");
      }
      return;
    }

    printf("Channel index (or ordinal number) is: %i\n", iOrdinalChannel);
  }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetOrdinalChannel( )**
- **cl_PeekChannel( )**

Name:	int cl_PeekVariable(hDevice, idTransaction, pszVariableName, pszVariable, iVariableSize)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
	const char* pszVariableName	• pointer to ASCIIZ string that specifies name of variable
	char* pszVariable	• pointer to buffer into which the value of the variable is returned
	int iVariableSize	• size of buffer into which the value of the variable is returned
Returns:	0 on success -1 on failure -2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_PeekVariable( )** function gets the value of a semantics-defined variable as it was observed at the time the function was called. The current list of semantics variables common to all protocols is (case-sensitive) CALLED, CALLING, CHANNEL Additional protocol-specific variable names can be defined by the semantics.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
pszVariableName	The pointer to the ASCIIZ string that specifies the name of the semantics-defined variable.
pszVariable	The pointer to the buffer into which the value of the semantics-defined variable is returned. The variable is returned as an ASCIIZ string.
iVariableSize	The size of the buffer into which the value of the semantics-defined variable is returned, where maximum size includes the terminating NUL of the ASCIIZ string.

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>

/* The Call Logging Device Handle */
extern long g_hDevice;

void PeekVariable(long idTransaction)
{
      int iRet;
      char szCalled[32];
      char szCalling[32];
      char szChannel[8];

      if (g_hDevice != EV_ANYDEV)
      {
            iRet = cl_PeekVariable(g_hDevice, idTransaction, "CALLED", szCalled,
sizeof(szCalled));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
```

```
                        printf("PeekVariable - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekVariable - cl_PeekVariable() failed\n");
                  }
            }
            else
            {
                  printf("Called party number is: \"%s\"\n", szCalled);
            }

            iRet = cl_PeekVariable(g_hDevice, idTransaction, "CALLING", szCalling,
sizeof(szCalling));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("PeekVariable - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekVariable - cl_PeekVariable() failed\n");
                  }
            }
            else
            {
                  printf("Calling party number is: \"%s\"\n", szCalling);
            }

            iRet = cl_PeekVariable(g_hDevice, idTransaction, "CHANNEL", szChannel,
sizeof(szChannel));
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("PeekVariable - Transaction already released\n");
                  }
                  else
                  {
                        printf("PeekVariable - cl_PeekVariable() failed\n");
                  }
            }
            else
            {
                  printf("Bearer channel number is: \"%s\"\n", szChannel);
            }
      }
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_INVALIDPARAMETER	invalid parameter
ECL_TRANSACTIONRELEASED	transaction already released
ECL_NOMEM	out of memory
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetVariable( )**
- **cl_PeekCalled( )**
- **cl_PeekCalling( )**
- **cl_PeekChannel( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_ReleaseTransaction(hDevice, idTransaction)	
Inputs:	long hDevice	• call logging device handle
	long idTransaction	• call logging transaction ID
Returns:	0 on success	
	-1 on failure	
	-2 if call logging transaction already released	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_ReleaseTransaction( )** function releases a call logging transaction. Once a transaction has been released, information about that particular call logging transaction can no longer be queried. Because a call logging system has no impact on a digital line and only observes the activity on the line, the **cl_ReleaseTransaction( )** function does not drop the monitored call. Rather, the purpose of the function is to instruct the Call Logging API to release the internal resources allocated for the specified call logging transaction.

The **cl_ReleaseTransaction( )** function is usually called from the call logging event handler when a CLEV_MESSAGE event is received and the event has the ECL_LAST_MESSAGE bit set in the iResult field of its call logging event data block (see section *1.7.2. Retrieving Event Data* for more information). If the **cl_ReleaseTransaction( )** function is called before this specific event is received, the application will not receive any additional call logging events related to the specified call logging transaction.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID to be released.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

- When the application has completed a call logging transaction, the **cl_ReleaseTransaction( )** function must be called to release internally allocated resources. Failure to do so may cause memory problems due to the allocated memory not being released (ECL_OUT_OF_MEMORY error).
- Once the **cl_ReleaseTransaction( )** function is called, the call logging transaction ID is no longer valid.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>

typedef struct
{
      const char* pszProtocol;
      const char* pszNetworkDeviceName;
      const char* pszUserDeviceName;
} DEVICEUSRATTR;

typedef struct
{
      time_t      timeConnect;
      time_t      timeDisconnect;
} TRANSACTIONUSRATTR;

extern DEVICEUSRATTR* GetDeviceUsrAttr(long hDevice);
extern TRANSACTIONUSRATTR* GetTransactionUsrAttr(long hDevice, long idTransaction);
extern TRANSACTIONUSRATTR* SetTransactionUsrAttr(long hDevice, long idTransaction);

long EventHandler(unsigned long hEvent)
{
      long hDevice;
      long lEvent;
      CL_EVENTDATA* pclEventData;
      int iRet;
      long idTransaction;
      DEVICEUSRATTR* pDeviceUsrAttr;
      TRANSACTIONUSRATTR* pTransactionUsrAttr;

      hDevice = sr_getevtdev(hEvent);
      if (hDevice == -1)
      {
            printf("EventHandler - sr_getevtdev() failed\n");
            return 1;
      }

      lEvent = sr_getevttype(hEvent);
      if (lEvent == -1)
      {
            printf("EventHandler - sr_getevttype() failed\n");
            return 1;
      }

      pclEventData = (CL_EVENTDATA*)sr_getevtdatap(hEvent);
      if (pclEventData == NULL)
      {
```

```
            printf("EventHandler - sr_getevtdatap() failed\n");
            return 1;
      }

      pDeviceUsrAttr = GetDeviceUsrAttr(hDevice);

      if (lEvent == CLEV_MESSAGE)
      {
            printf("EventHandler - CLEV_MESSAGE - iResult=%08X\n", pclEventData-
>iResult);

            iRet = cl_GetTransaction(hDevice, &idTransaction, pclEventData);
            if (iRet != 0)
            {
                  if (iRet == -2)
                  {
                        printf("EventHandler - Transaction already released\n");
                  }
                  else
                  {
                        printf("EventHandler - cl_GetTransaction() failed\n");
                  }
                  return 0;
            }

            printf("Transaction ID=%08X\n", idTransaction);

            if ((pclEventData->iResult & ECL_FIRST_MESSAGE) != 0)
            {
                  pTransactionUsrAttr = SetTransactionUsrAttr(hDevice, idTransaction);
            }
            else
            {
                  pTransactionUsrAttr = GetTransactionUsrAttr(hDevice, idTransaction);
            }

            if (pTransactionUsrAttr != NULL)
            {
                  if ((pclEventData->iResult & ECL_CONNECT_MESSAGE) != 0)
                  {
                        pTransactionUsrAttr->timeConnect = pclEventData->timeEvent;
                  }

                  if ((pclEventData->iResult & ECL_DISCONNECT_MESSAGE) != 0)
                  {
                        pTransactionUsrAttr->timeDisconnect = pclEventData->timeEvent;
                  }
            }

            if ((pclEventData->iResult & ECL_LAST_MESSAGE) != 0)
            {
                  free(pTransactionUsrAttr);
                  pTransactionUsrAttr = NULL;

                  iRet = cl_ReleaseTransaction(hDevice, idTransaction);
                  if (iRet != 0)
                  {
                        if (iRet == -2)
                        {
                              printf("EventHandler - Transaction already released\n");
                        }
                        else
                        {
                              printf("EventHandler - cl_ReleaseTransaction() failed\n");
                        }
```

```
                }
            }

            return 0;
        }
        else if (lEvent == CLEV_ALARM)
        {
            printf("EventHandler - CLEV_ALARM - iResult=%08X\n", pclEventData->iResult);
            return 0;
        }
        else if (lEvent == CLEV_ERROR)
        {
            printf("EventHandler - CLEV_ERROR - iResult=%08X\n", pclEventData->iResult);
            return 0;
        }

        printf("EventHandler - Unknown event(%08X)\n", lEvent);
        return 1;
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_GetTransaction( )**
- **cl_GetTransactionDetails( )**
- **cl_GetTransactionUsrAttr( )**
- **cl_SetTransactionUsrAttr( )**

Name:	int cl_SetTransactionUsrAttr(hDevice, idTransaction, pUsrAttr)
Inputs:	long hDevice • call logging device handle long idTransaction • call logging transaction ID void* pUsrAttr • pointer to user-defined attribute
Returns:	0 on success -1 on failure -2 if call logging transaction already released
Includes:	cllib.h
Mode:	synchronous
Platform:	DM3

Description

The **cl_SetTransactionUsrAttr( )** function sets the user-defined transaction attribute for a specified call logging transaction.

Parameter	Description
hDevice	The device handle of the call logging device.
idTransaction	The call logging transaction ID.
pUsrAttr	The pointer to the user-defined attribute for the specified call logging transaction.

Termination Events

None

Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The application is responsible for allocating the memory used to store the user-defined attribute.

Example

```
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>
#include <memory.h>
```

```
typedef struct
{
      time_t      timeConnect;
      time_t      timeDisconnect;
} TRANSACTIONUSRATTR;

TRANSACTIONUSRATTR* SetTransactionUsrAttr(long hDevice, long idTransaction)
{
      TRANSACTIONUSRATTR* pTransactionUsrAttr;
      int iRet;

      pTransactionUsrAttr = (TRANSACTIONUSRATTR*)malloc(sizeof(TRANSACTIONUSRATTR));
      if (pTransactionUsrAttr == NULL)
      {
            printf("SetTransactionUsrAttr - malloc() failed\n");
            return NULL;
      }

      memset(pTransactionUsrAttr, 0, sizeof(TRANSACTIONUSRATTR));

      iRet = cl_SetTransactionUsrAttr(hDevice, idTransaction, pTransactionUsrAttr);
      if (iRet != 0)
      {
            if (iRet == -2)
            {
                  printf("SetTransactionUsrAttr - Transaction already released\n");
            }
            else
            {
                  printf("SetTransactionUsrAttr - cl_SetTransactionUsrAttr() failed\n");
            }

            free(pTransactionUsrAttr);
            return NULL;
      }

      return pTransactionUsrAttr;
}
```

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_TRANSACTIONRELEASED	transaction already released
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_GetTransaction( )**
- **cl_GetTransactionDetails( )**
- **cl_GetTransactionUsrAttr( )**
- **cl_ReleaseTransaction( )**

Name:	int cl_SetUsrAttr(hDevice, pUsrAttr)	
Inputs:	long hDevice	• call logging device handle
	void* pUsrAttr	• pointer to user-defined attribute
Returns:	0 on success	
	-1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_SetUsrAttr( )** function sets the user-defined attribute for a call logging device. When possible, the user-defined attribute for a call logging device should be specified by the **pUsrAttr** parameter of the **cl_Open( )** function.

Parameter	Description
hDevice	The device handle of the call logging device.
pUsrAttr	The pointer to the user-defined attribute for the specified call logging device.

■ Termination Events

None

■ Cautions

- The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.
- The application is responsible for allocating the memory used to store the user-defined attribute.

■ Example

```
#include <srllib.h>
#include <cllib.h>
#include <stdio.h>
#include <malloc.h>
#include <memory.h>

/* The Call Logging Device Handle */
extern long g_hDevice;
```

```
typedef struct
{
      const char* pszProtocol;
      const char* pszNetworkDeviceName;
      const char* pszUserDeviceName;
} DEVICEUSRATTR;

extern DEVICEUSRATTR* GetDeviceUsrAttr(long hDevice);

void SetDeviceUsrAttr(const char* pszProtocol, const char* pszNetworkDeviceName, const
char* pszUserDeviceName)
{
      DEVICEUSRATTR* pNewDeviceUsrAttr;
      DEVICEUSRATTR* pOldDeviceUsrAttr;

      if (g_hDevice != EV_ANYDEV)
      {
            pNewDeviceUsrAttr = (DEVICEUSRATTR*)malloc(sizeof(DEVICEUSRATTR));
            if (pNewDeviceUsrAttr == NULL)
            {
                  printf("SetDeviceUsrAttr - malloc() failed\n");
                  return;
            }

            memset(pNewDeviceUsrAttr, 0, sizeof(DEVICEUSRATTR));
            pNewDeviceUsrAttr->pszProtocol = pszProtocol;
            pNewDeviceUsrAttr->pszNetworkDeviceName = pszNetworkDeviceName;
            pNewDeviceUsrAttr->pszUserDeviceName = pszUserDeviceName;

            pOldDeviceUsrAttr = GetDeviceUsrAttr(g_hDevice);

            if (cl_SetUsrAttr(g_hDevice, pNewDeviceUsrAttr) != 0)
            {
                  printf("SetDeviceUsrAttr - cl_SetUsrAttr() failed\n");

                  free(pNewDeviceUsrAttr);
                  return;
            }

            free(pOldDeviceUsrAttr);
      }
}
```

Errors

None.

See Also

- **cl_GetUsrAttr( )**
- **cl_Open**

Name:	int cl_StartTrace(hDevice, pszFileName)	
Inputs:	long hDevice	• call logging device handle
	const char* pszFileName	• pointer to ASCIIZ string
Returns:	0 on success	
	-1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_StartTrace( )** function is not supported on analog HiZ boards.

The **cl_StartTrace( )** function starts recording an L2 frames trace file. This function is used for testing and debugging. The SnifferMFC demo shows how this function is used.

Parameter	Description
hDevice	the device handle of the call logging device
pszFileName	a pointer to the ASCIIZ string that specifies the path and name of the L2 frames trace file

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

See the SnifferMFC demo for an example of how to use this function.

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_NULLPARAMETER	invalid NULL parameter
ECL_TRACESTARTED	trace already started
ECL_FILEOPEN	fopen failed
ECL_FILEWRITE	fwrite failed
ECL_UNSUPPORTED	function not supported
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

■ See Also

- **cl_StopTrace( )**
- **cl_DecodeTrace( )**

Name:	int cl_StopTrace(hDevice)	
Inputs:	long hDevice	• call logging device handle
Returns:	0 on success	
	-1 on failure	
Includes:	cllib.h	
Mode:	synchronous	
Platform:	DM3	

■ Description

The **cl_StopTrace( )** function is not supported on analog HiZ boards.

The **cl_StopTrace( )** function stops recording an L2 frames trace file. This function is used for testing and debugging. The SnifferMFC demo shows how this function is used.

Parameter	Description
hDevice	the device handle of the call logging device

■ Termination Events

None

■ Cautions

The Call Logging API is not multithread safe. Call logging functions must be called within the same thread.

■ Example

See the SnifferMFC demo for an example of how to use this function.

■ Errors

If the function returns a value < 0, use the SRL Standard Attribute function ATDV_LASTERR( ) to obtain the error code or use ATDV_ERRMSGP( ) to obtain a

descriptive error message. The error codes that can be returned by ATDV_LASTERR( ) are:

Error Code Value	Returned When
ECL_TRACENOTSTARTED	trace not started
ECL_FILECLOSE	fclose failed
ECL_UNSUPPORTED	function not supported
ECL_INTERNAL	internal Call Logging error; cause unknown

See section *3.2. Error Handling* for more information about the kinds of errors that can cause these codes to be returned.

See Also

- **cl_StartTrace( )**
- **cl_DecodeTrace( )**

Appendix A – Call Logging Sample Code

The following code provides an example of the function calls and procedures used to implement a network monitoring application.

NOTE: This sample code does not include error checking or testing on return values. It is only meant to demonstrate how the Call Logging functions are used.

```
// Application entry point
main()
{
      // Start Global Call
      gc_Start (NULL) ;

      // Open the call logging device
      long hDevice;
      cl_Open(&hDevice, ":P_ISDN:M_HDLC:N_dtiB1:U_dtiB2", NULL);

      // Enable the application event handler
      sr_enbhdlr(hDevice, EV_ANYEVT, appEventHandler);

      // Wait until <return> is hit
      char szUnusedInput[256];
      gets(szUnusedInput);

      // Disable the application event handler
      7dishdlr(hDevice, EV_ANYEVT, appEventHandler);

      // Close the call logging device
      cl_Close(hDevice);

      // Stop Global Call
      gc_Stop( ) ;
}


// Define the transaction bag structure that will be used as the transaction user
attribute
struct TRANSACTIONBAG
{
      char szCaller[256];
      char szCallee[256];
      char szChannel[256];
      time_t timeStart;
      time_t timeEnd;
};


// Application event handler
long appEventHandler(unsigned long hEvent)
{
```

```
      long hDevice = sr_getevtdev(hEvent);

      // Identify the type of event received
      switch (sr_getevttype(hEvent))
      {
      case CLEV_MESSAGE:

            // Get the call logging transaction ID
            CL_EVENTDATA* pclEventData = (CL_EVENTDATA*)sr_getevtdatap(hEvent);
            long idTransaction;
            cl_GetTransaction(hDevice, &idTransaction, pclEventData);

            // Is it the first message for this call logging transaction ?
            int iResult = pclEventData->iResult;
            if ((iResult & ECL_FIRST_MESSAGE) != 0)
            {
                  // Create the transaction bag structure
                  TRANSACTIONBAG* pTransactionBag = new TRANSACTIONBAG;
                  memset(pTransactionBag, 0, sizeof(TRANSACTIONBAG));

                  // Associate the transaction bag structure with the call logging
                     transaction
                  cl_SetTransactionUsrAttr(hDevice, idTransaction, pTransactionBag);

                  // Remember the time when the call logging transaction started
                  pTransactionBag->timeStart = pclEventData->timeEvent;
            }

            // Is it the last message for this call logging transaction ?
            if ((iResult & ECL_LAST_MESSAGE) != 0)
            {
                  // Retrieve the transaction bag structure for this call logging
                     transaction
                  TRANSACTIONBAG* pTransactionBag;
                  cl_GetTransactionUsrAttr(hDevice, idTransaction,
                  (void**)&pTransactionBag);

                  // Remember the time when the call transaction ended
                  pTransactionBag->timeEnd = pclEventData->timeEvent;

                  // Get the caller number
                  cl_GetCalling(hDevice, pclEventData, pTransactionBag->szCaller, 256);
                  // same as: cl_GetVariable(hDevice, pclEventData, "CALLING",
                     pTransactionBag->szCaller, 256);

                  // Get the callee number
                  cl_GetCalled(hDevice, pclEventData, pTransactionBag->szCallee, 256);
                  // same as: cl_GetVariable(hDevice, pclEventData, "CALLED",
                     pTransactionBag->szCallee, 256);

                  // Get the channel on which the call logging transaction took place
                  cl_GetChannel(hDevice, pclEventData, pTransactionBag->szChannel, 256);
                  // same as: cl_GetVariable(hDevice, pclEventData, "CHANNEL",
                     pTransactionBag->szChannel, 256);

                  // Application-specific: record the call logging transaction
statistics
                  RecordNetworkStatistics(pTransactionBag);

                  // Delete the transaction bag structure
                  delete pTransactionBag;
```

```
                // We are now done with this call logging transaction
                cl_ReleaseTransaction(hDevice, idTransaction);
            }

            // This event has been consumed
            return 0;
            break;
        }

        return 1;
}
```

Glossary

ASCII American Standard Code for Information Interchange

asynchronous function A function that returns immediately to the application and returns a completion/termination event at some future time. An asynchronous function allows the current thread to continue processing while the function is running.

asynchronous mode The classification for functions that operate without blocking other functions.

B channel A bearer channel used in ISDN interfaces. This circuit-switched, digital channel can carry voice or data at 64,000 bits/sec in either direction.

data structure Programming term for a data element consisting of fields, where each field may have a different definition and length. A group of data structure elements usually share a common purpose or functionality.

device handle A numerical reference to a device, obtained when the device is opened. This handle is used for all operations on that device.

driver A software module that provides a defined interface between the program and the hardware.

event An unsolicited communication from a hardware device to an operating system, application, or driver. Events are generally attention-getting messages, allowing a process to know when a task is complete or when an external event occurs.

Global Call A unified, high-level API that shields developers from the low-level signaling protocol details that differ in countries around the world. Allows the same application to easily work on multiple signaling systems worldwide (for example, ISDN, T-1 robbed bit, R2MF, pulsed, MF Socotel, Analog).

Integrated Services Digital Network (ISDN) An internationally accepted standard for voice, data, and signaling that provides users with integrated services using digital encoding at the user-network interface.

ISDN see *Integrated Services Digital Network*

PSTN see *Public Switched Telephone Network*

Public Switched Telephone Network (PSTN) Refers to the worldwide telephone network accessible to all those with either a telephone or access privileges.

semantics rules The guidelines used by the Call Logging API to analyze signaling data and manage call logging transactions. Semantics rules include semantics states, semantics variables, and a list of specific call logging events. Semantics rules are defined by and dependent on the analog or digital nature of the line, and/or on the CCS protocol being used.

semantics states The types of call states, as determined by the protocol, used to identify the current status of a monitored call. Examples of semantics states include dialing, alerting, connected and disconnected.

semantics variables The kinds of information to be monitored and collected for call logging transactions. Variables are protocol-dependent and may include the calling party number, called party number and bearer channel number.

SRL Standard Runtime Library

Standard Runtime Library A software resource containing Event Management and Standard Attribute functions and data structures used by all Intel telecom devices, but which return data unique to the device.

synchronous function Synchronous functions block an application or process until the required task is successfully completed or a failed/error message is returned.

termination condition An event that causes a process to stop.

termination event An event that is generated when an asynchronous function terminates.

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. In a multithreaded process, more than one thread is executed seemingly simultaneously. When the last thread finishes its task, 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.

time slot: 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.

unsolicited events An event that occurs without prompting, for example, CLEV_MESSAGE, CLEV_ERROR, CLEV_ALARM.

Index

V