ISDN Software Reference

for Linux and Windows

05-0867-005

COPYRIGHT NOTICE

Publication Date: November, 2001

Part Number: 05-0867-005

Dialogic, an Intel Company
1515 Route 10
Parsippany NJ 07054
U.S.A.

For **Technical Support**, visit the Dialogic support website at:
http://support.dialogic.com

For **Sales Offices** and other contact information, visit the main Dialogic website at:
http://www.dialogic.com

Table of Contents

List of Tables

List of Figures

1. How To Use This Guide

This *ISDN Software Reference* is intended for users who have purchased a Dialogic board that provides ISDN Primary Rate or Basic Rate connectivity and a Dialogic system software and SDK release for either LINUX or Windows.

Installation instructions for the ISDN Package software are provided in the release notes or for system software and SDK releases, in the Getting Started booklet in the CD-ROM jewel case. Refer to the appropriate hardware Quick Install card to install the Dialogic ISDN boards and any other Dialogic boards in the system.

1.1. Products Covered by this Guide

For information on the boards that are supported with the ISDN Package software being used, see the Release Notes for the Package or the Release Catalog for the System Software and SDK release that includes the ISDN Package.

1.2. Organization of this Guide

The *ISDN Software Reference* includes an overview of the Dialogic ISDN digital telephony interface and a complete reference to the Dialogic LINUX and Windows C library functions for ISDN Primary Rate Interface (PRI) and Basic Rate Interface (BRI) support. It is organized as follows:

Chapter 2 provides a brief overview of the Basic Rate Interface (BRI) and the Primary Rate Interface (PRI), including possible applications, and lists the supported protocols and Dialogic boards that can be used for BRI and PRI.

Chapter 3 presents an overview of ISDN technology, including signaling, framing, and ISDN call control states.

Chapter 4 presents an overview of the Dialogic ISDN library functions, including function category classifications.

Chapter 5 provides a detailed alphabetical reference to the Dialogic ISDN library functions.

Chapter 6 provides descriptions of the data structures used by various ISDN library functions.

Chapter 7 describes the events and errors returned by Dialogic ISDN library functions.

Chapter 8 provides guidelines for developing ISDN applications.

Appendix A contains ISDN call control scenarios, such as call establishment and termination, hold and retrieve, and network facility requests, and includes descriptions of the Two B Channel Transfer (TBCT) and Non-call Associated Signaling (NCAS) features.

Appendix B contains call control scenarios for the DPNSS protocol.

Appendix C contains information elements and ISDN message types for the DPNSS protocol.

Appendix D contains guidelines for establishing ISDN connections and instructions for building a cable that will connect the board to a network termination unit (NTU).

Appendix E lists related publications. This includes a list of Dialogic documentation.

In addition, this document provides a **Glossary** of related terms and an **Index**.

2. Introduction to Dialogic ISDN Products

The Integrated Services Digital Network (ISDN) is a collection of internationally accepted standards for defining interfaces and operation of digital switching equipment for the transmission of voice, data, and signaling. ISDN has the following characteristics and advantages:

- ISDN makes all transmission circuits end-to-end digital.
- ISDN adopts a standard out-of-band signaling system.
- ISDN brings significantly more bandwidth to the desktop.

The Dialogic ISDN products provide telecommunication applications access to the many features and benefits of ISDN. The Dialogic ISDN product line consists of the Dialogic ISDN firmware and the Dialogic ISDN software library. The Dialogic ISDN firmware and software work with the supported Dialogic boards to provide a network interface using the following ISDN technologies:

- **Basic Rate Interface (BRI)**, which allows the transfer of both voice and data over standard 64 Kbps lines. A BRI line consists of two 64 Kbps channels for a total of 128 Kbps.
- **Primary Rate Interface (PRI)**, which allows the transfer of voice and data over T-1 (1.544 Mbps) or E-1 (2.048 Mbps) lines.

This chapter describes the features and benefits of BRI and PRI and lists typical applications for each. This chapter also provides a list of currently supported BRI and PRI protocols and a list of Dialogic boards that can be used for BRI and PRI.

2.1. The Basic Rate Interface

There are two types of Dialogic BRI boards, BRI/SC and BRI/2:

- The **BRI/SC** boards allow individual routing of up to 32 B channels (voice/data channels) and 16 D channels (signaling channels) to any of the application-selectable SCbus time slots using the SCbus distributed switching capability. B channel traffic may be routed from the ISDN network or local

station set device to and from the SCbus. BRI/SC boards can be used in either a Windows or a LINUX operating environment.

The Dialogic BRI/SC protocol implementations comply with the North American standard ISDN BRI, Euro-ISDN protocol for BRI, and the INS64 standard used in Japan. See *Section 2.3. Dialogic ISDN Protocol Support* for a listing of currently supported BRI protocols.

- The **BRI/2** boards emulate two standard BRI station sets with display, and are designed to support the Euro-ISDN protocol. The BRI/2 boards provide analog voice processing, via the Dialogic Voice Functions (see Note 1 below) and the Dialogic ISDN API, and support many enhanced ISDN features. In addition, BRI/2 boards can facilitate four instances of Dialogic DSP-based Group 3 Fax (also referred to as DSP Fax, see Note 2 below) and provide ISDN B channel data communications. BRI/2 boards are currently supported only under the Windows operating system.

NOTES: **1.** For information on using Voice Functions, see the *Voice Software Reference – Features Guide* for the appropriate operating system.

2. For information on using DSP Fax with the BRI/2, see the *Fax Software Reference* the appropriate operating system.

The Dialogic BRI/SC and BRI/2 boards provide network access via the ISDN Basic Rate Interface (BRI). The BRI/SC boards can also function as a digital station interface, enabling direct access to BRI station sets (telephones) from PC-based computer telephony (CT) systems, and eliminating the need for local switch integration.

The BRI/SC boards may also be used for connecting voice processing applications to PBX or Public Switched Telephone Network (PSTN) BRI access lines.

2.1.1. Features of BRI

BRI offers advantages or access to features not available on PRI. For example, many ISDN PBX Primary Rate products are designed as terminal equipment (TE) for connection to the central office, and cannot provide network-side access to other terminal equipment. The BRI/SC or BRI/2 board can be used to connect to a PBX.

Both the BRI/SC and the BRI/2 boards provide access to ISDN Layer 3 Supplemental Services. These services can be divided into two categories:

- **Hold and Retrieve** allows the application to place calls on hold, to retrieve held calls and to respond to requests to hold or retrieve held calls using the following Dialogic functions: **cc_HoldCall( )**, **cc_RetrieveCall( )**, **cc_HoldAck( )**, **cc_HoldRej( )**, **cc_RetrieveAck( )**, and **cc_RetrieveRej( )**. Refer to the function descriptions in *Chapter 5. ISDN Function Reference* and to *Appendix D* for more information.
- **Messaging** allows the application to access other supplemental services, such as Called/Calling Party Identification, Message Waiting and Call Transfer. The services are invoked by formatting information elements (IEs) and sending them as non-call related Facility Messages (SndMsg_Facility) to the PBX or network. See the **cc_SndMsg( )**, **cc_SndNonCallMsg( )**, and **cc_SetInfoElem( )** functions for information on sending Facility Messages. See the **cc_GetCallInfo( )** and **cc_GetNonCallMsg( )** functions for information on retrieving Facility Messages. Also refer to *Appendix D* for more on BRI Supplemental Services.

In addition to the features described above, BRI/2 boards provide the following fax and data communications features:

- **Fax features** - BRI/2 boards support Dialogic DSP-based Group 3 Fax. Key features of DSP Fax include:
  - Four channels of voice and fax per board
  - Maximum of 16 fax channels per system (4 BRI/2 boards in one system)
  - Software-based fax modem
  - Compatibility with ITU-T Group 3 (T.4, T.30), ETSI NET/30

  NOTE: For more information on using DSP Fax with the BRI/2, see the *Fax Software Reference for Windows*.
- **Data features** - BRI/2 boards provide link layer access, across the B channel, which allows for reliable transfer of data across an ISDN network. The BRI/2 boards offer Network Device Interface Specification (NDIS) compatibility. NDIS is a Microsoft® standard that allows for multiple network adapters and multiple protocols to coexist. NDIS permits the high-level protocol components to be independent of the BRI/2 by providing a standard interface.

This means that the BRI/2 may be used by applications that use the standard networking APIs that are part of the Windows operating system. NDIS supports Remote Access Service (RAS) and Point-to-Point Protocol (PPP):

- **Remote Access Service (RAS)** - RAS is enabled via NDIS and allows users to interact with the service selections provided by the specified dial-up networking setup.
- **Point-to-Point Protocol (PPP)** - PPP is a method of exchanging data packets between two computers. PPP can carry different network layer protocols over the same link. When the PPP connection sequence is successfully completed, the remote client and RAS server can begin to transfer data using any supported protocol. PPP Multilink provides the ability to aggregate two or more physical connections to form one larger logical connection, improving bandwidth and throughput for remote connections.

The BRI/SC boards provide a different set of ISDN features. Advantages and features specific to BRI/SC boards include the following:

- **Data Link Layer Access -** the BRI/SC products have data link layer access (also known as LAPD Layer 2). This feature provides for the reliable transfer of data across the physical link (physically connected devices), and sends blocks of frames with the necessary synchronization, error control, and flow control. Layer 2 access is particularly useful when using a Dialogic ISDN board to connect to a switch using a Layer 3 protocol that is not provided in the firmware.
- **Point-to-Multipoint Configuration** - this feature allows BRI/SC protocols to support multiple TEs to be connected to a line that is configured to be a network. Up to eight TEs may be connected with a maximum of two active, non-held calls at a time. An unlimited number of calls may exist in a held state, but these calls cannot be retrieved if both B channels are already in use by other calls.
- **Tone Generation** - this feature allows BRI/SC protocols, under a network configuration, to generate and play tones on any B channel with the use of the on-board DSP chip. These tones can be requested and configured by the application or they can be generated by the firmware. For more information, see the **cc_ToneRedefine( )**, **cc_PlayTone( )**, and **cc_StopTone( )** function descriptions in *Chapter 5. ISDN Function Reference*.

- **Multiple D Channel Configuration** - this feature allows the D channel of each line to be configured at any time, and as many times as needed. The application can configure and reconfigure the protocol for each station interface, allowing different protocols to be run on different stations simultaneously. The application can also change between User side and Network side, assign and change the Service Profile Identifier (SPID), and change other attributes such as the generation of in-band tones. See the **cc_SetDChanCfg( )** function description in *Chapter 5. ISDN Function Reference* for more information.
- **5ESS Custom Messaging** - the 5ESS protocol has a custom messaging feature, which allows the application to send requests to drop calls and to redirect the state of calls. See the **cc_SndMsg( )** function description in *Chapter 5. ISDN Function Reference* for more information.

2.1.2. Typical BRI Applications

ISDN BRI technology offers call handling features, such as Automatic Call Distribution (ACD), call waiting, call monitoring, and caller ID, that can be used to develop BRI applications such as the following:

- Call center and business communication platforms
- Automated call rerouting applications such as debit card services, international callback, and long distance resale
- Wireless gateway access
- Voice processing system access for the station side of ISDN PBXs
- Protocol conversion equipment, which allows the application to convert calls from one network protocol to another network protocol, without resource boards

2.2. The Primary Rate Interface

The Dialogic Primary Rate Interface (PRI) product line enables SCbus applications to use the speed, power, and flexibility of ISDN. The Dialogic PRI firmware supports both T-1 and E-1 protocols.

The T-1 protocol implementations comply with the North American standard ISDN Primary Rate and the INS-1500 standard used in Japan. In North America and Japan, the Primary Rate includes 23 voice/data channels (B channels) and one signaling channel (D channel).

The E-1 protocol implementations comply with the E-1 Primary Rate interface protocol. The E-1 ISDN Primary Rate includes 30 voice/data channels (B channels) and two additional channels: one signaling channel (D channel) and one framing channel to handle synchronization.

See *Section 2.3. Dialogic ISDN Protocol Support* for specific supported T-1 and E-1 protocols.

2.2.1. Benefits of PRI

ISDN Primary Rate technology offers the benefits inherent in digital connectivity, such as fast call connection (setup and teardown), fast Dialed Number Identification Service (DNIS), and fast Automatic Number Identification (ANI) acquisition. These features support applications such as speed dialing, automated operator services, call waiting, call forwarding, and geographic analysis of customer databases.

ISDN PRI applications also can take advantage of the following features offered by the network:

- **Two B-Channel Transfer (TBCT).** Enables a user to request the switch to connect together two independent calls on the user's interface. The user who made the request is released from the calls and the other two users are directly connected. This feature is supported for 5ESS, 4ESS, and National ISDN-2 (NI2) protocols. For more on TBCT, see *Appendix A - Call Control Scenarios*.
- **Non-Call Associated Signaling (NCAS).** Allows users to communicate by means of user-to-user signaling without setting up a circuit-switched connection (it does not occupy B channel bandwidth). A temporary signaling connection is established and cleared in a manner similar to the control of a circuit-switched connection. This feature is supported for the 5ESS protocol. For more on NCAS, see *Appendix A - Call Control Scenarios*.

- **Vari-A-Bill.** A flexible billing option enabling a customer to modify the charge for a call while the call is in a stable state (for example, between answer and disconnect). This feature is available from the AT&T network only.
- **ANI-on-demand.** Allows the user to request a caller ID number to identify the origin of the call, when necessary.
- **Non-facility Associated Signaling (NFAS).** Provides support for multiple ISDN spans from a single D channel. The NFAS D channel is supported only on the DTI/240SC and DTI/300SC products.
- **User-to-user information.** An information element that may be included in setup, connect, or disconnect messages.
- **Call-by-Call service selection.** This feature allows the user to access different services, such as an 800 line or a WATS line, on a per call basis.
- **LAP-D Layer 2 (Data Link Layer) access.** This feature allows access to the data link layer, providing for the reliable transfer of data across the physical link (physically connected devices), and sending blocks of frames with the necessary synchronization, error control, and flow control. Layer 2 access is particularly useful when using a Dialogic ISDN board to connect to a switch using a Layer 3 protocol that is not provided in the firmware.

The following sections describe the PRI configurations required to support various ISDN applications.

2.2.2. PRI Configurations and Applications

Using the Dialogic ISDN Primary Rate product, software applications operating in the host PC can control Primary Rate line connectivity. The boards can be configured as terminating devices or installed in a variety of drop-and-insert configurations.

In a terminating configuration, incoming or outgoing calls on ISDN lines are processed by supported resource boards (such as voice boards). In a drop-and-insert configuration, incoming and outgoing calls (on individual channels) can either be processed by supported resource boards or passed on to additional network boards. Calls can also be both processed by supported resource boards and passed on to additional network boards, as well.

The following sections contain examples of typical applications for terminate and drop-and-insert configurations.

Terminate Configuration Applications

Dialogic ISDN products in a terminate configuration with one or more resource boards (for example, voice boards) allow for the development of a variety of applications, such as the following:

- **Audiotex** applications allow users to retrieve and listen to pre-recorded messages over the telephone. In some simpler applications, the user's only action is to initiate the call. More complex applications may require the user to respond to one or more prompts using, for example, the touch tone keys on the telephone.
- **Telemarketing** applications use voice processing technology to facilitate a high volume of inbound/outbound calls. The calls depend upon an operator-assisted transaction and/or are motivated by a specific event, such as the need to quickly distribute a promotional or informational message to a targeted audience of phone users.
- **Host Interactive Voice Response (HIVR)** applications enable fully automated transaction processing or transaction passing to occur over the telephone. Examples of these applications include, but are not limited to, banking by phone, order entry systems, and inventory control services.
- **Central Office (CO) Voice Mail** applications include many kinds of solutions which enable users to store and forward or record and retrieve voice messages. These applications may be simple and operate like an answering machine, or they may be more complex and include automated attendant functions.

Drop-and-Insert Configuration Applications

Dialogic ISDN products can be placed in a variety of drop-and-insert configurations, providing all the features and benefits of terminate configurations, plus the ability to access an operator or another call. Drop-and-insert configurations allow calls to be passed from one network module to another network module. The following types of applications are provided by drop-and-insert configurations:

- **Operator Services** applications are able to automate large numbers of telephone calls that require some kind of caller assistance. Such applications include, but are not limited to, partially automated directory assistance, calling card voice prompting, and collect calls.
- **Telemarketing** applications depend on a specific event/transaction. The primary difference between telemarketing applications in a terminate configuration and telemarketing applications in a drop-and-insert configuration is that the latter allow for the use of an existing PBX and telephone equipment.
- **Protocol Conversion** is a drop-and-insert configuration, without resource boards, used to convert calls from one network protocol to another network protocol.

2.3. Dialogic ISDN Protocol Support

A protocol is a set of rules or standards that defines the format and timing of data transmission between two devices. Like any evolving technology, a single standard ISDN implementation has yet to be agreed upon worldwide. Standards have been established in a number of countries or regions. The following table lists the protocols that Dialogic currently supports.

Table 1. ISDN Protocols

PRI T-1	PRI E-1	BRI/SC	BRI/2
4ESS 5ESS DMS/250 DMS/100 National ISDN-2 NTT (INS1500)	1TR6 DASS2 DPNSS Euro-ISDN (CTR4) Q.SIG TPH VN3	DMS-100 Custom ETS300 (Euro-ISDN) (CTR3) INS64 Lucent 5ESS Custom National ISDN-1 (NI1)	ETS300 (Euro-ISDN) (CTR3) Multi-link PPP (Channel bundling) PPP/MP

NOTES: **1.** Only one protocol may be configured per digital network interface at a given time.

2. Refer to the package release notes or system software catalog for updated protocol support information. Also, refer to the Dialogic **Worldview** website at http://www.dialogic.com and to the Dialogic **FirstCall™ Info Server** website at http://support.dialogic.com.

DPNSS and Q.SIG are ISDN PRI E-1 protocols that are used to pass calls transparently between PBXs. Unlike other PRI E-1 protocols, DPNSS and Q.SIG require special ISDN API library functions for processing calls placed on hold. Other ISDN library functions have modifications that are specific to the DPNSS protocol. For a listing of the functions that are specific to DPNSS and Q.SIG, see *Chapter 4. ISDN Function Overview*. For more detailed information on the DPNSS and Q.SIG ISDN library functions, see *Chapter 5. ISDN Function Reference*.

NOTE: To support DPNSS, Dialogic E-1 boards, such as the D/300SC-E1 or the DTI/601SC, must be software-enabled. The enablement utility can be downloaded from the Dialogic FirstCall™ InfoServer website. However, an order form must also be filled out to request an access code to use the DPNSS enablement utility. For more information on how this works, see the Dialogic FirstCall™ InfoServer website.

3. ISDN Technology Overview

The Integrated Services Digital Network (ISDN) is a digital communications network capable of carrying all forms (voice, computer, and facsimile) of digitized data between switched endpoints. This chapter provides an overview of ISDN technology, including the signaling method used to transmit data, framing and framing formats, and ISDN call control states.

3.1. Signaling

The digital data stream contains two kinds of information: user data and signaling data. The signaling data is used to control the transmission of user data. For example, in telephony applications user data is usually digitally encoded voice data; in file transfer applications, user data is packets of High-level Data Link Control (HDLC) encoded information. Signaling data carries information such as the current state of the channel (for example, whether the telephone is on-hook or off-hook). Signaling data can also indicate who is calling, the type of call (voice or data), and the number that was dialed.

ISDN protocols use an out-of-band signaling method, which means the signaling data is carried on a channel or channels separate from user data channels. The user data is transmitted in eight bit samples on B or "bearer" channels, and the signaling data is transmitted outside of the eight bit sample on D channels.

In addition to carrying signaling data out-of-band, ISDN technology uses common channel signaling (CCS). This means that one signaling channel (D channel) carries signaling data for more than one user data channel (B channel). In BRI, two channels of user data for every one channel of signaling data are transmitted. In PRI T-1 lines, 23 channels of user data (B channels) and one channel of signaling data (D channels) are transmitted. In PRI E-1 lines, 30 B channels and 1 D channel are transmitted. Common channel signaling also allows the transmission of additional information, such as ANI and DNIS digits, over the signaling channel.

3.2. Framing

Voice data from each time slot in a configuration is routed to a separate B channel. The signaling information for all B channels is routed to the D channel of the device. Information from the B and D channels is transmitted in frames. A single frame contains information from each of the channels, providing a "snapshot" of the data being transmitted at any given time.

The frames contain the eight bits of information about each time slot or channel. Each frame includes a series of bits used for synchronization, error checking, and diagnostics. A frame can be in one of several formats. In all formats, there is a flag bit between each frame to show where the frames begin and end. Different frame formats are supported in different networks to provide a variety of added features or benefits. The following frame formats are supported by the Dialogic ISDN software:

- ESF frame (Extended Superframe)
- D4 frame (Superframe)
- CEPT multiframe (with or without CRC4)

3.2.1. Data Link Layer (Layer 2) Frames

The frames that are transmitted over the Data Link Layer (Layer 2) contain information that controls the setup, maintenance and disconnection between the two physically connected devices (see Figure 1).

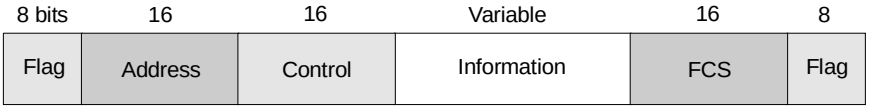

Figure 1. Layer 2 Frame (D Channel)

3.2.2. Network Layer (Layer 3) Frames

The Data Link Layer prepares the way for the transmission of Network Layer (Layer 3) frames of data (see Figure 2).

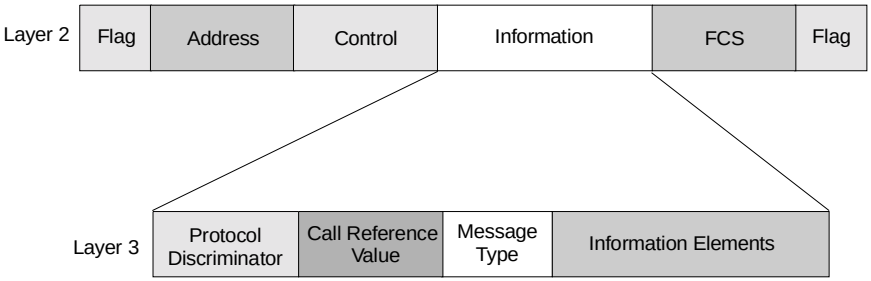

Figure 2. Layer 3 Frame (D Channel)

In general, the message format for Layer 3 frames comprises variable length fields with the following format:

- Protocol Discriminator – identifies the protocol type used to handle Layer 3 messages
- Call Reference Value (CRV) – a valued assigned to a call, by the network, for the duration of the call
- Message type – the set of messages used for establishing, controlling and tearing down a call
- Information elements (IE) – used with the message to provide additional information on the type and requirements of the call

Refer to the *Digital Network Interface Software Reference* for more on framing.

3.3. ISDN Call Control States

Each ISDN call that is received or generated by an application is processed through a series of call control states. Each state represents the completion of certain tasks and/or the current status of the call. The following table describes the ISDN call control states, based on standard Q.931 (Layer 3).

Table 2. Call Control States

Call State	Description
Accepted	An indication to the network that the incoming call has been accepted, but has not been connected to the end user. (Most voice applications do not need this.)
Alerting	The destination is reached and the phone is ringing. This state may be reported to the application or masked depending on the application directive.
Connected	An incoming or outgoing call is established. Typically, billing begins at this point and the B channel is cut through.
Dialing	Address and call setup information has been sent to, and acknowledged by, the network. Call establishment is in progress.
Disconnected	The network terminates the call and the application should drop the call.
Idle	A call is dropped and waiting for the application to release the call reference number (CRN).
Null	No call is assigned to the device (time slot or line).
Offered	An incoming call is offered by the network.

The call states change according to the sequence of functions called by the application and the events that originate in the network and system hardware. The current state of a call can be changed by:

- function call returns
- termination events (indicate completion of a function)
- unsolicited events

The way calls transition between states differs depending on whether the application uses asynchronous or synchronous programming:

- In general, in asynchronous mode, **events** trigger the transitions between call states. For example, the termination event, CCEV_ANSWERED, causes the call state to change to the Connected state. Likewise, the unsolicited event, CCEV_DISCONNECTED, causes the call state to change to the Disconnected state.
- **Synchronous** functions return at the successful completion of the function or if the function fails. The function waits for a completion or failure message from the firmware before it terminates, and the call transitions to another call state. For example, the **cc_MakeCall( )** and **cc_ReleaseCall( )** functions cause the call state to change upon their successful return. (Note that synchronous functions return information, **not** events.)

The following sections describe the state transitions for both asynchronous and synchronous call processes. (Refer to the *Voice Software Reference - Programmer's Guide* for the appropriate operating system for details on the supported programming models.)

3.3.1. Asynchronous Call Establishment

Figure 3 illustrates the call states associated with establishing or setting up a call in the asynchronous mode. The call establishment process for outbound calls is shown on the right side of the diagram; the inbound call set up process is shown on the left. All calls start from a Null state. See *Table 2* for descriptions of the call states.

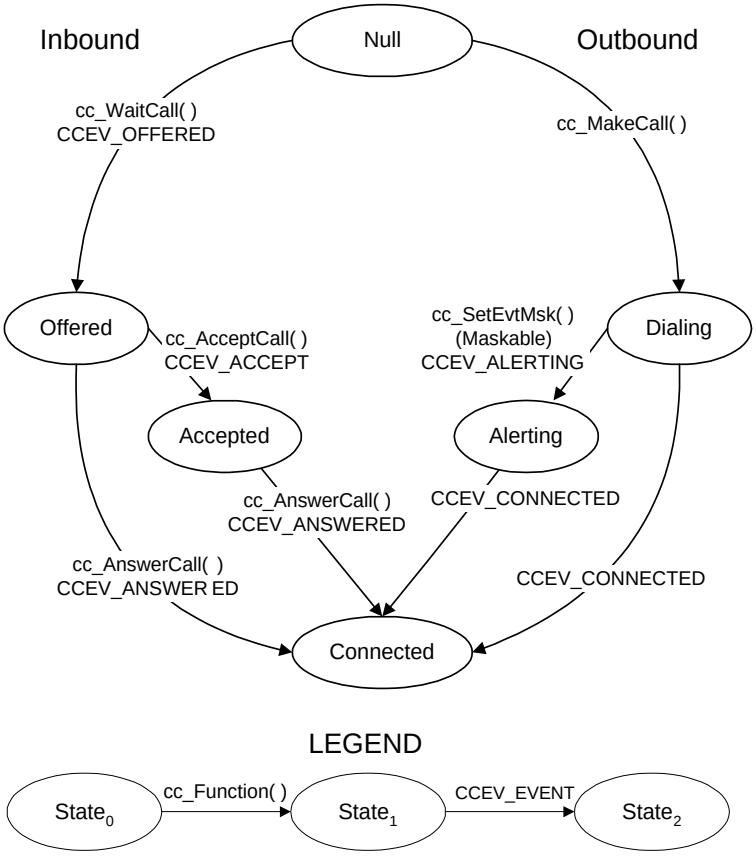

Figure 3. Asynchronous Call Establishment Process

Table 3 provides an example of a simple inbound call using the asynchronous call establishment process. The items denoted by an asterisk (*) are optional functions/events or maskable events that may be reported to the application. For more detailed call scenarios see *Appendix A*.

Table 3. Inbound Call Set-Up (Asynchronous Example)

Function/Event	**Action/Description**
cc_WaitCall( )	Issued once after line device opened with **cc_Open( )**.
CCEV_OFFERED	Indicates arrival of inbound call and initiates transition to Offered state
***cc_GetANI( )**	Request caller ID information
***cc_GetDNIS( )**	Retrieves DNIS digits received from the network
***cc_CallAck( )**	Requests additional call setup information
***cc_AcceptCall( )**	Issued to acknowledge that call was received but called party has not answered.
*CCEV_ACCEPT	Termination event - indicates call received, but not yet answered; causes transition to Accepted state.
cc_AnswerCall( )	Issued to connect call to called party (answer inbound call).
CCEV_ANSWERED	Termination event - inbound call connected; causes transition to Connected state.
* = Optional functions and events or maskable events	

Table 4 illustrates a simple scenario for making an outbound call using the asynchronous call establishment process. The items denoted by an asterisk (*) are optional functions/events or maskable events that may be reported to the application. For more detailed call scenarios, see *Appendix A*.

Table 4. Outbound Call Set-up (Asynchronous Example)

Function/Event	Action/Description
cc_MakeCall( )	Requests a connection using a specified line device. A CRN is assigned and returned immediately; call is transitioned to the Dialing state. CCEV_CONNECTED event sent if call is connected; otherwise a CCEV_TASKFAIL event is sent.
***cc_SetEvtMsk( )**	Specifies the events enabled or disabled for a specified line device.
*CCEV_ALERTING	Remote end was reached but a connection has not been established. When the call is answered, a CCEV_CONNECTED event is sent.
CCEV_CONNECTED	Indicates successful completion of **cc_MakeCall( )**. Call is in the Connected state.
* = Optional functions and events or maskable events	

3.3.2. Synchronous Call Establishment

Figure 4 illustrates the call states associated with establishing or setting up a call in the synchronous mode. The call establishment process for outbound calls is shown on the right side of the diagram; the inbound call set up process is shown on the left. All calls start from a Null state. See *Table 2* for descriptions of the call states.

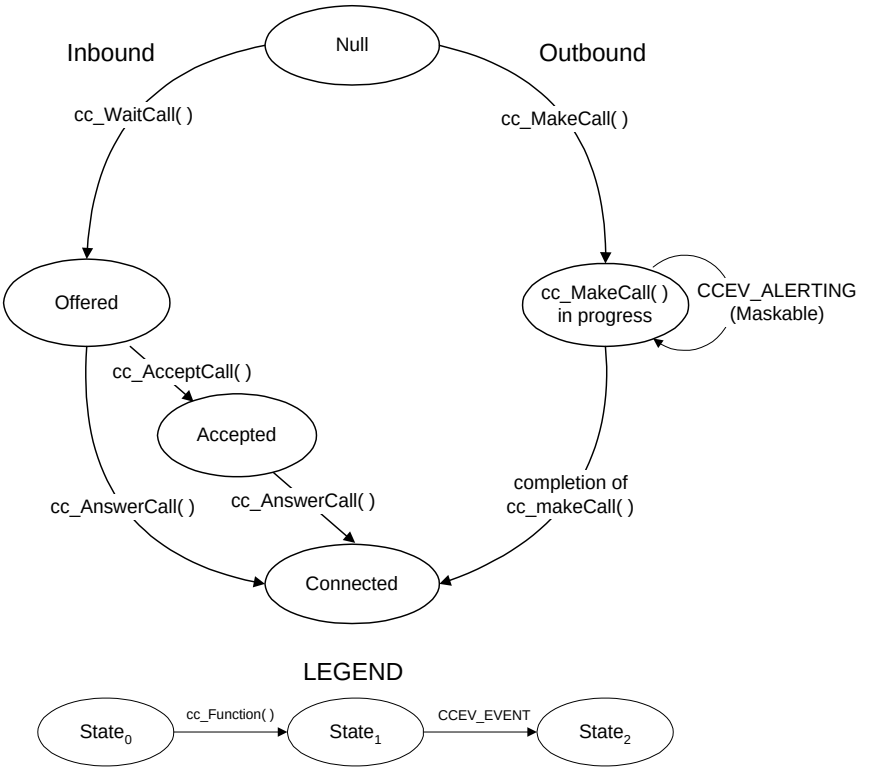

Figure 4. Synchronous Call Establishment Process

Table 5 provides an example of a simple inbound call using the synchronous call establishment process. The items denoted by an asterisk (*) are optional functions/events or maskable events that may be reported to the application. For more detailed call scenarios, see *Appendix A*.

Table 5. Inbound Call Set-Up (Synchronous Example)

Function	Action/Description
cc_WaitCall( )	Enables notification of an incoming call after line device opened with **cc_Open( )**. Call is in OFFERED state.
***cc_GetANI( )**	Request ANI information
***cc_GetDNIS( )**	Retrieves DNIS digits received from the network.
***cc_CallAck( )**	Requests additional call setup information
***cc_AcceptCall( )**	Issued to acknowledge that call was received but called party has not answered. Call is in Accepted state.
cc_AnswerCall( )	Issued to connect call to called party (answer inbound call). At the successful completion of **cc_AnswerCall( )**, the call is in Connected state.
NOTES: 1. * = Optional functions **2.** There are no termination events in synchronous mode.	

Table 6 illustrates a simple scenario for making an outbound call using the synchronous call establishment process. The item denoted by an asterisk (*) is a maskable event that may be reported to the application. For more detailed call scenarios, see *Appendix A*.

Table 6. Outbound Call Set-up (Synchronous Example)

Function/Event	Action/Description
cc_MakeCall( )	Requests a connection using a specified line device; a CRN is assigned and returned immediately.
*CCEV_ALERTING	Remote end was reached but a connection has not been established
none	When the **cc_MakeCall( )** function successfully completes, the call is in the Connected state.
NOTES: 1. * = Maskable events **2.** There are no termination events in synchronous mode.	

3.3.3. Asynchronous Call Termination

Figure 5 illustrates the call states associated with call termination or call teardown in the asynchronous mode initiated by either call disconnection or failure. A call can be terminated by the application or by the detection of call disconnect from the network. Either of these terminations can occur at any point in the process of setting up a call and during any call state. See *Table 2* for descriptions of the call states.

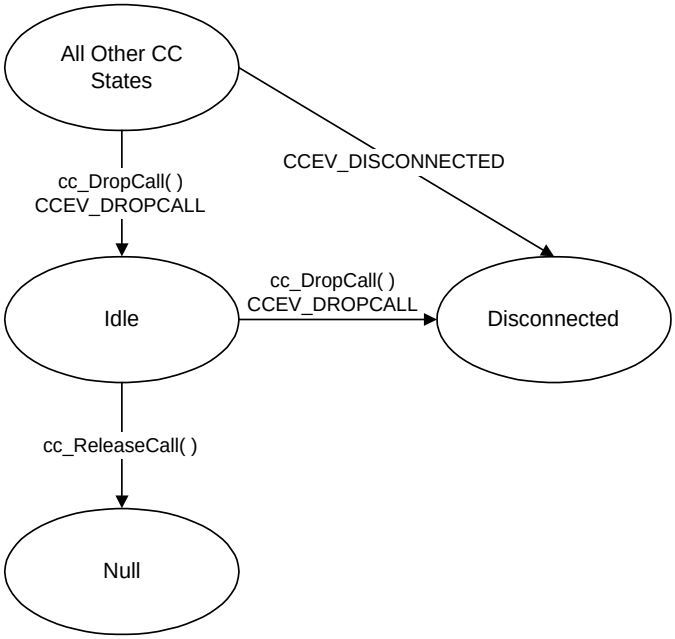

Figure 5. Asynchronous Call Disconnect or Failure Process

Table 7 presents an asynchronous call termination scenario. The item denoted by an asterisk (*) is an optional function call. For more detailed call scenarios, see *Appendix A*.

Table 7. Call Termination (Asynchronous Example)

Function/Event	Action/Description
CCEV_DISCONNECTED	Unsolicited event generated when call is terminated by network; initiates transition to Disconnected state.
cc_DropCall( )	Disconnects call specified by CRN. CCEV_DROPCALL event indicates completion of function
CCEV_DROPCALL	Termination event - call disconnected and changes to Idle state
***cc_GetBilling( )**	Retrieves billing information
cc_ReleaseCall ( ) **cc_ReleaseCallEx( )**	Issued to release all resources used for call; network port is ready to receive next call. Causes transition to Null state.
* = Optional functions	

3.3.4. Synchronous Call Termination

Figure 6 illustrates the call states associated with call termination or call teardown in the synchronous mode initiated by either call disconnection or failure. A call can be terminated by the application or by the detection of call disconnect from the network. Either of these terminations can occur at any point in the process of setting up a call and during any call state. See *Table 2* for descriptions of the call states.

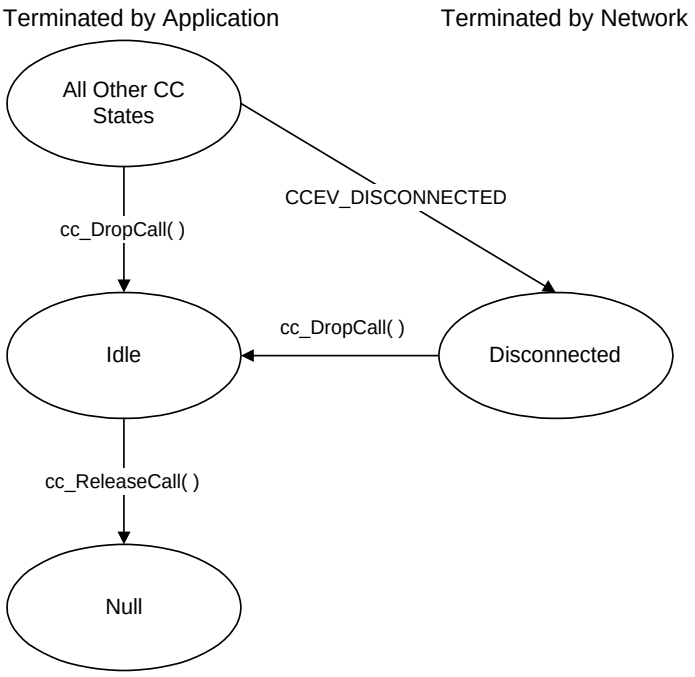

Figure 6. Synchronous Call Disconnect or Failure Process

Table 8 presents a synchronous call termination scenario. The item denoted by an asterisk (*) is an optional function call. For more detailed call scenarios, see *Appendix A*.

Table 8. Call Termination (Synchronous Example)

Function/Event	Action/Description
CCEV_DISCONNECTED	Unsolicited event generated when call is terminated by network; initiates transition to Disconnected state.
cc_DropCall( )	Disconnects call specified by CRN. Initiates transition to Idle state.
***cc_GetBilling( )**	Retrieves billing information
cc_ReleaseCall( )	Issued to release all resources used for call; network port is ready to receive next call. Causes transition to Null state.
NOTES: 1. * = Optional function **2.** There are no termination events in synchronous mode.	

4. ISDN Function Overview

This chapter provides the following information about the Dialogic ISDN library functions used to interact with the network in an ISDN environment:

- ISDN function categories
- a brief description of each ISDN library function
- the ISDN technologies supported for each function

For a complete description of each function, see *Chapter 5. ISDN Function Reference* in this guide.

4.1. ISDN Library Function Categories

The ISDN library functions can be divided into the following categories:

- **Call Control** - Perform basic call control actions, such as making, receiving, answering and dropping calls (see Table 9).
- **Optional Call Handling** - Perform additional call control actions, such as accepting calls, sending messages to the network, and setting and retrieving call-related information (see Table 10).
- **System Control** - Start and stop the system, that is, open and close devices and reset channels (see Table 11).
- **System Tools** - Perform system level tasks (see Table 12). These functions are divided into the following categories:
  - Configuration Tools - set and retrieve channel parameters and user attributes, and retrieve call states, call reference numbers, call reference values, etc.
  - Error Handling - retrieve error information (cause values and result values)
  - Tracing Functions - capture and store D channel information
  - Eventing Functions - set and retrieve event masks
- **Data Link Layer Handling** - Send and receive frames, that is, handle the transfer of frames between the application and the data link layer (see Table 14).
- **Hold and Retrieve** - Process calls on hold. These functions are used in BRI protocols and in the PRI DPNSS and Q.SIG protocols to place calls on hold,

retrieve calls from the Hold state, and to accept and reject hold requests and retrieve-from-hold requests (see Table 14).

- **Global Tone Generation** - Set, change, and control the In-band tones for BRI/SC protocols (see Table 15).

Table 9. Call Control Functions

Function	Description
cc_AnswerCall( )	accepts a connection request from the remote end
cc_DropCall( )	allows the application to disconnect a call
cc_MakeCall( )	requests connection on the specified line device
cc_ReleaseCall( )	releases the call reference number (CRN)
cc_ReleaseCallEx( )	releases all resources for the specified call (CRN)
cc_WaitCall( )	sets up the condition for processing an incoming call

Table 10. Optional Call Handling Functions

Function	Description
cc_AcceptCall( )	responds to an incoming call request
cc_CallAck( )	sends the first response to an incoming call
cc_CallProgress( )	sends a PROGRESS message to the network
cc_GetANI( )	retrieves Automatic Number Identification (ANI) information (caller ID)
cc_GetBilling( )	retrieves the charge information
cc_GetCallInfo( )	retrieves the information elements associated with the CRN
cc_GetChanId( )	retrieves the last channel information received from messages for a specified CRN
cc_GetDNIS( )	retrieves the dialed number information string (destination address)
cc_GetInfoElem( )	retrieves the information elements associated with a line device
cc_GetMoreDigits( )	collects more digits via overlap receiving
cc_GetNonCallMsg( )	retrieves the information associated with a GLOBAL or NULL CRN event
cc_GetSigInfo( )	retrieves the signaling information of an incoming message
cc_GetVer( )	retrieves the library version number
cc_ReqANI( )	requests the ANI (caller ID) from the network in ANI-on-demand environments
cc_SetBilling( )	sets the billing rate
cc_SetCallingNum( )	sets the default calling number

cc_SetInfoElem( )	sets an information element (IE)
cc_SetMinDigits( )	sets the minimum number of digits to be collected
cc_SndMsg( )	sends a non-call state associated message to the network
cc_SndNonCallMsg( )	sends a non-Call State related ISDN message to the network, with a GLOBAL or NULL CRN

Table 11. System Control Functions

Function	Description
cc_Open( )	opens a device
cc_Close( )	closes a previously opened device
cc_Restart( )	resets the channel to the Null state

Table 12. System Tool Functions

Function	Description
Configuration Tools:	
cc_CallState( )	retrieves the state of the call
cc_CRN2LineDev( )	returns the line device number associated with a specified call reference number
cc_GetCES( )	retrieves the connection endpoint suffix
cc_GetCRN( )	retrieves the call reference number
cc_GetDChanState( )	retrieves the status of the D Channel
cc_GetDLinkCfg( )	retrieves the configuration of a logical link
cc_GetDLinkState( )	retrieves the logical data link state
cc_GetNetCRV( )	retrieves the network call reference value for a specified call reference number
cc_GetParm( )	returns the default channel parameters
cc_GetParmEx( )	retrieves parameters containing variable data passed from the firmware.
cc_GetSAPI( )	retrieves the service access point ID
cc_GetUsrAttr( )	returns the user attribute
cc_SetChanState( )	changes the maintenance state of an indicated channel
cc_SetDChanCfg( )	sets the D-channel configuration for a BRI station device
cc_SetDLinkCfg( )	configures a logical link
cc_SetDLinkState( )	sets the logical data link state
cc_SetParm( )	sets default call parameters

cc_SetParmEx( )	sets parameters requiring variable data to be passed down to the firmware
cc_SetUsrAttr( )	sets the user attribute

Error Handling Functions:	
cc_CauseValue( )	retrieves the cause of the last failure on a given device
cc_ResultMsg( )	returns a pointer to an error string
cc_ResultValue( )	retrieves the error/cause code related to an event
Tracing Functions:	
cc_StartTrace( )	starts the capture of all D channel information into a specified log file
cc_StopTrace( )	stops the trace and closes the file
Eventing Functions:	
cc_GetEvtMsk( )	retrieves the current ISDN event mask
cc_SetEvtMsk( )	sets the event mask
cc_TermRegisterResponse( )	sends the acknowledgment for the CCEV_TERM_REGISTER event

Table 13. Data Link Layer Handling Functions

Function	Description
cc_GetFrame( )	retrieves the frame received by the application
cc_SndFrame( )	sends a frame to the data link layer

NOTE: These functions are available only when Layer 2 access is configured.

Table 14. Hold and Retrieve Functions

Function	Description
cc_HoldCall( )	places active calls on hold
cc_HoldAck( )	accepts hold requests from remote equipment
cc_HoldRej( )	rejects hold requests from remote equipment
cc_RetrieveCall( )	retrieves a call placed on hold from Hold state
cc_RetrieveAck( )	accepts a retrieve from hold request from remote equipment
cc_RetrieveRej( )	rejects a retrieve from hold request from remote equipment

Table 15. Global Tone Generation Functions

Function	Description
cc_PlayTone( )	plays a user-defined tone
cc_StopTone( )	stops the tone that is currently playing on a channel
cc_ToneRedefine( )	redefines the tones in the firmware tone template table

4.2. API Functions and Supported ISDN Technologies

The following table lists all of the ISDN API functions and indicates which functions can be used in each of the supported ISDN technologies. *Chapter 5. ISDN Function Reference* also provides information about supported technologies within each function description.

Table 16. ISDN API Functions and Supported Technologies

ISDN API Functions	PRI	BRI/2	BRI/SC	DPNSS	Q.SIG
cc_AcceptCall( )	*	*	*	*	*
cc_AnswerCall( )	*	*	*	*	*
cc_CallAck( )	*	*	*	*	*
cc_CallProgress( )	*	*	*	*	*
cc_CallState( )	*	*	*	*	*
cc_CauseValue( )	*	*	*	*	*
cc_Close( )	*	*	*	*	*
cc_CRN2LineDev( )	*	*	*	*	*
cc_DropCall( )	*	*	*	*	*
cc_GetANI( )	*	*	*	*	*
cc_GetBChanState( )	*		*	*	*
cc_GetBilling( )	*				
cc_GetCallInfo( )	*	*	*	*	*
cc_GetCES( )			*		
cc_GetChanId( )		*	*		
cc_GetCRN( )	*	*	*	*	*
cc_GetDChanState( )	*				
cc_GetDLinkCfg( )			*		
cc_GetDLinkState( )	*	*	*	*	*
cc_GetDNIS( )	*	*	*	*	*

ISDN API Functions	PRI	BRI/2	BRI/SC	DPNSS	Q.SIG
cc_GetEvtMsk( )	*	*	*	*	*
cc_GetFrame( )	*		*		*
cc_GetInfoElem( )	*		*	*	*
cc_GetLineDev( )	*	*	*	*	*
cc_GetMoreDigits( )	*				*
cc_GetNetCRV( )	*		*	*	*
cc_GetNonCallMsg( )	*	*	*	*	*
cc_GetParm( )	*	*	*	*	*
cc_GetParmEx( )	*	*	*	*	*
cc_GetSAPI( )			*		
cc_GetSigInfo( )	*		*	*	*
cc_GetUsrAttr( )	*	*	*	*	*
cc_GetVer( )	*	*	*	*	*
cc_HoldAck( )		*	*	*	*
cc_HoldCall( )		*	*	*	*
cc_HoldRej( )		*	*	*	*
cc_MakeCall( )	*	*	*	*	*
cc_Open( )	*	*	*	*	*
cc_PlayTone( )			*		
cc_ReleaseCall( )	*	*	*	*	*
cc_ReleaseCallEx( )	*	*	*	*	*
cc_ReqANI( )	*				

ISDN API Functions	PRI	BRI/2	BRI/SC	DPNSS	Q.SIG
cc_Restart( )	*	*	*	*	*
cc_ResultMsg( )	*	*	*	*	*
cc_ResultValue( )	*	*	*	*	*
cc_RetrieveAck( )		*	*		*
cc_RetrieveCall( )		*	*	*	*
cc_RetrieveRej( )		*	*		*
cc_SetBilling( )	*				
cc_SetCallingNum( )	*		*	*	*
cc_SetChanState( )	*			*	*
cc_SetDChanCfg( )	*		*		
cc_SetDLinkCfg( )			*		
cc_SetDLinkState( )	*		*	*	*
cc_SetEvtMsk( )	*	*	*	*	*
cc_SetInfoElem( )	*	*	*	*	*
cc_SetMinDigits( )	*	*	*	*	*
cc_SetParm( )	*	*	*	*	*
cc_SetParmEx( )	*	*	*	*	*
cc_SetUsrAttr( )	*	*	*	*	*
cc_SndFrame( )	*		*		*
cc_SndMsg( )	*	*	*	*	*
cc_SndNonCallMsg( )	*	*	*	*	*
cc_StartTrace( )	*	*	*	*	*

ISDN API Functions	PRI	BRI/2	BRI/SC	DPNSS	Q.SIG
cc_StopTone( )			*		
cc_StopTrace( )	*	*	*	*	*
cc_TermRegisterResponse( )			*		
cc_ToneRedefine( )			*		
cc_WaitCall( )	*	*	*	*	*

5. ISDN Function Reference

The Dialogic ISDN API functions are application-specific programming interfaces that provide standard software interrupts, calls, and data formats for developing ISDN applications. This chapter provides a detailed description of each ISDN API function included in the *cclib.h* file.

This chapter also includes the following information:

- function description format
- programming convention format
- definitions of call reference numbers (CRNs), call reference values (CRVs), and line device handles
- instructions for interpreting function failures

5.1. Function Description Format

The following table describes the information that is provided for each ISDN API function. The functions are listed in alphabetical order in this chapter.

NOTE: Refer to the "Technology" line in the function overview table for specific ISDN technology applicability.

Table 17. ISDN Function Description Format

Section	Provides:	
Function Header	the function name and briefly states its purpose.	
Function Overview	an overview of the function, including the following:	
	• 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.
	• Category	Lists the category classification of the function (see *Chapter 4. ISDN Function Overview*).
	• Mode	Indicates whether the function is asynchronous, synchronous, or both.
	• Technology	Indicates the technology/technologies (BRI/2, BRI/SC, PRI, DPNSS, Q.SIG) supported by the function. A filled box designates a supported technology.
Function Description	a detailed description of the function operation, including parameter descriptions	
Termination Events	the events that may be sent to the application at the completion of the asynchronous function. (This does not apply to synchronous programming models.)	
Cautions	warnings and reminders	
Example	C language coding example(s) showing how the function can be used in application code	
Errors	specific error codes that may be returned by the function	
See Also	a list of related functions	

5.2. Programming Conventions

The Dialogic ISDN library functions use the following format:

cc_function(reference, parameter1, parameter2, parameterN, mode)

Where:

- **cc_function** is the function name
- **reference** is an input field that directs the function to a specific line device or call when the reference is a CRN or a line device handle (see *Section 5.3. Function References: CRNs, CRVs, and Line Device Handles* below for more information)
- **parameter1, parameter2, parameterN** are input fields
- **mode** is an input field indicating how the function is executed. Set the value to EV_ASYNC for asynchronous mode execution or EV_SYNC for synchronous mode execution.

5.3. Function References: CRNs, CRVs, and Line Device Handles

Most functions ask for the line device handle and/or the call reference number (CRN), which together enable applications to be written independent of the hardware type or signaling protocol.

The line device handle is a unique logical number assigned to a specific device or device group by the ISDN API. The line device handle enables the API function to address any system resource using a single device identifier. The system architecture also permits more than one device to be addressed as a unit, as needed to process a call.

A call reference number (CRN) identifies a call on a specific *line device*. The CRN is created by the ISDN API library when a call is requested either by the application or the network. The relationship between the CRN and a line device is established when a call is requested and acknowledged by the other end. The valid lifespan of the CRN is the duration of the call. Afterward, the CRN can be reassigned.

A call reference value (CRV), which conforms to the Q.931 standard, is a network-assigned value that is used to identify a call on a specific line device. The CRV is transmitted over the network and maintained by the ISDN firmware. The ISDN firmware maintains a table to match the host-assigned CRN and the network-associated CRV. Use the **cc_GetCRN( )** function to obtain the CRN. The CRV for a particular CRN can be obtained by using the **cc_GetNetCRV( )** function.

The following list summarizes the use and assignment of CRNs and CRVs and offers some additional points to keep in mind when using Dialogic ISDN library functions:

- Each CRN is a unique number in the system.
- A call is **not** associated with a **physical port**.

New CRNs and CRVs are created either when CCEV_OFFERED occurs or when **cc_MakeCall( )** successfully sends a setup message to the network.

- The CRN/CRV is no longer valid after **cc_ReleaseCall( )**, **cc_ReleaseCallEx( )**, or **cc_Restart( )** is issued.
- After a CRN/CRV of a given value is released, it may be reassigned for subsequent calls.

5.4. Interpreting Function Call Failures

The Dialogic ISDN software architecture uses two different levels of fault reporting to indicate the failure or success of a function call:

- High level check – Function call failure or success is indicated by a result value: 0 = success and <0 = failure. That is, when the function fails, the function call is rejected immediately and a value <0 is returned. This means that the library or driver is unable to execute the request because it is not ready or because the request is not valid in the current state. When a value <0 is received, use the **cc_CauseValue( )** function to retrieve the reason for the failure. Use the **cc_ResultMsg( )** function to interpret the reason.
- Low level check (asynchronous functions only) – In asynchronous functions, the firmware returns a termination event, in addition to the result value, to indicate the success or failure of the function. When a function fails, a result

value <0 is returned along with the CCEV_TASKFAIL event. However, when an asynchronous function call fails, a result value of 0 (indicating success) can be returned along with the CCEV_TASKFAIL termination event, which indicates failure. This means that the library has accepted the request that was sent to the firmware, but at that moment, the request cannot be fulfilled due to specific circumstances or conditions. To retrieve the reason for the failure, use the **cc_ResultValue( )** function. Use the **cc_ResultMsg( )** function to interpret the value.

For more information on CCEV_TASKFAIL and other termination events, see *Chapter* 7. ISDN Events and Errors. For information on handling events, see *Section 8.2.2. Handling Events*.

Name:	int cc_AcceptCall(crn, rings, mode)	
Inputs:	CRN crn	• call reference number
	int rings	• number of rings before return
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_AcceptCall( )** function responds to an incoming call request. A CCEV_OFFERED event on the completion of **cc_WaitCall( )** signifies an incoming call request. The **cc_AcceptCall( )** function sends an ALERTING message to the network to indicate that the destination is ringing and to stop the network from sending any further information. The ALERTING message also stops the protocol layer 3 timer under fast connect. After the successful completion of the **cc_AcceptCall( )** function, the call state changes from Offered to Accepted.

This command is not required in most applications if the application can respond within the protocol timeout restriction.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
rings:	Specifies how long the protocol handler will wait before returning to the calling entity. The **rings** parameter is not used for ISDN and will be ignored. Set **rings** to 0.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Events

- CCEV_ACCEPT - indicates that an ALERTING message has been sent to the network.

- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   printf("Accepting call\n");
   if ( cc_AcceptCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}
```

```
int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Error codes from the **cc_AcceptCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

- **cc_WaitCall( )**
- **cc_AnswerCall( )**

Name:	int cc_AnswerCall(crn, rings, mode)	
Inputs:	CRN crn	• call reference number
	int rings	• number of rings before return
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category	Call control	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_AnswerCall( )** function accepts a connection request from the remote end and connects the call. This function is equivalent to a conventional "offhook" command in answering an incoming call. The **cc_AnswerCall( )** function is used any time after:

- CCEV_OFFERED, CCEV_PROGRESSING, or CCEV_ACCEPT is received in asynchronous mode
- the successful completion of **cc_WaitCall( )** in synchronous mode

In asynchronous mode, the CCEV_ANSWERED event indicates successful completion of the **cc_AnswerCall( )** function. After the successful completion of the function call, the call state changes from Offered or Accepted, if **cc_AcceptCall( )** was used, to Connected.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
rings:	Specifies how long the protocol handler will wait before returning to the calling entity. In ISDN systems, **rings** must be set to zero or an error will be returned.
Mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Events

- CCEV_ANSWERED - indicates that a CONNECT message has been sent to the network.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

This function is called only after an inbound call has been detected.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);
```

```
}

int callfail(CRN crn)
{
        LINEDEV ld;
       cc_CRN2LineDev(crn,&ld);
       procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Error codes from the **cc_AnswerCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

- **cc_WaitCall( )**
- **cc_DropCall( )**

Name:	int cc_CallAck(crn, newLineDev, msg_id)	
Inputs:	CRN crn	• call reference number
	LINEDEV newLineDev	• new line device handle
	int msg_id	• message id
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_CallAck( )** function allows the application to send the first response to an incoming call, after the CCEV_OFFERED event is received, in asynchronous mode, or after **cc_WaitCall( )** returns, in synchronous mode.

NOTE: Controlling the first response to the incoming setup message is optional. The ISDN firmware assumes the control by default unless it is set up by the application (see the **cc_SetEvtMsk( )** function description for details).

Parameter	Description
crn:	The call reference number. Each call needs a valid CRN.
newLineDev:	The new line device handle for the channel to be used for the call. This parameter is reserved for future use. Set **newLineDev** to 0.
msg_id:	The message ID, either CALL_PROCEEDING or CALL_SETUP_ACK. **NOTE:** Applications that require overlap receiving should set **msg_id** to CALL_SETUP_ACK.

The application can use this function to indicate one of the following conditions to the network:

1. The received setup message contains all the information necessary to set up the call. The application should use the function in one of the following ways:
   - **cc_CallAck(crn, 0, CALL_PROCEEDING)** if B channel is acceptable

- **cc_CallAck(crn, newLineDev, CALL_PROCEEDING)** if a new B channel is desired.

2. The received setup message contains insufficient destination information. The application should use the function in one of the following ways:

   - **cc_CallAck(crn, 0, CALL_SETUP_ACK)** if the B channel is acceptable
   - **cc_CallAck(crn, newLineDev, CALL_SETUP_ACK)** if a new B channel is desired

■ Termination Event

- CALL_SETUP_ACK and CALL_PROCEEDING - indicate that the call setup message has been received.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   /*
```

```
    * The cc_CallAck() function needs to be called after
    * cc_WaitCall and before cc_CallProgress(), cc_AcceptCall()
    * and cc_AnswerCall()
    */
   if ( cc_CallAck(crn,0,CALL_PROCEEDING) < 0 )
        callfail(crn);

   printf("Accepting call\n");
   if ( cc_AcceptCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Error codes from the **cc_CallAck( )** function include the following:

Error Code	Description
E_ISBADIF | ERR_ISDN_LIB	Bad interface number
E_ISBADTS | ERR_ISDN_LIB	Bad time slot

See Also

- **cc_SetEvtMsk( )**

Name:	int cc_CallProgress(crn, indicator)	
Inputs:	CRN crn	• call reference number
	int indicator	• progress indicator
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_CallProgress( )** function sends a PROGRESS message to the network. This function can be called after CCEV_OFFERED occurs, in asynchronous mode, or after the **cc_WaitCall( )** function successfully completes, in synchronous mode. Applications may use the message on the D channel to indicate that the connection is not an ISDN terminal or that in-band information is available.

The **cc_CallProgress( )** function is not needed in terminating mode. It may be used in a drop-and-insert configuration when an in-band Special Information Tone (SIT) or call progress tone is sent to the network.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
indicator:	Specifies the progress indicator. The values are: • CALL_NOT_END_TO_END_ISDN - In drop-and-insert configurations, the application has the option of providing this information to the network. • IN_BAND_INFO - In drop-and-insert configurations, the application has the option of notifying the network that in-band tones are available.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
       procdevfail(devhdl);

   /*
   •    using cc_CallProgress(crn,CALL_NOT_END_TO_END_ISDN)
   •    to indicate that the remote is not an ISDN terminal.
   •    This function call is optional.
    */
   if ( cc_CallProgress(crn,CALL_NOT_END_TO_END_ISDN))
       callfail(crn);

   printf("Accepting call\n");
   if ( cc_AcceptCall(crn, 0, EV_SYNC) < 0 )
       callfail(crn);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
       callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
       callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
       callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
       LINEDEV ld;
       cc_CRN2LineDev(crn,&ld);
       procdevfail(ld);
```

```
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Error codes from the **cc_CallProgress( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

None

Name:	int cc_CallState(crn, state_buf)	
Inputs:	CRN crn	• call reference number
	int* state_buf	• pointer to requested state number
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_CallState( )** function retrieves the state of a call associated with a particular call reference number (CRN). The call state, which is stored in the firmware, changes only when a valid message is sent or received during a given state. For more on call states, see *Section 3.3. ISDN Call Control States*.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
state_buf:	The pointer to the location where the state value is returned. The supported states are as follows: • CCST_ACCEPTED - An inbound call was accepted; the call is in the Accepted state. • CCST_ALERTING - The call is waiting for the destination party to answer; the call is in the alerting state (call alerted sent or received) • CCST_CONNECTED - An inbound or outbound call was connected; the call is in the Connected state. • CCST_DIALING - An outbound call request was received; the call is in the Dialing state. • CCST_DISCONNECTED - The call was disconnected from the network; the call is in the Disconnected state. • CCST_IDLE - The call is not active; the call is in the Idle state.

Parameter	Description
	• CCST_NULL - The call was released; the call is in the Null state. • CCST_OFFERED - An inbound call was received; the call is in the Offered state.

■ Cautions

Due to normal process latency time, the state value acquired may not reflect the current state of the call. The state retrieved will be associated with the last event received by the application.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   int state;


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
      procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
   {
      callfail(crn);
   /*
    * using cc_CallState(crn, &state) to retrieve the current
    * call state from the firmware.
    */

      if ( cc_CallState(crn,&state) < 0 )
           callfail(crn);

      if(state == CCST_DISCONNECTED)
```

```
      {
            if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
                 callfail(crn);

            if ( cc_ReleaseCall(crn) < 0 )
                 callfail(crn);

            exit(1);
      }
   }
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Error codes from the **cc_CallState( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISCALLRELATED	Call related event

See Also

None

Name: int cc_CauseValue(linedev)
Inputs: LINEDEV linedev • line device handle
Returns: cause value code
Includes: cclib.h
Category: System tools
Mode: synchronous
Technology: BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_CauseValue( )** function retrieves the error/cause code of a failure on a given device when a function returns a -1. Use **ResultMsg( )** to retrieve the ASCII string and interpret the reason code associated with the cause value.

NOTES: **1.** The **cc_CauseValue( )** function is equivalent to **ATDV_LASTERR( )**.

2. To retrieve the cause of an event or the cause information element of ISDN messages, use the **cc_ResultValue( )** function.

There are three cause/error locations:

- Firmware (ERR_ISDN_FW) - returned when there is a firmware-related cause/error. Firmware errors are listed in the *isdncmd.h* file.
- Network (ERR_ISDN_CAUSE) - returned with a disconnect or reject event (for example, CCEV_DISCONNECTED, CCEV_HOLDREJ, or CCEV_RETRIEVEREJ). Network cause values are listed in the *isdncmd.h* file.
- ISDN Library (ERR_ISDN_LIB) - returned when there is a library-related cause/error. Library errors are listed in the *isdnerr.h* file.

See *Section 7.2. Error Handling* for a listing of errors and cause values.

Parameter	Description
linedev:	The specified line device handle.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   int state;
   int reason;
   char *msg;


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
       procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
       callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
       callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
       LINEDEV ld;
       cc_CRN2LineDev(crn,&ld);
       procdevfail(ld);
}

int procdevfail(LINEDEV handle)
```

```
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

The **cc_CauseValue( )** function returns -1 when there is no error/cause code available for the specified line device. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

■ See Also

- **cc_ResultMsg( )**

Name:	int cc_Close(linedev)
Inputs:	LINEDEV linedev • line device handle
Returns:	0 on success < 0 on failure
Includes:	cclib.h
Category:	System control
Mode:	synchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_Close( )** function closes a previously opened line device. The application can no longer access the device via the specified line device handle after this function is called.

Parameter	Description
linedev:	The line device handle returned when the line device is opened.

■ Cautions

- The **cc_Close( )** function affects only the link between the calling process and the device. Other processes and devices are unaffected.
- The **cc_Close( )** function must be issued while the line device is in the Null state. The Null state occurs immediately after a call to either **cc_Open( )**, **cc_ReleaseCall( )** or **cc_ReleaseCallEx( )**.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
```

```
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
       procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
      callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
       callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
       callfail(crn);

   if ( cc_Close(devhdl ) < 0 )
       procdevfail(devhdl);


}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the returned value. Error codes are defined in the files *ccerr.h*, *isdnerr.h* and *isdncmd.h*.

Typically, a < 0 return code for the **cc_Close( )** function indicates that the function reference (the device number) is not valid for the function call.

See Also

- **cc_Open( )**
- **cc_CallState( )**

Name:	int cc_CRN2LineDev(crn, linedevp)	
Inputs:	CRN crn	• call reference number
	LINEDEV *linedevp	• pointer to a buffer to store the line device handle
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System control	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_CRN2LineDev( )** function is a utility function that matches a CRN to its line device handle. The function returns the line device handle associated with the specified call reference number (CRN).

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
linedevp:	Points to the buffer where the line device handle will be stored.

■ Cautions

The call reference number (CRN) exists after CCEV_OFFERED is received by the application or **cc_MakeCall( )** is issued. The CRN is valid until the **cc_ReleaseCall( )** function is issued.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
```

```
   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_CRN2LineDev( )** function indicates that the function reference (the CRN) is not valid for the function call.

Name:	int cc_DropCall(crn, cause, mode)	
Inputs:	CRN crn	• call reference number
	int cause	• reason for dropping the call
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Call control	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_DropCall( )** function allows the application to disconnect a call, specified by a CRN, at any time. The application must specify a reason for dropping the call. Valid causes are listed in *Table 18*.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
cause:	The reason for disconnecting or rejecting the call. The values listed in *Table 18* indicate common causes for dropping a call.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

Table 18. cc_DropCall( ) Causes

Value	Description
ACCESS_INFO_DISCARDED	Access information discarded
BAD_INFO_ELEM	Information element nonexistent or not implemented
BEAR_CAP_NOT_AVAIL	Bearer capability not available
CALL_REJECTED	Call rejected
CAP_NOT_IMPLEMENTED	Bearer capability not implemented

Value	Description
CHAN_DOES_NOT_EXIST	Channel does not exist
CHAN_NOT_IMPLEMENTED	Channel type not implemented
CHANNEL_UNACCEPTABLE	Channel is unacceptable
DEST_OUT_OF_ORDER	Destination out of order
FACILITY_NOT_IMPLEMENT	Requested facility not implemented
FACILITY_NOT_SUBSCRIBED	Facility not subscribed
FACILITY_REJECTED	Facility rejected
INCOMPATIBLE_DEST	Incompatible destination
INCOMING_CALL_BARRED	Incoming call barred
INTERWORKING_UNSPEC	Interworking unspecified
INVALID_CALL_REF	Invalid call reference
INVALID_ELEM_CONTENTS	Invalid information element
INVALID_MSG_UNSPEC	Invalid message, unspecified
INVALID_NUMBER_FORMAT	Invalid number format
MANDATORY_IE_LEN_ERR	Message received with mandatory information element of incorrect length
MANDATORY_IE_MISSING	Mandatory information element missing
NETWORK_CONGESTION	Network congestion
NETWORK_OUT_OF_ORDER	Network out of order
NO_CIRCUIT_AVAILABLE	No circuit available
NONEXISTENT_MSG	Message type nonexistent or not implemented
NORMAL_CLEARING	Normal clearing

Value	Description
NO_ROUTE	Network has no route to the specified transient network, or the network has no route to the destination.
NO_USER_RESPONDING	No user responding
NUMBER_CHANGED	Number changed
OUTGOING_CALL_BARRED	Outgoing call barred
PRE_EMPTED	Call preempted
PROTOCOL_ERROR	Protocol error, unspecified
REQ_CHANNEL_NOT_AVAIL	Requested circuit/channel unavailable
RESP_TO_STAT_ENQ	Response to status inquiry
SERVICE_NOT_AVAIL	Service not available
TEMPORARY_FAILURE	Temporary failure
TIMER_EXPIRY	Recovery on timer expired
UNASSIGNED_NUMBER	Unassigned number
UNSPECIFIED_CAUSE	Unspecified cause
USER_BUSY	User busy
WRONG_MESSAGE	Message type invalid in call state or not implemented
WRONG_MSG_FOR_STATE	Message type not compatible with call state

■ Termination Events

- CCEV_DROPCALL - indicates that a DISCONNECT message has been sent to the network.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

In order to release the call reference number, this function must be followed by a **cc_ReleaseCall( )** to prevent a blocking condition or memory allocation errors.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
      .
      .
      .
      .
      .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}


int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
```

```
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_DropCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

- **cc_ReleaseCall( )**
- **cc_MakeCall( )**
- **cc_WaitCall( )**

Name:	int cc_GetANI(crn, ani_buf)	
Inputs:	CRN crn	• call reference number
	char *ani_buf	• pointer to buffer where ANI will be stored
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetANI( )** function retrieves Automatic Number Identification (ANI) information (the calling party number) received in the ISDN setup message. The data returned is in a NULL terminated ASCII string.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
ani_buf:	The address of the buffer where ANI information is to be stored.

■ Cautions

Make sure the size of **ani_buf** is sufficient for the ANI string. Refer to the file *cclib.h* for the maximum allowable string defined by CC_ADDRSIZE. Typically, ANI strings are 4 to 20 characters long. CC_ADDRSIZE should be used to define the size of the buffer.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
```

```
    CRN       crn = 0;
    char     *devname = "dtiB1T1";
    char      ani_buf[CC_ADDRSIZE];

    if ( cc_Open( &devhdl, devname, 0 ) < 0 )
    {
         printf("Error opening device: errno = %d\n", errno);
         exit(1);
    }

    if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
         procdevfail(devhdl);

    printf("Retrieving ANI\n");
    if ( cc_GetANI(crn, ani_buf) < 0 )
         callfail(crn);

    if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
         callfail(crn);
       .
       .
       .
       .
       .
    if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
         callfail(crn);

    if ( cc_ReleaseCall(crn) < 0 )
         callfail(crn);

    if ( cc_Close( devhdl ) < 0 )
         printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetANI( )** function include the following:

Error Code	Description
E_ISBADCRN | ERR_ISDN_LIB	Bad call reference number
E_ISBADPAR | ERR_ISDN_LIB	Bad input parameter
E_ISNOINFOBUF | ERR_ISDN_LIB	Information buffer not ready

See Also

- **cc_WaitCall( )**
- **cc_ReqANI( )**
- **cc_MakeCall( )**

Name:	int cc_GetBChanState(linedev, bchstate_buf)	
Inputs:	LINEDEV linedev	• line device handle for the B channel
	int *bchstate_buf	• pointer to the location where the B channel state value is stored
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetBChanState( )** function retrieves the status of the B channel at any time.

Parameter	Description	
linedev:	The line device handle for the B channel.	
bchstate_buf:	Points to the buffer containing the requested B channel state value. The definitions of the possible channel states are:	
	ISDN_IN_SERVICE	B channel is in service
	ISDN_MAINTENANCE	B channel is in maintenance
	ISDN_OUT_OF_SERVICE	B channel is out of service

■ Caution

This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
```

```
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   int       bchanstate;

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

  /*
   * Using cc_GetBChanState() to get the current
   * B channel status.
   */
   if ( cc_GetBChanState(devhdl,&bchanstate) < 0 )
        procdevfail(devhdl);
   else if ( bchanstate != ISDN_IN_SERVICE )
   {
      printf("B Channel is not in service...\n");

      if ( cc_Close( devhdl ) < 0 )
           printf("Error closing device, errno = %d\n", errno);

      exit(1);
   }

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
```

```
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetBChanState( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

None

Name:	int cc_GetBilling(crn, billing_buf)	
Inputs:	CRN crn	• call reference number
	char *billing_buf	• pointer to billing string buffer
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	PRI (4ESS only)	

■ Description

The **cc_GetBilling( )** function gets the call charge information associated with the specified call. The billing information is in a NULL terminated ASCII string. The information is retrieved from the network.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
billing_buf:	Pointer to the buffer where the billing string will be stored. Use CC_BILLSIZE to define the buffer size.

■ Cautions

- Make sure the size of **billing_buf** is large enough to hold the string. The maximum string is defined by CC_BILLSIZE.
- **cc_GetBilling( )** may not function in all service-provider environments. Check whether retrieving Vari-A-Bill billing information is an option with the service provider.
- The **cc_GetBilling( )** function is valid only after the call is terminated and before the **cc_ReleaseCall( )** function is called.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
```

```
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   char billingbuf[CC_BILLSIZE];

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }
   printf("Making call\n");
   if ( cc_MakeCall(devhdl,&crn,"9933000",NULL,30,EV_SYNC) < 0 )
        procdevfail(devhdl);

      .
      .
      .
      .
      .
     /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

         /*
           Using cc_ GetBilling(crn,billingbuf)to
           retrieve the call charge information.
           Note that not every network supports this feature
         */
   if ( cc_GetBilling(crn,billingbuf) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
       LINEDEV ld;
       cc_CRN2LineDev(crn,&ld);
       procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
       int reason;
       char *msg;
       reason = cc_CauseValue(handle);
       cc_ResultMsg(handle,reason,&msg);
       printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetBilling( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNOINFOBUF	Information buffer not ready

■ See Also

None

Name:	cc_GetCallInfo(crn, info_id, valuep)	
Inputs:	CRN crn	• call reference number
	int info_id	• call information identifier
	char *valuep	• pointer to information buffer
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
	isdnlib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetCallInfo( )** function gets the information elements associated with the CRN of an incoming message. The **cc_GetCallInfo( )** function must be used immediately after the message is received if the application requires the call information. The library will not queue the call information; subsequent messages on the same line device will be discarded if the previous messages are not retrieved.

NOTE: For new applications, use the **cc_GetSigInfo( )** function instead of **cc_GetCallInfo( )**. The **cc_GetSigInfo( )** function allows buffers to be set up, which enables the application to store and retrieve multiple messages.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
info_id:	The call information identifier (see *Table 19* below).
valuep:	Points to the buffer where the call information will be stored.

The following table provides definitions of possible **info_id** parameters.

Table 19. cc_GetCallInfo( ) Info_ID Definitions

info_id	Definition
U_IES	Information Elements (IEs) in CCITT format. The **cc_GetCallInfo( )** function retrieves all unprocessed IEs in CCITT format. Be sure to allocate enough memory (up to 256 bytes) to hold the retrieved IEs. The IEs are returned as raw data and must be parsed and interpreted by the application. Use IE_BLK to retrieve the unprocessed IEs. For a description of the IE_BLK data structure, see *Section 6.6. IE_BLK*. See *Appendix C* for descriptions of information elements specific to the DPNSS protocol.
UUI	User-to-user information. The user information data returned is application-dependent. The user information is retrieved using the USRINFO_ELEM data structure. For a description of the return format for UUI, see *Section 6.16. USRINFO_ELEM*.

■ Cautions

- Make sure the size of the information buffer is large enough to hold the string.
- The CCEV_NOFACILITYBUF event will be received by the application for every incoming ISDN message that contains the Network Facility IE. The event is received because the first four IEs are stored in the ISDN library and the remaining ones are discarded. The CCEV_NOFACILITYBUF event can be ignored. The IE can be retrieved using the **cc_GetCallInfo(U_IES)** function or the **cc_GetInfoElem( )** function. The ability to retrieve just the Network Facility IE using the **cc_GetCallInfo( )** function is no longer supported.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
```

```
void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   char      dnis_buf[CC_ADDRSIZE];
   char      infbuf[MAXLEN_IEDATA];    /* buffer raw information in CCITT format */
   int       i;         /* for loop counter to print out information buffer contents */

   /* open the ISDN line device */
   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   printf("Getting call information\n");
   if ( cc_GetCallInfo(crn, U_IES, &infbuf) < 0 )
        callfail(crn);
   else
        /* print out the raw buffer information */
        printf("Received call information buffer contents:\n")
        for (i=1; i<=MAXLEN_IEDATA; i++) {
          printf("%c", infbuf[I]);
        } /* end for */
   }

   /* answer the call */
   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
      .
      .
      .
      .
      .
   /* drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   /* release the CRN */
   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* close the ISDN line device */
   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}  /* end main */

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
```

```
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetCallInfo( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNOINFOBUF	Information buffer not ready
ERR_ISDN_LIB | E_ISNOFACILITYBUF	Network Facility buffer not ready

■ See Also

None

Name:	int cc_GetCES(cesp, evtdatap)	
Inputs:	char *cesp	• pointer to connection endpoint suffix buffer
	void *evtdatap	• pointer to an event block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC	

■ Description

The **cc_GetCES( )** function retrieves the connection endpoint suffix (CES) associated with the CCEV_D_CHAN_STATUS event received from the event queue. The connection endpoint suffix specifies the telephone equipment associated with the station. Eight IDs (1 - 8) are supported when used as a network-side terminal. When used as a station-side terminal, only one ID (1) is supported.

Parameter	Description
cesp:	Pointer to the space containing the connection endpoint suffix.
evtdatap:	Pointer to the structure containing the event data. The pointer value is acquired through the Dialogic Standard Runtime Library (SRL) function **sr_getevtdatap( )**.

■ Cautions

- The **cc_GetCES( )** function applies only to BRI protocols.
- This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
```

```
long EventHandler(event)
{
   int            rc;
   L2_BLK         frame;
   unsigned int   resultValue;
   unsigned char  sapi, ces;
   int            device;
   void           *datap;

   device = sr_getevtdev();
   datap  = sr_getevtdatap();

   ...

   switch(event)
   {
      case CCEV_D_CHAN_STATUS:

         cc_GetSapi(&sapi, datap);
         cc_GetCes(&ces, datap);

         resultValue = cc_ResultValue(datap);

         switch(resultValue & ~(ERR_ISDN_FW))
         {
            case E_LINKUP:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_UP;
               break;

            case E_LINKDOWN:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_DOWN;
               break;

            case E_LINKDISABLED:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_DISABLED;
               break;

            default:
               printf("Got a bad result value (0x%X)\n", resultValue);
         }
         break;

      case CCEV_L2FRAME:
         if(rc = cc_GetFrame (dev, &frame) == 0)
         {
            sapi = frame.sapi;
            ces  = frame.ces;
            printf("Got a frame of length=%d for Sapi=%d Ces=%d\n", frame.length, sapi,
ces);
         }
         else
            printf("cc_GetFrame failed!\n");
         break;

      ...
   }

   return 0;
}
```

Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetCES( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number

See Also

- **cc_GetDLinkState( )**
- **cc_GetSAPI( )**

Name:	cc_GetChanId(crn, chanId)	
Inputs:	CRN crn	• call reference number
	CHAN_ID *chanId	• pointer to channel ID structure
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC	

■ Description

The **cc_GetChanId( )** function gets the last channel information received from messages for the specified CRN. This function is used after a call-related event is received for the board device (e.g., dtiB1)

Parameter	Description
crn:	The call reference number.
chanId:	Pointer to the Channel ID structure that contains the last channel preference known for the specified CRN.

■ Cautions

The channel ID information associated with a specific CRN may change after an incoming call is connected or answered.

■ Example

```
#include <windows.h> /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

/** Globals **/
CRN normal_crn = 0;         /* crn of a normal call */
CRN waitingcall_crn = 0;

/** Function prototypes **/
long event_handler(unsigned long event_handle);
int callfail (CRN crn);
int procdevfail (LINEDEV handle);
```

```
void main( )
{
   LINEDEV boarddev;
   .
   .

   if ( cc_Open ( &boarddev, "dtiB1", 0 ) < 0 )
   {
      printf ("Error opening board device: errno = %d\n", errno);
      exit (1);
   }

   .
   .

   sr_setparm ( SRL_DEVICE, SR_MODELTYPE, &sr_mode );
   sr_enbhdlr ( EV_ANYDEV, EV_ANYEVT, event_handler );

   if ( cc_WaitCall (boarddev, &normal_crn, NULL, -1, EV_ASYNC) < 0 )
      procdevfail (linedev);

   .
   .
   .

}

long event_handler(unsigned long event_handle)
{
   CRN crn;
   int event;
   void *datap;
   char ani_buf[CC_ADDRSIZE];
   CHAN_ID chanId;

   event  = sr_getevttype(event_handle);
   datap  = sr_getevtdatap(event_handle);
   cc_GetCRN(&crn, datap);

   switch(event)
   {
      case CCEV_OFFERED:

         cc_GetChanId(crn, &chanId);
         switch(chanId.channel)
         {
            case NO_BCHAN:  /* waiting call */

               cc_GetANI (crn, ani_buf);
               if (strcmp (ani_buf, "9933000") == 0)
               {
                  waitingcall_crn = crn;

                  /* If no resource is available, then an active one must be freed */

                  if( cc_DropCall (active_crn, NORMAL_CLEARING, EV_ASYNC) < 0 )
                     callfail(crn);

                  /** Optionally, the active call can be put on Hold
                  if( cc_HoldCall (active_crn, EV_ASYNC) < 0 )
                     callfail(crn);
                  **/
               }
               else /* this call is not for us */
```

```
            {
               if( cc_DropCall (crn, NORMAL_CLEARING, EV_ASYNC) < 0 )
                  callfail(crn);
            }
            break;

         case DCHAN_IND:  /* non circuit switched calls */
            /* Ignore this call */
            break;

         default:  /* normal call */

            normal_crn = crn;
            switch(chanId.mode)
            {
               case PREFFERED:

                  if( cc_AcceptCall(crn, EV_ASYNC) < 0 )
                     callfail(crn);
                  break;

               case EXCLUSIVE:

                  if( cc_AcceptCall (crn, 0, EV_ASYNC) < 0 )
                     callfail(crn);
                  break;
            }
      }
      break;


   case CCEV_ACCEPT:

      if( cc_AnswerCall (crn, 0, EV_ASYNC) < 0 )
         callfail(crn);
      break;

   case CCEV_ANSWERED:

      if(crn == waitingcall_crn)
      {
          normal_crn = crn;
          waitingcall_crn = 0;
      }
      break;

         case CCEV_HOLDACK:

      if( (crn == normal_crn) && (callwaiting_crn != 0) )
      {
         cc_GetChanId(crn, &chanId);
         freed_device = chanId.channel;
         if( cc_AcceptCall(callwaiting_crn, num_rings, EV_ASYNC) < 0 )
            callfail(crn);
         active_crn = 0;
      }
      break;

   case CCEV_DROPCALL:

      if( cc_ReleaseCallEx (crn) < 0 )
         callfail(crn);

      if( (crn == normal_crn) && (callwaiting_crn != 0) )
      {
```

```
            cc_GetChanId(crn, &chanId);
            if( cc_AcceptCall(callwaiting_crn, num_rings, EV_ASYNC) < 0 )
               callfail(crn);
         }
         break;

case CCEV_RELEASE:

         if(active_crn == crn)
            active_crn = 0;
         else if(waitingcall_crn == crn)
            waitingcall_crn = 0;
         break;


       .
       .
       .
   }
}


int callfail (CRN crn)
{
   LINEDEV ld;

   cc_CRN2LineDev (crn,&ld);
   procdevfail (ld);
}

int procdevfail (LINEDEV handle)
{
   int reason;
   char *msg;

   reason = cc_CauseValue (handle);
   cc_ResultMsg (handle,reason, &msg);
   printf ("reason = %x - %s\n", reason, msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetChanId( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

- **cc_AcceptCall( )**
- **cc_MakeCall( )**

Name:	int cc_GetCRN(crnp, evtdatap)	
Inputs:	CRN *crnp	• pointer to the CRN
	void *evtdatap	• pointer to an event block
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetCRN( )** function retrieves the call reference number for the event in the event queue pointed to by **evtdatap**. The **evtdatap** pointer may be acquired using the **sr_getevtdatap( )** function in the Dialogic Standard Runtime Library (SRL).

If the event is channel or time slot related and not call related, **cc_GetCRN( )** returns a value < 0. If 0 is returned, the event is call related and the call reference number will be in the location pointed to by **crnp**.

Parameter	Description
crnp:	The address where the returned call reference number (CRN) is stored.
evtdatap:	Pointer to the structure containing the event data block. The pointer value is acquired using the SRL function **sr_getevtdatap( )**.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
```

```
#include "cclib.h"

main()
{
   .
   .
   .
   .
   if ( sr_enbhdlr( devhdl,CCEV_DISCONNECTED,discCallHdlr ) < 0 )
   {
        printf( "dtiEnable for DISCONNECT failed:  %s\n",
                              ATDV_ERRMSGP( SRL_DEVICE ) );
        return( 1 );
   }
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }
}

/***********************************************/
/* discCallHdlr - disconnect the active call       */
/***********************************************/
int discCallHdlr( )
{
    int devindx;
    int dev;
    int len;
    void *datap;
    CRN crn;

    dev = sr_getevtdev();
    len = sr_getevtlen();
    datap = sr_getevtdatap();

    cc_GetCRN (&crn, datap);

    if ( cc_DropCall( crn,NORMAL_CLEARING,EV_ASYNC ) < 0 )
    {
        int lasterr = cc_CauseValue(dev);
        char *errmsg;
        if ( cc_ResultMsg(dev,lasterr,&errmsg) == 0 )
            printf( "\tcc_DropCall() Error: ( %x ) %s\n",lasterr,errmsg);
    }

    return( 0 );
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_GetCRN( )** function indicates that the function reference (the device number) is not valid for the function call.

See Also

- **cc_WaitCall( )**
- **cc_MakeCall( )**

Name:	int cc_GetDChanState(boarddev, dchstate_buf)	
Reference:	LINEDEV boarddev	• line device handle for the D channel board
	int *dchstate_buf	• pointer to the location where the requested D channel state value is stored
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	PRI (excluding DPNSS and Q.SIG)	

■ Description

The **cc_GetDChanState( )** function retrieves the status of the D channel of a specified board at any time.

Parameter	Description	
boarddev:	The line device handle for the D channel board.	
dchstate_buf:	Points to the buffer containing the requested D channel state value. The definitions of the possible channel states are:	
	DATA_LINK_UP	D channel layer 2 is operable
	DATA_LINK_DOWN	D channel layer 2 is inoperable

■ Cautions

The **cc_GetDChanState( )** function applies only to ISDN PRI technology. For ISDN BRI technology, use the **cc_GetDLinkState( )** function.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
```

```
void main()
{
  short devhdl;   /* device handle for D channel */
  int dchanstate; /* the space for cc_GetDChanState output */
  .
  .
  .
  if ( cc_Open(devhdl,"dtiB1T1", 0 ) < 0 )
     exit(1);

   */
   * Using cc_GetDChanState() to get the
   * layer 2 status.
   */
  if ( cc_GetDChanState(devhdl,&dchanstate) < 0 )
       procdevfail(devhdl);
  else if ( dchanstate != DATA_LINK_UP )
  {
      printf("D Channel link is inoperable...\n");
      exit(1);
  }

  /* The layer 2 is OK, continue the program. */

  .
  .
  .
  .
  .
  if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetDChanState( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number

See Also

- **cc_SetDChanCfg( )**

Name:	int cc_GetDLinkCfg(bdev, dlinkptr, dlinkcfgptr)	
Inputs:	LINEDEV bdev	• device handle
	DLINK *dlinkptr	• pointer to data link information block
	DLINK_CFG *dlinkcfgptr	• pointer to location of D channel logical link configuration block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC	

■ Description

The **cc_GetDLinkCfg( )** function retrieves the configuration of a logical link. The logical link is configured using the **cc_SetDLinkCfg( )** function.

Parameter	Description
bdev:	Station device handle.
dlinkptr:	Pointer to the data link information block. See *Section 6.4. DLINK* for a description of the elements of this data structure.
dlinkcfgptr:	Pointer to the buffer containing the data link logical link configuration block. See *Section 6.5. DLINK_CFG* for a description of the elements of this data structure.

■ Cautions

Make sure that the DLINK_CFG data structure is initialized to zero.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
```

```
#include "cclib.h"

/* Global variables */

LINEDEV      ldev;   /* Board device handle */


main()
{
   DLINK dlink;
   DLINK_CFG cfg;

   .
   .
   .

   dlink.sapi = 0;
   dlink.ces = 1;

   /* Get config parameters for ces 1, sapi 0 */
   if ( cc_GetDLinkCfg(ldev, &dlink, &cfg) < 0) {
      printf("error");
   } else {
      printf(" tei=0x%X  state=0x%X  protocol=0x%X\n",
                           cfg.tei, cfg.state, cfg.protocol);
      .
      .
      .
   }
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetDLinkCfg( )** function include the following:

Error Code	Description
E_BADDEV | ERR_ISDN_LIB	Bad Device Descriptor
E_INVNDIINTERFACE | ERR_ISDN_LIB	Invalid NDI Interface
E_INVNRB | ERR_ISDN_LIB	Invalid NRB

■ See Also

- **cc_SetDLinkCfg( )**

Name: int cc_GetDLinkState(bdev, dlinkptr, state_buf)

Inputs:

LINEDEV bdev	• device handle
DLINK *dlinkptr	• pointer to data link information block
int *state_buf	• pointer to location of D channel state

Returns: 0 on success
< 0 on failure

Includes: cclib.h

Category: System tools

Mode: synchronous

Technology: BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_GetDLinkState( )** function retrieves the logical data link state (operable, inoperable or disabled) of the specified board device for PRI or station device for BRI.

Parameter	Description
bdev:	Board device handle for PRI, station device handle for BRI.
dlinkptr:	Pointer to the data link information block (DLINK). See *Section 6.4. DLINK* for a definition of the data link information block data structure. See the Example code for details.
state_buf:	Pointer to the buffer containing the requested data link state value. Possible data link states are:

DATA_LINK_UP	channel layer 2 is operable
DATA_LINK_DOWN	channel layer 2 is inoperable
DATA_LINK_DISABLED	channel layer 2 was disabled and cannot be reestablished

■ Cautions

None

■ Example

```
/* BRI code example */

#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/* Global variables */

LINEDEV         lbuf;


main ()
{
DLINK dlink;
char  ces;
.
.
.
      /* open BRI station device bris1 */
      dlink.sapi = 0;
      /* check state of each data link */
      for(ces = 1; ces <= 8; ces++)
      {
          /* initialize connection endpoint suffix */
          dlink.ces = ces;
          /* get current data link state */
          if ( cc_GetDLinkState(l_buf, &dlink, &state_buf) == 0)
          {
             /* if data link is up */
             if (state_buf == DATA_LINK_UP)
                printf("ces%02x) is up \n", ces);
             /* if data link is not up */
             else if (state_buf == DATA_LINK_DOWN)
                printf("ces(%02x) is down\n", ces);
          }
          else
             printf("error");
      }
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetDLinkState( )** function include the following:

Error Code	Description
ERR_ISDN_LIB|E_ISBADIF	Bad interface number
ERR_ISDN_LIB|E_ISNOMEM	Cannot map or allocate memory

See Also

- **cc_GetCES( )**
- **cc_GetSAPI( )**

Name:	int cc_GetDNIS(crn, dnis_buf)	
Inputs:	CRN crn	• call reference number
	char *dnis_buf	• pointer to DNIS buffer
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetDNIS( )** function gets the dialed number information string (destination address/called party number) associated with a specific call reference number (CRN). The information is in a NULL terminated ASCII string.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
dnis_buf:	The address of the buffer where the DNIS (destination address/called party number) will be stored. Use CC_ADDRSIZE to define the size of the buffer.

■ Cautions

Make sure the size of the **dnis_buf** is large enough to hold the DNIS. It must be no smaller than CC_ADDRSIZE. Refer to the file *cclib.h* for the maximum allowable string.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
```

```
   char     *devname = "dtiB1T1";
   char      dnis_buf[CC_ADDRSIZE];

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   printf("Getting DNIS\n");
   if ( cc_GetDNIS(crn,dnis_buf) < 0 )
        callfail(crn);
   else
        printf("cc_GetDNIS succeeded: %s\n",dnis_buf);


   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be

used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetDNIS( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNOINFOBUF	Information buffer not ready

See Also

- **cc_WaitCall( )**
- **cc_MakeCall( )**

Name: int cc_GetEvtMsk(bdev, maskp)
Inputs: LINEDEV bdev • device handle
ULONG *maskp • pointer to mask buffer
Returns: 0 on success
< 0 on failure
Includes: cclib.h
Category: System tools
Mode: synchronous
Technology: BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_GetEvtMsk( )** function retrieves the current ISDN event mask for a specified board device handle for PRI or station device handle for BRI.

Parameter	Description
bdev:	Board device handle for PRI, station device handle for BRI.
maskp:	The address of the buffer that stores the current ISDN event mask.

Table 20 lists the possible **bitmask** values and the actions that they control.

Table 20. Bitmask Values

Bitmask Type	Action	Default Setting
CCMSK_ALERT	Receiving CCEV_ALERTING	Event enabled
CCMSK_CALLACK_SEND	Application sends first response to SETUP message: CALL_SETUP_ACK CALL_PROCEEDING	Firmware sends first response to SETUP
CCMSK_PROCEEDING	Receiving CCEV_PROCEEDING	Event enabled
CCMSK_PROGRESS	Receiving CCEV_PROGRESSING	Event enabled

Bitmask Type	Action	Default Setting
CCMSK_SERVICE	Receiving CCEV_SERVICE. When this event arrives, the application should respond with a status message using **cc_SndMsg( )**. The firmware will not automatically respond to this message.	Not enabled
CCMSK_SERVICE_ACK	Receiving CCEV_SETCHANSTATE. When this event is masked, the **cc_SetChanState( )** function may be blocked.	Not enabled
CCMSK_SETUP_ACK	Receiving CCEV_SETUP_ACK	Event not enabled
CCMSK_STATUS	Receiving CCEV_STATUS	Not enabled
CCMSK_STATUS_ ENQUIRY	Receiving CCEV_STATUS_ENQUIRY. When this event arrives, the application should respond with a status message using the **cc_SndMsg( )** function. The firmware will not automatically respond to this message.	Not enabled
CCMSK_TERMINATE	Delay RELEASE COMPLETE message until host application calls the **cc_ReleaseCall( )** function.	Firmware does not wait for **cc_ReleaseCall( )** and sends RELEASE COMPLETE when RELEASE is received.
CCMSK_TMREXPEVENT	Receiving the CCEV_TIMER event. This event is generated when some timer expires at Layer 3. Timer ID, Call ID and the value of the timer are returned.	Not enabled

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   ULONG    *evtmskvalue;
   USHORT    uSubMask;

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   if ( cc_GetEvtMsk (devhdl, &evtmskvalue) < 0 ) {
        procdevfail(devhdl);
   }
   else {
      /* check the event mask for enabled/disabled events */
       uSubMask = (USHORT) (evtmskvalue);

       if (CCMSK_ALERT && (uSubMask & CCMSK_ALERT) {
              printf("ALERTING EVENT ENABLED\n");
       }
       if (CCMSK_PROCEEDING && (uSubMask & CCMSK_PROCEEDING) {
              printf("PROCEEDING EVENT ENABLED\n");
       }
       if (CCMSK_PROGRESS && (uSubMask & CCMSK_PROGRESS) {
              printf("PROGRESSING EVENT ENABLED\n");
       }
       if (CCMSK__SETUP_ACK && (uSubMask & CCMSK_SETUP_ACK) {
              printf("SETUP_ACK event enabled for firmware to send SETUP_ACK\n");
       }
   }
        .
        .
       etc...
        .
        .
        .
        .
} /* end main */
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetEvtMsk( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

■ See Also

cc_SetEvtMsk( )

Name:	int cc_GetFrame(linedev, rcvfrmptr)	
Inputs:	LINEDEV linedev	• line device handle for D channel
	L2_BLK *rcvfrmptr	• pointer to the received frame buffer
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Data link layer handling	
Mode:	synchronous	
Technology:	BRI/SC; PRI (excluding DPNSS)	

■ Description

The **cc_GetFrame( )** function retrieves the frame received by the application. This function is used after a CCEV_L2FRAME event is received. Each CCEV_L2FRAME event is associated with one frame. This function is used for the data link layer only.

To enable Layer 2 access, set parameter number 24 to 01 in the firmware parameter file. When Layer 2 access is enabled, only the **cc_GetFrame( )** and **cc_SndFrame( )** functions can be used (no calls can be made).

Parameter	Description
linedev:	The line device handle for the D channel.
rcvfrmptr:	The pointer to the buffer where the received frame is to be stored. The L2_BLK data structure contains the retrieved frame. See *Section 6.7. L2_BLK* for a description of the data structure. See the Example code for details.

■ Cautions

- The **cc_GetFrame( )** function is called only after a CCEV_L2FRAME event is received. Refer to the protocol-specific parameter file.
- This function is not supported for the BRI/2 board or for the PRI DPNSS protocol.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"


/* Global variables */


int  size = 8;
LINEDEV  lbuf;
L2_BLK;  rcvfrmptr;
typedef long int (*EVTHDLRTYP)( );
.
.
.
int evt_hdlr( )
{
   int rc = 0;
   int ldev = sr_getevtdev( );
   unsigned long *ev_datap = (unsigned long *)sr_getevtdatap( );
   int len = sr_getevtlen( );

switch(sr_getevttype( ))
   {
     .
     .
     .
     case CCEV_L2FRAME:                  /* New frame received */


     if (rc = cc_GetFrame(ldev, &rcvfrmptr) != 0)
     {
     /* Process error condition */
     }
     else
     /* Process the frame and call control function */
     .
     .
     .


     break;
     .
     .
     }
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetFrame( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISNOINFO	Information not available

See Also

- **cc_SndFrame( )**

Name:	int cc_GetInfoElem(linedev, iep)	
Reference:	LINEDEV linedev	• line device handle of the B channel board
Inputs:	IE_BLK *iep	• pointer to the information element buffer
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetInfoElem( )** function gets information elements associated with a line device for an incoming message. The **cc_GetInfoElem( )** function must be used immediately after the message is received if the application requires the call information. The library will not queue the call information; subsequent messages on the same line device will be discarded if the previous messages are not retrieved.

Parameter	Description
linedev:	The B channel board line device handle.
iep:	The starting address of the information element block. The information elements are contained in the IE_BLK data structure. See *Section 6.6. IE_BLK* for a description of the data structure. See the example code for details.

■ Caution

- This function is not supported for the BRI/2 board.
- The CCEV_NOFACILITYBUF event will be received by the application for every incoming ISDN message that contains the Network Facility IE. The event is received because the first four IEs are stored in the ISDN library and the remaining ones are discarded. The CCEV_NOFACILITYBUF event can be ignored. The IE can be retrieved using the **cc_GetCallInfo(U_IES)** function or the **cc_GetInfoElem( )** function. The ability to retrieve just the

Network Facility IE using the **cc_GetCallInfo( )** function is no longer supported.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <srllib.h>
#include <cclib.h>

long evt_handl;

void main(...)
{
    char        dev_name[20];
    LINEDEV     devhdl[MAXDEVS];
    CRN         crn[MAXDEVS];
    int         ch;


    sprintf(dev_name, "dtiB1");
    if(cc_Open(devhdl[0], dev_name, 0) != 0)
    {
       printf("cc_Open(%s) failed\n", dev_name);
       exit(0);
    }

    for(ch = 1; ch < MAXDEVS; ch++)
    {sprintf(dev_name, "dtiB1T%d", ch);
        if(cc_Open(devhdl[ch], dev_name, 0) != 0)
        {
           printf("cc_Open(%s) failed\n", dev_name);
           exit(0);
        }

        if(cc_WaitCall(devhdl[ch], &crn[ch], NULL, -1, EV_ASYNC) == -1)
        {
           printf("cc_WaitCall(%s) failed\n", dev_name);
           exit(0);
        }
    }

    while(FOREVER)
    {
        sr_waitevtEx(devhdl, MAXDEVS, -1, &evt_handl);
        process_event(evt_handl);
    }
 }



int  process_event(evthandl)
unsigned long evthandl;
{

    LINEDEV event_dev  = sr_getevtdev(evthandl);

    switch (sr_getevttype(evthandl))
    {
       case CCEV_NOTIFY:
       {
          IE_BLK ie;
```

```
          int i;

          printf("%s: CCEV_NOTIFY cc_GetInfoElem =%d\n",
                ATDV_NAMEP(event_dev), cc_GetInfoElem(event_dev, &ie) );
          printf("IE %d)= ", ie.length);
          for(i = 0; i < ie.length; i++)
            printf("%02X, ", (unsigned char)ie.data[i]);
          printf("\n");
                  .
                  .
      }
      break;
                  .
                  .
                  .
   }
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetInfoElem( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNOINFOBUF	Information buffer not ready
ERR_ISDN_LIB | E_ISNOFACILITYBUF	Network Facility buffer not ready

■ See Also

- **cc_GetCallInfo( )**
- **cc_SetInfoElem( )**

Name:	int cc_GetLineDev(linedevp, evtdatap)	
Inputs:	LINEDEV *linedevp	• pointer to the line device handle
	void *evtdatap	• pointer to an event block
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetLineDev( )** function retrieves the line device handle for an event from the event queue.

Parameter	Description
linedevp:	Pointer to the space containing the line device handle.
evtdatap:	Pointer to the structure containing the event data. The pointer value is acquired through the Dialogic SRL function **sr_getevtdatap( )**.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

main()
{
   .
   .
   .
   .
   if ( sr_enbhdlr( devhdl,CCEV_DISCONNECTED,discCallHdlr ) < 0 )
   {
        printf( "dtiEnable for DISCONNECT failed:  %s\n",
```

```
                            ATDV_ERRMSGP( SRL_DEVICE ) );
        return( 1 );
   }
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitcvt(-1);
   }
}

/************************************************/
/* discCallHdlr - disconnect the active call       */
/************************************************/
int discCallHdlr( )
{
    int dev;
    int len;
    void *datap;
    CRN crn;

    len = sr_getevtlen();
    datap = sr_getevtdatap();

    cc_GetLineDev(&dev,datap);
    cc_GetCRN (&crn, datap);

    if ( cc_DropCall( crn,NORMAL_CLEARING,EV_ASYNC ) < 0 )
    {
        int lasterr = cc_CauseValue(dev);
        char *errmsg;
        if ( cc_ResultMsg(dev,lasterr,&errmsg) == 0 )
            printf( "\tcc_DropCall() Error: ( %x ) %s\n",lasterr,errmsg);
    }

    return( 0 );
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_GetLineDev( )** function indicates that the function reference (the CRN) is not valid for the function call.

■ See Also

- **cc_GetCRN( )**

Name:	int cc_GetMoreDigits(crn, numofdigs, timeout, mode)	
Inputs:	CRN crn	• call reference number
	int numofdigs	• number of digits to be collected
	long timeout	• timeout value in seconds
	unsigned long mode	• asynchronous or synchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Call control	
Mode:	asynchronous or synchronous	
Technology:	PRI (excluding DPNSS)	

■ Description

The **cc_GetMoreDigits( )** function collects more digits via overlap receiving. After an incoming call is received, the application examines the completeness of the destination address. If more digits are needed, the application calls the **cc_GetMoreDigits( )** function with the number of additional digits to be collected. The function is returned when all requested digits are collected.

The collected digits can be retrieved by calling the **cc_GetDNIS( )** function. When enough digits have been collected, the application must use the **cc_CallAck( )** function to acknowledge the setup message.

Parameter	Description
crn:	The call reference number. Each call needs a valid CRN.
numofdigs:	The number of digits to be collected.
timeout:	Specifies the amount of time in seconds in which the additional digits must be collected. The function returns unconditionally when the timer expires. The **timeout** parameter is used to prevent a blocked situation in which the application expects more digits than the network provides. The timeout value must be a non-zero positive value. (A value < 0 means that the additional digits can be collected "forever.") **NOTE:** The **timeout** parameter is used in synchronous mode only.
mode:	Specifies either asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Event

- CCEV_MOREDIGITS - indicates that the function call is successful and the requested digits have been received.
- CCEV_TASKFAIL - indicates that the function failed and that the request/message was rejected by the firmware. The application can use **cc_Restart( )** after this event is received to reset the channel.

■ Cautions

This function is not supported for the PRI DPNSS protocol.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

#define MIN_DNIS    4

void main()
```

```
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_GetDNIS(crn, dnis_buf) < 0 )
        callfail(crn);

     /*
      * The cc_GetMoreDigits() function can only be called
      * after the cc_WaitCall and before cc_CallProgress(),
      * cc_AcceptCall() and cc_AnswerCall().
      */

   if ( (more_digits = (MIN_DNIS - strlen(dnis_buf) ) > 0)
           if ( cc_GetMoreDigits(crn, more_digits, time_out, EV_SYNC) < 0 )
                   callfail(crn);

   printf("Accepting call\n");
   if ( cc_AcceptCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
```

```
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetMoreDigits( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

cc_WaitCall( )

Name:	int cc_GetNetCRV(crn, netcrvp)	
Inputs:	CRN crn	• call reference number
	int *netcrvp	• pointer to network call reference buffer
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetNetCRV( )** function retrieves the network call reference value (CRV) for a specified call reference number (CRN). The CRN is assigned during either the **cc_MakeCall( )** function for outbound calls or the **cc_WaitCall( )** function for incoming calls. If an invalid host CRN value is passed, for example, the CRN of an inactive call, **cc_GetNetCRV( )** will return a value < 0 indicating failure.

NOTE: The **cc_GetNetCRV( )** function can be used to invoke the Two B Channel Transfer (TBCT) feature. The TBCT feature is invoked by sending a FACILITY message to the Network containing, among other things, the Call Reference Values (CRVs) of the two calls to be transferred. See *Appendix A - Call Control Scenarios* for more information on TBCT.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
netcrvp:	The pointer to the buffer where the network call reference value (CRV) will be stored.

■ Cautions

This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
```

```
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( sr_enbhdlr(devhdl, CCEV_OFFERED, (HDLR)OfferedHdlr) == < 0 )
   {
          printf ( 'sr_enbhdlr for OFFERED failed: %s\n', ATDV_ERRMSGP(devhdl));
          return (1);
   }

}

 /*
  * OfferedHdlr - Accept the incoming call
  */

  OfferedHdlr()
  {
      LINEDEV dev;
      int     len;
      void    *datap;
      CRN     crn;
      int     netcrv;

      dev    = sr_getevtdev();
      len    = sr_getevtlen();
      datap  = sr_getevtdatap();

  /* Obtain the call reference number */
  if (cc_GetCRN(&crn, datap) != 0
  {
        printf ( 'cc_GetCRN: error\n" );
        return < 0;

  }

  /* Use the CRN obtained above to get the Network CRV (Call Reference Value) */
  if (cc_GetNetCRV(crn, &netcrv) == 0)
  {
        printf ( "cc_GetNetCRV(%X, %d) success\n" , crn, netcrv);
        return 0;
  }
  else
  {
        printf ( "cc_GetNetCRV(%X) failure !!!\n", crn);
        return -1;
  }
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetNetCRV( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_BADCALLID	An invalid CRN was entered for which no call exists
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_BADDSL	Bad DSL value

■ See Also

- **cc_GetCRN( )**
- **cc_WaitCall( )**

Name:	int cc_GetNonCallMsg(devhndl, msgblkptr)	
Inputs:	LINEDEV devHndl	• board device handle associated with NULL or GLOBAL CRN event
	NONCRN_BLK *msgblkptr	• pointer to NULL or GLOBAL information structure
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetNonCallMsg( )** function retrieves call data for a GLOBAL or NULL CRN event, at the time the event occurs. The **cc_GetNonCallMsg( )** function must be used immediately after the event is received if the application needs the call information. The library will not queue the call information; subsequent messages on the same board device will overwrite this information if it is not retrieved immediately.

NULL events correspond to messages received with a dummy, or NULL, call reference value (CRV). These messages are of significance to all calls or channels on a particular trunk, that is, they do not correspond to a particular call. Therefore, the messages are delivered on the board level device (for example, briS1). The **cc_GetNonCallMsg( )** function can be used to retrieve information for the following NULL events:

- CCEV_INFONULL
- CCEV_NOTIFYNULL
- CCEV_FACILITYNULL

GLOBAL events correspond to messages received with a Zero call reference value. These messages are of significance to all calls or channels on a particular trunk, that is, they do not correspond to a particular call. Therefore, the messages are delivered on the board level device (for example, briS1). The **cc_GetNonCallMsg( )** function can be used to retrieve the information for the following GLOBAL events:

- CCEV_INFOGLOBAL
- CCEV_NOTIFYGLOBAL
- CCEV_FACILITYGLOBAL

Parameter	Description
devhndl:	The board device on which the GLOBAL or NULL event occurred.
msgblkptr:	Pointer to the NONCRN_BLK data structure that contains the information related to the GLOBAL or NULL CRN event. For a description of the data structure, see *Section 6.9. NONCRN_BLK*. See the Example code below for details.

■ Cautions

- Some IEs may require a Call Reference Value (CRV) to be part of the contents. The Call Reference, in this case, must be the Call Reference value assigned by the network, not the Call Reference Number (CRN) that is generated by Dialogic and retrieved using the **cc_GetCRN( )** function. It is up to the application to correctly format and order the IEs. Refer to the ISDN Recommendation Q.931 or the switch specification of the application's ISDN protocol for the relevant CCITT format. See the Example code for details.
- In order to receive GLOBAL and NULL events, an appropriate handler must be enabled on the board level device (see the **sr_enbhdlr( )** function in the *System Runtime Library Programmer's Guide*).
- The information related to a GLOBAL or NULL event must be retrieved immediately as it will be overwritten by the next event.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

int callinfoNullHdlr( )
{
   int devhndl;
   int i;
   NONCRN_BLK nullDataBlk;
   unsigned char tmpbuf[5], sbuf[256], *sptr;

   devhdnl = sr_getevtdev();
```

```
   if(cc_GetNonCallMsg(devhndl, &nullDataBlk) == 0)
   {
       int i;

       printf("Sapi = 0x%x.\n",nullDataBlk.sapi);
       printf("CES = 0x%x.\n",nullDataBlk.ces);
       printf("Raw IE data length = %d.\n",nullDataBlk.length);

       printf("IE data =:\n");
       for(i = 0; i < nullDataBlk.Iength; i++)
       {
          printf("0x%02x ", (unsigned char)nullDataBIk.data[i]);
       }
   }
   else
       tx_message("GetNonCallMsg failure" ,brd,devindx+1);

   return( 0 );
}

void main()
{
   LINEDEV  devhdl    = 0;
   CRN      crn = 0;
   char     devname = "briS1";      /* Device name for BRI station one */
   char     dnis_buf[CC_ADDRSIZE];
   char     infbuf[MAXLEN_IEDATA];  /* buffer raw information in CCITT format */
   iIt      i;                      /* for loop counter to print out information buffer
contents */

   /* open the ISDN line device */
   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   if ( sr_enbhdlr( devhdl, CCEV_INFONULL, (HDLR)callinfoNullHdlr ) == -1 )
   {
      printw( "dtiEnable for CCEV_CALLINFO failed:  %s\n",
         ATDV_ERRMSGP( SRL_DEVICE ) );
      return( 1 );
   }


      .
      .
      .
      .
      .

   /* close the ISDN line device */
   if ( cc_Close( devhdl ) < 0 )
        printf("Error closing device, errno = %d\n", errno);

}  /* end main */
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_GetNonCallMsg( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_BADPAR	Bad input parameter
ERR_ISDN_ISNOINFOBUF	Information buffer not ready
ERR_ISDN_LIB | ISNOFACILITYBUF	Network facility buffer not ready

See Also

- **cc_SndMsg( )**
- **cc_SndNonCallMsg( )**

Name:	int cc_GetParm(linedev, parm_id, valuep)	
Inputs:	LINEDEV linedev	• line device handle
	int parm_id	• parameter identifier
	CC_PARM *valuep	• pointer to value buffer
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetParm( )** function gets the current parameter values of the line device. The application can retrieve only one parameter at a time.

The parameter values are set using the **cc_SetParm( )** function or during the initialization of the MAKECALL Block. See the **cc_SetParm( )** function description and *Section 8.3.2. MAKECALL Block Initialization and Settings* for more information.

Parameter	Description
linedev:	The line device handle.
parm_id:	The specified parameter ID. Definitions and return values are listed in *Table 21*.
valuep:	Pointer to the address of the buffer in which the requested information will be stored.

The following table lists the **cc_GetParm( )** function parameter ID definitions.

Table 21. cc_GetParm( ) Parameter ID Definitions

Define	Description	Possible return values
BC_XFER_CAP	Bearer channel, information transfer capability	BEAR_CAP_SPEECH - speech BEAR_CAP_UNREST_DIG - unrestricted data BEAR_REST_DIG - restricted data
BC_XFER_MODE	Bearer channel, Information Transfer Mode	ISDN_ITM_CIRCUIT - circuit switch
BC_XFER_RATE	Bearer channel, Information Transfer Rate	BEAR_RATE_64KBPS - 64K bps transfer rate
USRINFO_LAYER1_ PROTOCOL	Layer 1 protocol to use on bearer channel	ISDN_UIL1_CCITTV.110 - CCITT standardized rate adaptation ISDN_UIL1_G711uLAW - Recommendation G.117 u-Law ISDN_UIL1_G711ALAW - Recommendation G.117 a-Law ISDN_UIL1_G721ADCPM - Recommendation G.721 32 kbits/s ADCPM and Recommendation I.460 ISDN_UIL1_G722G725 - Recommendation G.722 and G.725 - 7kHz audio ISDN_UIL1_H261 - Recommendation H.261 - 384 kbits/s video ISDN_UIL1_NONCCITT - Non-CCITT standardized rate adaptation ISDN_UIL1_CCITTV120 - CCITT standardized rate adaptation V.120 ISDN_UIL1_CCITTX31 - CCITT standardized rate adaptation X.31 HDLC

Define	Description	Possible return values
USR_RATE	User rate to use on bearer channel (layer 1 rate)	ISDN_UR_EINI460 - determined by E bits in I.460 ISDN_UR_56000 - 56 kbits ISDN_UR_64000 - 64 kbits ISDN_UR_134 - 134.5 bits, X.1 ISDN_UR_12000 - 12 kbits, V.6
CALLED_NUM_TYPE	Called number type	EN_BLOC_NUMBER - number is sent en-block (in whole, not overlap sending) INTL_NUMBER - international number for international call. (Verify availability with service provider.) NAT_NUMBER - national number for call within national numbering plan (accepted by most networks) LOC_NUMBER - subscriber number for a local call. (Verify availability with service provider.) OVERLAP_NUMBER - overlap sending; number is not sent in whole (not available on all networks)
CALLED_NUM_PLAN	Called number plan	UNKNOWN_NUMB_PLAN - unknown number plan ISDN_NUMB_PLAN - ISDN/telephony (E.164/E.163) (accepted by most networks) TELEPHONY_NUMB_PLAN - telephony numbering plan PRIVATE_NUMB_PLAN - private numbering plan
CALLING_NUM_TYPE	Calling number type	Same values as CALLED_NUM_TYPE

Define	Description	Possible return values
CALLING_NUM_PLAN	Calling number plan	Same values as CALLED_NUM_PLAN
CALLING_ PRESENTATION	Calling presentation indicator	PRESENTATION_ALLOWED - allows the display of the calling number at the remote end
CALLING_SCREENING	Calling screening indicator field	USER_PROVIDED - user provided, not screened (passes through)
RECEIVE_INFO_BUF	Multiple IE buffer – the number of messages that can be stored in the information queue.	Buffer number (any number in the range of 1 to MAX_RECEIVE_INFO_BUF) The **cc_GetParm( )** function returns the buffer number.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   CC_PARM   value;

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   if ( cc_GetParm(devhdl, BC_XFER_RATE, &value) < 0 )
        procdevfail(devhdl);
   else
```

```
        printf("Parameter BC_XFER_RATE has value: 0x%x\n",value);

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        procdevfail(devhdl);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);


}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetParm( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

See Also

- **cc_GetParmEx( )**
- **cc_SetParm( )**

Name:	int cc_GetParmEx(linedev, parm_id, valuep)	
Inputs:	LINEDEV linedev	• line device handle
	int parm_id	• parameter identifier
	PARM_INFO *valuep	• pointer to buffer containing the variable data
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetParmEx( )** function is an extension of the **cc_GetParm( )** function that allows the application to retrieve parameters containing variable data passed from the firmware.

Parameter	Description
linedev:	The line device handle.
parm_id:	The specified parameter ID. The **cc_GetParmEx( )** function can be used to retrieve all the parameters listed in *Table 21* in the **cc_GetParm( )** function description. In addition, for BRI/SC only, the **cc_GetParmEx( )** function can be used to retrieve the parameter values listed in *Table 22*.
valuep:	The address of the buffer in which the requested information will be stored. The PARM_INFO data structure contains he retrieved information. See *Section 6.10. PARM_INFO* for a description of the data structure.

The following table lists the **cc_GetParmEx( )** function parameter ID definitions.

Table 22. cc_GetParmEx( ) Parameter ID Definitions

Define	Description	Return values
DIRECTORY_NUMBER	Directory Number	String of length **parmdatalen**
SPID_NUMBER	Service Provider Identifier	String of length **parmdatalen**
SUBADDR_NUMBER	Subaddress Number	String of length **parmdatalen**

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV    devhdl    = 0;
   CRN        crn = 0;
   char       *devname = "dtiB1T1";
   PARM_INFO  parminfo, parm_ret_value;
   char databuffer[256];

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

  /* initialize PARM_INFO structure */
   parminfo.parmdatalen = strlen(DN);
   strcpy (parminfo.parmdata, DN); /* directory number */

  /* Specify the Directory Number */
   if cc_SetParmEx(ldev, DIRECTORY_NUMBER, &parminfo) < 0)
   {
      printf("Error in cc_SetParmEx(): %d\n", cc_CauseValue(ldev);
   }

   /* Get the Directory Number */
   if ( cc_GetParmEx(devhdl, DIRECTORY_NUMBER, &parm_ret_value) < 0 )
```

```
        procdevfail(devhdl);
    else {
        strcpy(databuffer, parm_ret_value.parmdata);
        printf("Parameter DIRECTORY_NUMBER has value: %s\n", databuffer);
    }

    if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

    if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .

    if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

    if ( cc_ReleaseCall(crn) < 0 )
        procdevfail(devhdl);

    if ( cc_Close( devhdl ) < 0 )
       printf("Error closing device, errno = %d\n", errno);


}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetParmEx( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

See Also

- **cc_GetParm( )**
- **cc_SetParmEx( )**

Name:	int cc_GetSAPI(sapip, evtdatap)	
Inputs:	char *sapip	• pointer to service access point ID buffer
	void *evtdatap	• pointer to an event block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC	

■ Description

The **cc_GetSAPI( )** function retrieves the service access point ID (SAPI) associated with the CCEV_D_CHAN_STATUS event received from the event queue.

Parameter	Description
sapip:	The address of the location where the output service access point ID is returned.
evtdatap:	Pointer to the structure containing the event data. The pointer value is acquired through the Dialogic Standard Runtime Library (SRL) function **sr_getevtdatap( )**.

■ Cautions

The **cc_GetSAPI( )** function applies only to BRI protocols. (This function is not supported for the BRI/2 board.)

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

long EventHandler(event)
{
   int            rc;
```

```
   L2_BLK         frame;
   unsigned int   resultValue;
   unsigned char  sapi, ces;
   int            device;
   void           *datap;

   device = sr_getevtdev();
   datap  = sr_getevtdatap();

   ...

   switch(event)
   {
      case CCEV_D_CHAN_STATUS:

         cc_GetSapi(&sapi, datap);
         cc_GetCes(&ces, datap);

         resultValue = cc_ResultValue(datap);

         switch(resultValue & ~(ERR_ISDN_FW))
         {
            case E_LINKUP:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_UP;
               break;

            case E_LINKDOWN:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_DOWN;
               break;

            case E_LINKDISABLED:
               DataLinkState[SAPI_ID][CES_ID] = DATA_LINK_DISABLED;
               break;

            default:
               printf("Got a bad result value (0x%X)\n", resultValue);
         }
         break;

      case CCEV_L2FRAME:
         if(rc = cc_GetFrame (dev, &frame) == 0)
         {
            sapi = frame.sapi;
            ces  = frame.ces;
            printf("Got a frame of length=%d for Sapi=%d Ces=%d\n", frame.length, sapi,
ces);
         }
         else
            printf("cc_GetFrame failed!\n");
         break;

      ...
   }

   return 0;
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function

can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetSAPI( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISNULLPTR	Null pointer error

See Also

- **cc_GetCES( )**
- **cc_GetDLinkState( )**

Name:	int cc_GetSigInfo(valuep, info_id, evtdatap)	
Inputs:	char *valuep	• pointer to information buffer
	int info_id	• signal information identifier
	void *evtdatap	• pointer to an event block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetSigInfo( )** function gets the signaling information of an incoming message. The library uses **evtdatap** to retrieve the associated signaling information elements (IEs) and puts the information in the queue.

In order to use the **cc_GetSigInfo( )** function for a channel, the application needs to specify the size of the queue by calling the **cc_SetParm( )** function and setting the RECEIVE_INFO_BUF to the desired size.

Parameter	Description
valuep:	Points to the buffer where the call information will be stored.
info_id:	Specifies the type of signaling information to be retrieved (see *Table 23* below).
evtdatap:	Points to the event data block. This pointer may be acquired by using **sr_getevtdatap( )**, an element of the Dialogic Standard Runtime Library (SRL).

The following table provides definitions of possible **info_id** parameters.

Table 23. cc_GetSigInfo( ) Info_ID Definitions

Info_ID	Definition
U_IES	Information elements (IEs) in CCITT format. The **cc_GetSigInfo( )** function retrieves all unprocessed IEs in CCITT format. Be sure to allocate enough memory (up to 256 bytes) to hold the retrieved information elements. The IEs are returned as raw data and must be parsed and interpreted by the application. Use IE_BLK to retrieve the unprocessed IEs. For a description of the IE_BLK data structure, see *Section 6.6. IE_BLK*. **NOTE:** Information Elements (IEs) that are specific to the DPNSS protocol are described in *Appendix C*.
UUI	User-to-user information - the data returned is application-dependent and is retrieved using the USRINFO_ELEM data structure. For a description of the return format for UUI, see *Section 6.16. USRINFO_ELEM*.

■ Cautions

- Make sure the size of the information buffer pointed to by the **valuep** parameter is large enough to hold the complete set of information elements requested by the **info_id** parameter.
- To use the **cc_GetSigInfo( )** function for a channel, the application must specify the size of the queue by calling the **cc_SetParm( )** function and setting the RECEIVE_INFO_BUF to the desired size. Failure to set the size of RECEIVE_INFO_BUF will result in an error.
- This function is not supported for the BRI/2 board.
- The CCEV_NOFACILITYBUF event will be received by the application for every incoming ISDN message that contains the Network Facility IE. The event is received because the first four IEs are stored in the ISDN library and the remaining ones are discarded. The CCEV_NOFACILITYBUF event can be ignored. The IE can be retrieved using the **cc_GetCallInfo(U_IES)** function or the **cc_GetInfoElem( )** function. The ability to retrieve just the

Network Facility IE using the **cc_GetCallInfo( )** function is no longer supported.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

#define MAX_QUEUE_SIZE    20


void main()
{
  LINEDEV ldev;
  char *devname = "dtiB1T1";

  .
  .
  .

/*get line device handler */

if (cc_Open(&ldev,devname,0)<0)
  {
    printf("ERROR opening device : errno = %d\n", errno);
    exit(1);
  }

/* using cc_SetParm to set size of the buffer queue for cc_GetSigInfo */
if (cc_SetParm (ldev, RECEIVE_INFO_BUF, MAX_QUEUE_SIZE) < 0)
  {
     procdevfail (ldev);
    exit (1);
  }

  .
  .
  .

/* Retrieve events from SRL */
FOREVER
while (1)
  {
    sr_waitevt(-1);
  }

  exit(0);
}


int evt_hdlr()
{
  LINEDEV ldev = sr_getevtdev();
  unsigned long *ev_datap = (unsigned long *)sr_getevtdatap();
  int len = sr_getevtlen();
  IE_BLK ie_blk;
```

```
  .
  .
  .

  switch(sr_getevttype()){

  .
  .
  .

  case CCEV_CALLINFO:
        /* retrieve signaling information from queue */
        if ( cc_GetSigInfo(&ie_blk, U_IES, ev_datap) <0)
           {
              /* failed, get failure reason */

              procdevfail (ldev);
           }
        else
          {
              /* succeeded, process signaling information */

              .
              .
              .
           }
        break;

  }
    .
    .
    .

}

int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_GetSigInfo( )** function include the following:

Error Code	Description
E_ISMISGSI | ERR_ISDN_LIB	The **cc_GetSigInfo( )** function is not supported because the application did not use **cc_SetParm( )** to allocate the signaling queue.
E_ISLSIEBUF | ERR_ISDN_LIB	The signaling information queue was too small to accommodate the incoming data. The signaling information associated with this event is lost. The application needs to increase the buffer size so that the queue is large enough to hold the information string.

See Also

- **cc_GetCallInfo( )**
- **cc_SetParm( )**

Name:	int cc_GetUsrAttr(linedev, usr_attr)	
Inputs:	LINEDEV linedev	• line device handle
	long *usr_attr	• user attribute information
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetUsrAttr( )** function gets the established attribute for the line device. The attributes are user defined and are established using the **cc_SetUsrAttr( )** function. The user attribute value can be a memory pointer used to identify a board and a channel on a board, or a pointer to a user-defined structure such as the current state or the active call reference number.

Parameter	Description
linedev:	The line device handle.
usr_attr:	The location where the returned user attribute will be stored.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

main()
{
   int chan = 1;
   char devname[16];
   .
   .
   .
```

```
   .
   if ( sr_enbhdlr( devhdl,CCEV_DISCONNECTED,discCallHdlr ) < 0 )
   {
           printf( "dtiEnable for DISCONNECT failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
           return( 1 );
   }

   printf(devname,"dtiB1T%d",chan);
   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( cc_SetUsrAttr(devhdl,chan))
        procdevfail(devhdl);
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitcvt(-1);
   }

   cc_Close(devhdl);

}

/*************************************************/
/* discCallHdlr - disconnect the active call       */
/*************************************************/
int discCallHdlr( )
{
    int devindx;
    int dev;
    int len;
    void *datap;
    CRN crn;
    long chan;

    dev = sr_getevtdev();
    len = sr_getevtlen();
    datap = sr_getevtdatap();


    cc_GetCRN(&crn, datap);
    if ( cc_GetUsrAttr(dev,&chan))
        procdevfail(devhdl);
    else
        printf("Call disconnected at chan %ld\n",chan);

    if ( cc_DropCall( crn,NORMAL_CLEARING,EV_ASYNC ) < 0 )
        callfail(crn);

    return( 0 );
}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
```

```
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_GetUsrAttr( )** function indicates that the function reference (the device handle) is not valid for the function call.

■ See Also

- **cc_SetUsrAttr( )**

Name:	int cc_GetVer(linedev, majorp, minorp)	
Inputs:	LINEDEV linedev	• line device handle
	int *majorp	• pointer to major number
	int *minorp	• pointer to minor number
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_GetVer( )** function retrieves the firmware version number associated with the specified line device handle. The firmware version number consists of two parts: the major version number and the minor version number.

Parameter	Description
linedev:	The specified line device handle.
majorp:	Pointer to the space prepared to receive the major firmware version number.
minorp:	Pointer to the space prepared to receive the minor firmware version number.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
  short devhdl;  /* device handle for D channel */
  int majnum;    /* major version number */
```

```
  int minnum;    /* minor version number */
  .
  .
  .
  if ( cc_Open(devhdl,"dtiB1", 0 ) < 0 )
     printf("Error opening device: errno = %d\n", errno);
   /*
    * Use cc_GetVer() to get the version values.
    */
  if ( cc_GetVer(devhdl,&majnum,&minnum) < 0 )
        procdevfail(devhdl);
  else
     printf("Major version number 0x%x, minor version number 0x%x\n"
                    ,majnum,minnum);

  /* continue the program. */
  .
  .
  .
  .
  .
  if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_GetVer( )** function indicates that the function reference (the device handle) is not valid for the function call.

■ See Also

None

Name: int cc_HoldAck(crn)
Inputs: CRN crn • call reference number
Returns: 0 on success
<0 on failure
Includes: cclib.h
Category: Hold and Retrieve
Mode: synchronous
Technology: BRI/2; BRI/SC; PRI (DPNSS and Q.SIG only)

■ Description

The **cc_HoldAck( )** function allows the application to accept a hold request from remote equipment. This function is called only after the call is in the Connected state and after the CCEV_HOLDCALL event is received. See *Section 7.1. Event Categories* for information on CCEV_HOLDCALL.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.

■ Cautions

A call must be in the Connected state and the CCEV_HOLDCALL event must be received before the **cc_HoldAck( )** function is invoked or the function will fail.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

main()
{
   char *devname = “dtiB1T1”;
   LINEDEV ldev;
   .
   .
   .
   .
if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
```

```
      exit(1);
   }
   if ( sr_enbhdlr( ldev,CCEV_HOLDCALL,HoldCallHdlr ) < 0 )
   {
          printf( "dtiEnable for HoldCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }

   cc_Close(ldev);

}

/*************************************************/
/* HoldCallHdlr - accept the hold call request     */
/*************************************************/
int HoldCallHdlr( )
{
    CRN crn_buf;
    int ldev = sr_getevtdev();
    int len = sr_getevtlen();
    vod *ev_datap = sr_getevtdatap();


    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_HoldAck( crn_buf ) < 0 )
         procdevfail(ldev);

    return( 0 );
}



int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_HoldAck( )** function include the following:

Error Code	Description
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

See Also

- **cc_RetrieveCall( )**

Name:	int cc_HoldCall(crn, mode)	
Inputs:	CRN crn	• call reference number
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success	
	<0 on failure	
Includes:	cclib.h	
Category:	Hold and Retrieve	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (DPNSS and Q.SIG only)	

■ Description

The **cc_HoldCall( )** function allows the application to place an active call on hold. For PRI protocols and for BRI Network-side, the call must be in the Connected state to be put on hold. For BRI User-side, the call can be put on hold any time after the CCEV_PROCEEDING message is received.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
mode:	Specifies synchronous (EV_SYNC) or asynchronous (EV_ASYNC) mode.

■ Termination Events

- CCEV_HOLDACK - indicates that the call has been placed on hold.
- CCEV_HOLDREJ - indicates that the hold request was rejected by the firmware or remote equipment.

In synchronous mode, the CCEV_TASKFAIL event is returned if the function fails.

■ Cautions

For PRI protocols and BRI Network-side, the **cc_HoldCall( )** function is valid only when the call is in the Connected state. For BRI User-side, the function can be called any time after the CCEV_PROCEEDING message is received.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <memory.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

void main()
{
   LINEDEV   ldev;
   CRN       crn_buf = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(ldev, &crn_buf, NULL, -1, EV_SYNC) < 0 )
     {
        procdevfail(ldev);
          .
     }

   if ( cc_AnswerCall(crn_buf, 0, EV_SYNC) < 0 )
     {
         procdevfail(ldev);

        .
     }
        .
        .

   if ( cc_HoldCall(crn_buf, EV_SYNC) < 0 )
     {
     procdevfail(ldev);
    .
     }
    .
    .
   if ( cc_RetrieveCall(crn_buf, EV_SYNC) < 0 )
     {
     procdevfail(ldev);
    .
     }
    .
    .
      /* Drop the call */
   if ( cc_DropCall(crn_buf, NORMAL_CLEARING, EV_SYNC) < 0 )
     {
     procdevfail(ldev);
          .
     }

   if ( cc_ReleaseCall(crn_buf) < 0 )
     {
     procdevfail(ldev);
          .
```

```
    }

   /* Close the device */
   if ( cc_Close( ldev ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}


int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_HoldCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ABORTED	Task aborted by the **cc_Restart( )** function
ERR_ISDN_LIB | E_BADSTATE	The **cc_Restart( )** function is in progress
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

See Also

- **cc_RetrieveCall( )**

Name:	int cc_HoldRej(crn, cause)	
Inputs:	CRN crn	• call reference number
	int cause	• standard ISDN Network error code
Returns:	0 on success	
	<0 on failure	
Includes:	cclib.h	
Category:	Hold and Retrieve	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (DPNSS and Q.SIG only)	

■ Description

The **cc_HoldRej( )** function allows the application to reject a hold request from remote equipment. This function is called only after the call is in the Connected state and after the CCEV_HOLDCALL event is received. See *Section 7.1. Event Categories* for information on CCEV_HOLDCALL.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
cause:	A standard ISDN Network cause/error code is returned indicating the reason the hold request was rejected. Possible causes include TEMPORARY_FAILURE (Cause 41), NETWORK_OUT_OF_ORDER (Cause 38), and NETWORK_CONGESTION (Cause 42). For a complete list of ISDN Network cause/error codes, see *Section 7.2.2. Cause/Error Codes from the ISDN Network*.

■ Cautions

- The **cc_HoldRej( )** function should be called only after the call is in the Connected state and after receiving the CCEV_HOLDCALL event or the function will fail.
- Not all ISDN Network cause/error codes are universally supported across switch types. Before using a particular cause code, compare its validity with the appropriate switch vendor specifications.

Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

main()
{

   char *devname = “dtiB1T1”;
   LINEDEV ldev;
   .
   .
   .
   .

   if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }
   if ( sr_enbhdlr( ldev,CCEV_HOLDCALL,HoldCallHdlr ) < 0 )
   {
          printf( "dtiEnable for HoldCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }

   cc_Close(ldev);

}

/***********************************************/
/* HoldCallHdlr - reject the hold call request     */
/***********************************************/
int HoldCallHdlr( )
{
    CRN crn_buf;
    LINEDEV ldev = sr_getevtdev();
    int len = sr_getevtlen();
    void *ev_datap = sr_getevtdatap();


    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_HoldRej( crn_buf,TEMPORARY_FAILURE ) < 0 )
         procdevfail(ldev);

    return( 0 );
}
```

```
int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_HoldRej( )** function include the following:

Error Code	Description
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

See Also

- **cc_RetrieveCall( )**

Name:	int cc_MakeCall(linedev, crnp, numberstr, makecallp, timeout, mode)	
Inputs:	LINEDEV linedev	• device handle
	CRN *crnp	• pointer to call reference number
	char *numberstr	• destination phone number
	MAKECALL_BLK *makecallp	• pointer to outbound call information
	int timeout	• time interval
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Call control	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_MakeCall( )** function allows the application to request a connection to make an outgoing call on the specified line device or, for the call waiting feature, on the specified board device (BRI and Windows only). When this function is issued asynchronously, a call reference number (CRN) is assigned and returned immediately if the function is successful. If the function is issued synchronously, the CRN will be available at the successful completion of the function. All subsequent communications between the application and the driver regarding the call use the CRN as a reference.

Parameter	Description
linedev:	The line device handle or, for applications using the call waiting feature, the board device handle (BRI and Windows only).
crnp:	Pointer to the buffer where the call reference number will be stored.
numberstr:	The destination (called party 's) telephone number string. The maximum length is 32 digits.
makecallp:	The pointer to the MAKECALL_BLK structure, which is a list of parameters used to specify the outgoing call. See *Section 6.8. MAKECALL_BLK* for a description of the MAKECALL_BLK structure and for definitions and possible values for the parameters contained in the structure. For information on initializing the MAKECALL_BLK structure, see *Section 6.8.1. MAKECALL_BLK Initialization*.
timeout:	The amount of time (in seconds) during which the call must be established. (Used in synchronous mode only.) If the timeout value expires before the remote end answers the call, then the **cc_MakeCall( )** function returns -1. Setting the **timeout** parameter to 0 causes this parameter to be ignored.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Events

- CCEV_CONNECTED - indicates that a CONNECT message has been received by the network.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

- Every field of the MAKECALL_BLK structure must be filled. Use the value ISDN_NOT_USED if the field does not apply to the specific ISDN provisioning or service. If a string field is not used, set the field to NULL.

- BRI only: Applications using the call waiting feature must use the **cc_MakeCall( )** function in asynchronous mode.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void build_makecall_blk( MAKECALL_BLK *makecall_blk )

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   MAKECALL_BLK makecall_blk;

   if ( cc_Open(&devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   /* initialize the MAKECALL Block */
   build_makecall_blk(&makecall_blk)
   if ( cc_MakeCall(devhdl, &crn, "9933000", &makecall_blk, 30, EV_SYNC) < 0 )
        procdevfail(devhdl);
        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);
}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

```
void build_makecall_blk( MAKECALL_BLK *makecall_blk )
{
   memset(makecall_blk,0xff,sizeof(MAKECALL_BLK));
   makecall_blk->isdn.BC_xfer_cap = BEAR_CAP_SPEECH;
   makecall_blk->isdn.BC_xfer_mode = ISDN_ITM_CIRCUIT;
   makecall_blk->isdn.BC_xfer_rate = BEAR_RATE_64KBPS;
   makecall_blk->isdn.facility_coding_value = ISDN_CPN;
   makecall_blk->isdn.destination_number_type = NAT_NUMBER;
   makecall_blk->isdn.destination_number_plan = ISDN_NUMB_PLAN;
   makecall_blk->isdn.origination_number_type = ISDN_NOTUSED;
   makecall_blk->isdn.origination_number_plan = ISDN_NOTUSED;
   makecall_blk->isdn.origination_phone_number[0] = '\0';
   makecall_blk->isdn.facility_feature_service = ISDN_SERVICE;
   makecall_blk->isdn.usrinfo_layer1_protocol = ISDN_UIL1_G711ULAW;
   makecall_blk->isdn.usr_rate = ISDN_NOTUSED;
   makecall_blk->isdn.usrinfo_bufp = NULL;
   makecall_blk->isdn.nsfc_bufp = NULL;
   makecall_blk->isdn.destination_sub_number_type = OSI_SUB_ADDR;
   makecall_blk->isdn.destination_sub_phone_number[0] = 1;
   makecall_blk->isdn.destination_sub_phone_number[1] = 2;
   makecall_blk->isdn.destination_sub_phone_number[2] = 3;
   makecall_blk->isdn.destination_sub_phone_number[3] = '\0';
   makecall_blk->isdn.u.bri.channel_id.channel = NO_BCHAN;
   makecall_blk->isdn.u.bri.channel_id.channel_mode = PREFFERED;

}
```

Call Waiting Example (Windows Only):

```
#include <windows.h>
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/** Function prototypes **/
void build_makecall_blk( MAKECALL_BLK *makecall_blk );
int callfail(CRN crn);
int procdevfail(LINEDEV handle);

void main( )
{
   CRN crn = 0;
   CHAN_ID chanId;
   LINEDEV devhdl = 0;
   char *devname = "dtiB1";
   MAKECALL_BLK makecall_blk;

   if ( cc_Open(&devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device, errno = %d\n", errno);
      exit(1);
   }

   /* initialize the MAKECALL Block */
   build_makecall_blk(&makecall_blk);

   if ( cc_MakeCall(devhdl,&crn,''9933000'',&makecall_blk,30,EV_SYNC) < 0 )
      procdevfail(devhdl);

   cc_GetChanId(&crn, &chanId);
```

```
   printf("The call is conducted on channel %d\n", chanId.channel);

   .
   .
   .

   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
      callfail(crn);
   if ( cc_ReleaseCall(crn) < 0 )
      callfail(crn);
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);
}

int callfail(CRN crn)
{
   LINEDEV ld;

   cc_CRN2LineDev(crn,&ld);
   procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
   int reason;
   char *msg;

   reason = cc_CauseValue(handle);
   cc_ResultMsg(handle,reason,&msg);
   printf("reason = %x - %s\n",reason,msg);
}

void build_makecall_blk( MAKECALL_BLK *makecall_blk )
{
   memset(makecall_blk,0xff,sizeof(MAKECALL_BLK));
   makecall_blk->isdn.BC_xfer_cap = BEAR_CAP_SPEECH;
   makecall_blk->isdn.BC_xfer_mode = ISDN_ITM_CIRCUIT;
   makecall_blk->isdn.BC_xfer_rate = BEAR_RATE_64KBPS;
   makecall_blk->isdn.facility_coding_value = ISDN_CPN;
   makecall_blk->isdn.destination_number_type = NAT_NUMBER;
   makecall_blk->isdn.destination_number_plan = ISDN_NUMB_PLAN;
   makecall_blk->isdn.origination_number_type = ISDN_NOTUSED;
   makecall_blk->isdn.origination_number_plan = ISDN_NOTUSED;
   makecall_blk->isdn.origination_phone_number[0] = '\0';
   makecall_blk->isdn.facility_feature_service = ISDN_SERVICE;
   makecall_blk->isdn.usrinfo_layer1_protocol = ISDN_UIL1_G711ULAW;
   makecall_blk->isdn.usr_rate = ISDN_NOTUSED;
   makecall_blk->isdn.usrinfo_bufp = NULL;
   makecall_blk->isdn.nsfc_bufp = NULL;
   makecall_blk->isdn.destination_sub_number_type = OSI_SUB_ADDR;
   makecall_blk->isdn.destination_sub_phone_number[0] = 1;
   makecall_blk->isdn.destination_sub_phone_number[1] = 2;
   makecall_blk->isdn.destination_sub_phone_number[2] = 3;
   makecall_blk->isdn.destination_sub_phone_number[3] = '\0';
   makecall_blk->isdn.channel_id.channel = NO_BCHAN;
   makecall_blk->isdn.channel_id.channel_mode = PREFFERED;
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_MakeCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISINVNETWORK	Invalid network type
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier

■ See Also

- **cc_DropCall( )**
- **cc_ReleaseCall( )**
- **cc_SetInfoElem( )**

Name:	int cc_Open(linedevp, devicename, rfu)
Inputs:	LINEDEV *linedevp • pointer to the buffer storing the line device handle
	char *devicename • name of the device
	int rfu • reserved for future use
Returns:	0 on success
	<0 on failure
Includes:	cclib.h
Category:	Call control
Mode:	synchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_Open( )** function opens a device and returns a unique line device handle to identify the physical device that carries the call. The **cc_Open( )** function is used to open both network board and channel (that is, time slot) devices. Once a device is opened, all subsequent references to the opened device must be made using the line device number that is returned to and placed in **linedevp**.

NOTE: A channel cannot be opened twice in the same process.

Parameter	Description
linedevp:	A unique number placed in the location pointed to by **linedev**. The number identifies the specific device opened by this function.
devicename:	The pointer to the ASCII string that defines the device or devices that will be associated with the returned line device. The format for **devicename** is as follows: PRI: • dtiBx for board • dtiBxTy for time slot BRI: • briSx for board • briSxTy for time slot
rfu:	Reserved for future use. Set **rfu** to 0.

Each PRI structure is composed of one D channel and 23 (T1) or 30 (E1) B (bearer) channels. A PRI board device, such as dtiB1, is defined as a station and controls the D channel. A PRI time slot device, such dtiB1T1, is defined as a bearer channel under a station.

Each BRI structure is composed of one D channel and two B (bearer) channels. A BRI board device, such as briS1, is defined as a station and controls the D-channel the same way as a PRI board device. A BRI time slot device, such as briS1T1, is defined as a bearer channel under a station and is handled the same way as a PRI line device.

■ Cautions

Do not open a D or B channel more than once from the same process, or unpredictable results could occur.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “briS1T1”;

   if ( cc_Open( &devhdl, devname, 0) < 0 )
   {
        printf(“Error opening device: errno = %d\n”, errno);
        exit(1);
   }

   printf(“Waiting for call\n”);
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);

}
```

```
int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, check *errno.h* to retrieve the reason for the failure.

Typically, a < 0 return code for the **cc_Open( )** function indicates that the function reference (the device name) is not valid for the function call.

■ See Also

- **cc_Close( )**

Name:	int cc_PlayTone(devHdl, pToneParm, tptp, mode)	
Inputs:	int devHdl	• channel device handle
	toneParm *pToneParm	• pointer to the toneParm structure
	DV_TPT *tptp	• pointer to the DV_TPT structure
	int mode	• asynchronous/synchronous
Returns:	0 on success	
	<0 on failure	
Includes:	cclib.h	
Category:	Global Tone Generation	
Mode:	asynchronous or synchronous	
Technology:	BRI/SC	

■ Description

The **cc_PlayTone( )** function allows the application to play a user-defined tone. The tone's attributes are defined in the structure toneParm, which is pointed to by the parameter **pToneParm**.

Parameter	Description
devHdl:	The channel device handle of the channel on which the tone is to be played.
pToneParm:	The pointer to the tone parameter structure. See *Section 6.14. ToneParm* for a description of the toneParm structure.
tptp:	Points to the DV_TPT data structure, which specifies the terminating condition (DX_MAXTIME) for the function. DX_MAXTIME is the only allowed terminating condition; no other termination conditions are supported. See the *Voice Software Reference – Features Guide* for the appropriate operating system for a description of the DV_TPT data structure.
mode:	Specifies whether to run the function asynchronously (EV_ASYNC) or synchronously (EV_SYNC).

■ Termination Events

- CCEV_PLAYTONE - indicates that the tone was successfully played.
- CCEV_PLAYTONEFAIL - indicates that the request to play a tone failed.

Use the SRL Event Management functions to handle the termination event.

■ Cautions

- The channel must be in the Idle state when calling the **cc_PlayTone( )** function.
- The **cc_PlayTone( )** command is a host tone command that allows the application to play a user-defined tone. This command cannot be used to set or play the firmware-applied call progress tones. The call progress tones and user-defined tones operate independently, except that when the firmware is playing a tone, the application may not play a tone on the same channel at the same time. For information on changing the firmware-applied call progress tones, see the **cc_ToneRedefine( )** function description.
- This function is not supported for the BRI/2 board or for PRI protocols.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <cclib.h>
#include <dxxxlib.h>

#define TID_1   101
main()
{
   toneParm     ToneParm;
   DV_TPT       tpt;
   int          devHdl;
/*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( cc_Open(&devHdl, "briS1T1", 0) ) < 0 ) {
      printf( "Error opening device : errno < %d\n", errno );
      exit( 1 );
   }

    ToneParm.freq1 = 330;
    ToneParm.freq2 = 460;
    ToneParm.amp1 = -10;
    ToneParm.amp2 = -10;
    ToneParm.toneOn1 = 400;
    ToneParm.toneOff1 = 0;
    ToneParm.duration = -1;

tpt.tp_type = IO_EOT;
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 6000;
tpt.tp_flags = TF_MAXTIME;

   /*
```

```
    * Play a Tone with its
    * Frequency1 = 330, Frequency2 = 460
    * amplitude at -10dB for both
    * frequencies and duration of infinity
 * This is a Panasonic Local Dial Tone.
    *
    */
    if (cc_PlayTone( devHdl, &ToneParm, &tpt, EV_SYNC ) == -1 ){
      printf( "Unable to Play the Tone\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",

      cc_CauseValue( devHdl ), cc_ResultMsg( devHdl ) );
      cc_close( devHdl);
      exit( 1 );
   }
   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */
   /*
    * Close the opened Voice Channel Device
    */
   if ( cc_Close( devHdl ) != 0 ) {
      printf( "Error closing devicd, errno= %d\n", errno );
   }
   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_PlayTone( )** function include the following:

Error Code	Description
ERR_TONEINVALIDMSG	Invalid message type
ERR_TONEBUSY	Busy executing previous command
ERR_TONECP	System error with CP
ERR_TONEDSP	System error with DSP
ERR_TONEFREQ1	Invalid value specified in parameter freq 1
ERR_TONEFREQ2	Invalid value specified in parameter freq 2
ERR_TONEAMP1	Invalid value specified in parameter amp1
ERR_TONEAMP2	Invalid value specified in parameter amp2

ERR_TONEON1	Invalid value specified in parameter toneOn1
ERR_TONEOFF1	Invalid value specified in parameter toneOff1
ERR_TONEDURATION	Invalid value specified in parameter duration
ERR_TONECHANNELID	Invalid channel ID

See Also

- **cc_ToneRedefine( )**
- **cc_StopTone( )**

Name:	int cc_ReleaseCall(crn)
Inputs:	CRN crn • call reference number
Returns:	0 on success < 0 on failure
Includes:	cclib.h
Category:	Call control
Mode:	synchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_ReleaseCall( )** function instructs the driver and firmware to release all internal resources for the specified call. An inbound call will be rejected after **cc_DropCall( )** is used. Every issue of **cc_DropCall( )** must be followed by **cc_ReleaseCall( )**.

When **cc_WaitCall( )** is used in synchronous mode and **cc_ReleaseCall( )** is issued subsequently, the next inbound call on the same channel will be pending until the **cc_WaitCall( )** function is issued again. If **cc_WaitCall( )** is used in asynchronous mode, the inbound call notification can be received immediately after **cc_ReleaseCall( )**.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.

■ Cautions

- Windows only: For new applications, it is recommended that the **cc_ReleaseCallEx( )** function be used instead of the **cc_ReleaseCall( )** function. Under load conditions, or if the remote end delays transmitting the RELEASE COMPLETE message, an application could experience a significant delay while the **cc_ReleaseCall( )** function unblocks and returns control to the application. This delay can be up to 8 seconds long if the RELEASE COMPLETE message is never returned. The **cc_ReleaseCall( )** function is supported only in synchronous mode; therefore, this problem occurs only in applications that use the asynchronous single-threaded programming model. In this case, when this blocking function is called within a handler processing the CCEV_DROPCALL event, it could create a

bottleneck for processing any other event and, thereby, could affect call handling performance.

- After a connection is terminated, the **cc_ReleaseCall( )** function should be called to release the call reference number (CRN). Failure to call this function may cause a blocking condition or a memory allocation error. Channels will remain allocated if call-related resources are not released.
- After the **cc_ReleaseCall( )** function is issued, the CRN is no longer valid.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call with NORMAL CLEARING cause value */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
```

```
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_ReleaseCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISNULLPTR	Null pointer error

■ See Also

- **cc_AnswerCall( )**
- **cc_MakeCall( )**
- **cc_WaitCall( )**
- **cc_DropCall( )**
- **cc_ReleaseCallEx( )**

Name:	int cc_ReleaseCallEx(crn, mode)
Inputs:	CRN crn • call reference number unsigned long mode • synchronous or asynchronous
Returns:	0 on success < 0 on failure
Includes:	cclib.h
Category:	Call control
Mode:	synchronous or asynchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_ReleaseCallEx( )** function instructs the driver and firmware to release all Dialogic ISDN resources for the specified call. Every **cc_DropCall( )** must be followed by **cc_ReleaseCallEx( )** or **cc_ReleaseCall( )** (see Note below). An inbound call will be rejected after **cc_DropCall( )** and prior to **cc_ReleaseCallEx( )**.

NOTE: Windows only: It is recommended that, for new applications, the **cc_ReleaseCallEx( )** function be used instead of the **cc_ReleaseCall( )** function. See the Cautions in the **cc_ReleaseCall( )** function description for more information.

Under PRI, the firmware sends the RELEASE message to the network automatically, by default. However, the host can be configured to control when to send the RELEASE message to the network by using a parameter configuration file set prior to download time. Unlike PRI, the BRI board passes this control to the host application by default. The host application then sends the RELEASE message through the **cc_ReleaseCallEx( )** function. See the Host-Controlled Disconnect Process scenario in *Appendix A - Call Control Scenarios* for more information on how to use this function.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Events

- CCEV_RELEASECALL - indicates that all Dialogic ISDN resources have been released for the call.
- CCEV_RELEASECALLFAIL – indicates that the **cc_ReleaseCallEx( )** function failed.

■ Cautions

The **cc_ReleaseCallEx( )** function is not supported for LINUX applications.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/* GLOBAL VARIABLES */
LINEDEV   ldev = 0;
CRN       crn = 0;
char     *devname = "dtiB1T1";  /* device name for ISDN board 1 timeslot 1 */

void main()
{
  int rc;
  int srlmode = SR_STASYNC;  /* mode for SRL initialization (disabling the internal SRL
event thread) */
   .
   .
   .
  /* open the ISDN line device */
  if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

 /* disable the internal SRL thread in order to use sr_waitevt() for event processing */

          if ( sr_setparm( SRL_DEVICE,SR_MODELTYPE,&srlmode) == -1 )
         {
            printf( "sr_setparm of SRL failed\n" );
          }

/* set the event handlers */
   sr_enbhdlr( EV_ANYDEV, EV_ANYEVT, (HDLR)defaultHandler);

   /* wait for an incoming call */
  if ((rc = cc_WaitCall(ldev, &crn, NULL, EV_ASYNC)) < 0)
  {
      rc = cc_CauseValue(ldev);
      printf("ERROR in cc_WaitCall(), error code = %x (%s) \n",
         rc, cc_ResultMsg(rc));
```

```
      exit(1);
  }
  else
  printf("Waiting for call...\n");
   .
   .
   .
  /* wait indefinitely for events */
  sr_waitevt(-1);

  /* continue with application */
        .
        .
        .
        .
} /* end main */

/*********************************************/
/* defaultHandler - default event handler    */
/*********************************************/
int defaultHandler()
{
   int devindx;
   int dev = sr_getevtdev();
   int len = sr_getevtlen();
   int typ = sr_getevttype();
   void *datap = sr_getevtdatap();

   /* process all incoming events in one switch statement */
   switch(typ)
   {
        .
        .
        .
      case CCEV_DROPCALL:
        printf("Asynchronous cc_DropCall() completed\n");
        /* release the call asynchronously so as not to block within an async handler */
        if (cc_ReleaseCallEx(crn, EV_ASYNC)) = = -1) {
           printf("Error in cc_ReleaseCall(): error code = %x\n", cc_CauseValue(ldev));
        }
        else printf("Asynchronous cc_ReleaseCall sent...waiting for completion event
CCEV_RELEASECALL...\n");

      /* the next case is the completion event for the asynchronous cc_ReleaseCall */
      case CCEV_RELEASECALL:
        printf("Asynchronous cc_ReleaseCall completed\n");
        break;

   } /* end switch */

} /* end defaultHandler() */
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_ReleaseCallEx( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_BADSTATE	Bad state
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

- **cc_DropCall( )**

Name:	int cc_ReqANI(crn, ani_buf, reqtype, mode)	
Inputs:	CRN crn	• call reference number
	char *ani_buf	• pointer to address of ANI buffer
	int reqtype	• type of information requested
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous or asynchronous	
Technology:	PRI (4ESS only)	

■ Description

The **cc_ReqANI( )** function returns the caller ID for Automatic Number Identification (ANI)-on-demand services. The caller ID is usually included in the ISDN setup message. However, if the caller ID does not exist and the serving network is AT&T, the driver will automatically request the caller ID from the network if the ANI-on-demand feature is enabled. The information is returned in a NULL terminated ASCII string.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
ani_buf:	The address of the buffer where ANI information is stored.
reqtype:	The type of information requested. Possible values are: • ISDN_CPN_PREF - calling party number preferred • ISDN_BN_PREF - billing number preferred • ISDN_CPN - calling party number only • ISDN_BN - billing number only • ISDN_CA_TSC - special uses
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

The **cc_ReqANI( )** function can operate as either a multitasking or non-multitasking function. It is a multitasking function when the caller number is offered upon request and the network provides this type of service (such as

AT&T's ANI-on-demand service). **cc_ReqANI( )** is a non-multitasking function when the calling party number is received or when the network does not offer an ANI-on-demand service. Thus, if ANI is already available, the function returns immediately because it does not have to instruct the interface device to query the switch.

In EV_ASYNC mode, the function will always return an event. In EV_SYNC mode, the function will return automatically with the ANI if one is available. Otherwise, the function will wait for completion of the ANI-on-demand request.

NOTE: If ANI is always available, use the **cc_GetANI( )** function, instead of the **cc_ReqANI( )** function, for a faster return.

■ Termination Events

- CCEV_REQANI - indicates that the ANI (caller ID) has been received from the network. This event occurs only when the ANI-on-demand feature is used.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

- Make sure the size of **ani_buf** is sufficient for the ANI string. Refer to the file *cclib.h* for the maximum allowable string.
- The ANI-on-demand feature is available only on the AT&T ISDN network.
- **cc_ReqANI( )** may not function in all service-provider environments. Check whether retrieving billing information is an option with the service provider.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   char      ani_buf[CC_ADDRSIZE];
```

```
   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   printf("Requesting ANI\n");
   if ( cc_ReqANI(crn, ani_buf, ISDN_CPN_PREF, EV_SYNC) < 0 )
        callfail(crn);
   else
        printf("cc_ReqANI succeeded: %s\n",ani_buf);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_ReqANI( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNULLPTR	Null pointer error
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

- **cc_GetANI( )**
- **cc_WaitCall( )**

Name:	int cc_Restart(linedev, mode)	
Inputs:	LINEDEV linedev	• device handle
	unsigned long mode	• asynchronous or synchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System control	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_Restart( )** function resets the channel to Null state. This function typically is used after the recovery of trunk errors or alarm conditions, or when the application needs to reset the channel to NULL state.

When the **cc_Restart( )** function is called, the following activities take place on the B channel specified in **linedev**. If the application is using the call waiting feature (BRI and Windows only) and a board device has been specified in **linedev**, the following activities will take place on all of the B channels associated with the board device. The activities take place in the order listed:

1. The active call is disconnected and all new incoming calls are blocked.
2. The call reference number and all call information is cleared.
3. When the function is returned, the channel is in blocked state. The application must reissue a new **cc_WaitCall( )** to accept a new call.

Parameter	Description
linedev:	Specifies the line device handle for the channel or, for applications using the call waiting feature, the board device handle (BRI and Windows only).
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode. **NOTE:** For synchronous applications, the **cc_Restart( )** function must be issued by the same process as the device controlling process.

■ Termination Event

- CCEV_RESTART - indicates that the Restart operation has been completed. The channel is in Null state.
- CCEV_RESTARTFAIL - indicates that the function call failed. The application can use **cc_ResultValue( )** after this event is received to verify the reason for failure.

■ Cautions

In asynchronous mode, the application cannot call any call state-related function prior to CCEV_RESTART.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <process.h>
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
#define ESC      27     /* ASCII ESC value */
#define SPACE    ' '    /* ASCII space value */

void WaitCallThread(void *argp);

main()
{
  short devhdl;     /* device handle for B channel */
  .
  .
  .
  /* open the channel 1 device */
  if ( cc_Open(&devhdl,"dtiB1T1", 0 ) < 0 )
     exit(1); /* or different actions */

  /* start a child thread */
  if ( _beginthread( WaitCallThread ,0,(void *)&devhdl) < 0 )
      exit(1);
  .
  .
  while(1)
  {
     int keyin; /* key stroke */

     printf("Press <SPACE> to make a call, or press <ESC> to quit\n");
     keyin = getch();
     if ( keyin == ESC )
         break;
     else if (keyin == SPACE)
         makecall(devhdl);  /* make a call */

     /* else repeat the loop */
  }
```

```
  .
  .
  cc_Close(devhdl);
}

/*****************************************************************
/* NAME: void WaitCallThread(devhdlptr)
/* DESCRIPTION: Waits for incoming call.
/* INPUT: devhdlptr - pointer to devhdl
/* RETURNS: none.
 *****************************************************************/
void WaitCallThread(void *devhdlptr)
{
  short devhdl = *devhdlptr;  /* device handle */
  CRN crn;                    /* call reference number */

  .
  .
  .
  while(1)
  {
     .
     .
     .
     /* wait for an incoming call */
     if ( cc_WaitCall(devhdl,&crn,NULL,-1,EV_SYNC) < 0 )
         procdevfail(devhdl);

     if ( cc_AnswerCall(crn,0,EV_SYNC) < 0 )
         callfail(crn);

         .
         .
         .
         .
  }
  .
  .
  .

}

/*
/* function to make an outbound call
 */
int makecall(short devhdl)
{
  CRN crn;                       /* call reference number */
  char *dialnum = “12019933000”; /* outgoing phone number */

     .
     .
     .
  /*
   *This cc_Restart() allows the B channel line device to
   *escape from cc_WaitCall(), or other multitasking
   *functions.
   */
  if ( cc_Restart(devhdl, EV_SYNC) < 0 )
       procdevfail(devhdl);

  /*
   * Now it’s safe to make a call.  To avoid contention,
   * a mechanism to prevent other multitasking functions
   * from being invoked again, such as cc_WaitCall(),
```

```
   * is recommended here.
   */
  if ( cc_MakeCall(devhdl, &crn, dialnum, NULL, 20, EV_SYNC)
           < 0 )
     procdevfail(devhdl);

     .
     .
     .
     .

}

/* function to process error conditions */
int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle, reason, &msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_Restart( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

None

Name:	int cc_ResultMsg(linedev, ResultCode, msg)	
Inputs:	LINEDEV linedev	• line device handle
	int ResultCode	• function return value
	char **msg	• pointer to the address where the result message is stored
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

cc_ResultMsg( ) is a convenience function that interprets the function return code. The **cc_ResultMsg( )** function retrieves either the cause value of an event from the **cc_CauseValue( )** function or the function return code from the **cc_ResultValue( )** function.

Parameter	Description
linedev:	The line device handle.
ResultCode:	The cause value of the event or return code (retrieved from **cc_ResultValue( )** or **cc_CauseValue( )**).
msg:	The pointer to the buffer address where the result message will be stored.

■ Caution

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
```

```
void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
         procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
         callfail(crn);
         .
         .
         .
         .
         .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
         callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
         callfail(crn);

   if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle, reason, &msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_ResultMsg( )** function indicates that the function reference (the line device number) is not valid for the function call.

See Also

- **cc_ResultValue( )**

Name:	int cc_ResultValue(evtdatap)
Inputs:	void *evtdatap • pointer to an event block
Returns:	0 on success < 0 on failure to retrieve the error/cause value > 0 error/cause value associated with event
Includes:	cclib.h
Category:	System tools
Mode:	synchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_ResultValue( )** function gets an error/cause code of an event. The code identifies the return value of a function or the cause of an event. Result values are located in *isdnerr.h* and *isdncmd.h*. Use **cc_ResultMsg( )** to return the text definition for the code.

NOTE: See *Section 7.2. Error Handling* for a listing of error/cause codes.

Parameter	Description
evtdatap:	Points to the event data block. This pointer may be acquired via **sr_getevtdatap( )**, an element of the Dialogic Standard Runtime Library (SRL).

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

main()
{
   .
   .
   .
```

```
   .
   if ( sr_enbhdlr(devhdl, CCEV_DISCONNECTED, discCallHdlr) < 0 )
   {
           printf( "dtiEnable for DISCONNECT failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
           return( 1 );
   }
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }

}

/************************************************/
/* discCallHdlr - disconnect the active call       */
/************************************************/
int discCallHdlr( )
{
    int devindx;
    int dev;
    int len;
    void *datap;
    CRN crn;
    int reason;

    dev = sr_getevtdev();
    len = sr_getevtlen();
    datap = sr_getevtdatap();

    cc_GetCRN (&crn, datap);

    reason = cc_ResultValue(datap);
    printf("Call disconnected reason = 0x%x\n",reason);

    if ( cc_DropCall(crn, NORMAL_CLEARING, EV_ASYNC) < 0 )
    {
        int lasterr = cc_CauseValue(dev);
        char *errmsg;
        if ( cc_ResultMsg(dev, lasterr, &errmsg) == 0 )
             printf( "\tcc_DropCall() Error: ( %x ) %s\n",lasterr,errmsg);
    }

    return( 0 );
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_ResultMsg( )** function to interpret the returned value. Error codes are defined in *Section 7.2. Error Handling* and in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_ResultValue( )** function indicates that the function reference (the event block pointer) is not valid for the function call.

See Also

- **cc_ResultMsg( )**

Name:	int cc_RetrieveAck(crn)
Inputs:	CRN crn • call reference number
Returns:	0 on success <0 on failure
Includes:	cclib.h
Category:	Hold and Retrieve
Mode:	synchronous
Technology:	BRI/2; BRI/SC; PRI (Q.SIG only)

■ Description

The **cc_RetrieveAck( )** function allows the application to accept a request to retrieve a call from hold from remote equipment. The call must be in the Hold state and the CCEV_RETRIEVECALL event must be received before the function is called.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.

■ Cautions

- The BRI protocols and the PRI Q.SIG protocol are the only protocols that allow the retrieval of a call in the Hold state to be accepted or rejected. The **cc_RetrieveAck( )** function cannot be used in the DPNSS protocol or any PRI protocol other than Q.SIG.
- The **cc_RetrieveAck( )** function is valid only after the call is in the Hold state and after the CCEV_RETRIEVECALL event is received.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

main()
{

   char *devname = "dtiB1T1";
   LINEDEV ldev;
```

```
   .
   .
   .
if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }
   if ( sr_enbhdlr( ldev,CCEV_HOLDCALL,HoldCallHdlr ) < 0 )
   {
          printf( "dtiEnable for HoldCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }
   if ( sr_enbhdlr( ldev,CCEV_RETRIEVECALL,RetrieveCallHdlr ) < 0 )
   {
          printf( "dtiEnable for RetrieveCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }

   cc_Close(ldev);

}

/*************************************************/
/* HoldCallHdlr - accept the hold call request     */
/*************************************************/
int HoldCallHdlr( )
{
    CRN crn_buf;
    LINEDEV ldev = sr_getevtdev();
    int len = sr_getevtlen();
    void *ev_datap = sr_getevtdatap();

    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_HoldAck( crn_buf ) < 0 )
         procdevfail(ldev);

    return( 0 );
}
/*********************************************************/
/* RetrieveCallHdlr - accept the retrieve call request     */
/*********************************************************/
int RetrieveCallHdlr( )
{
    CRN crn_buf;
    LINEDEV ldev = sr_getevtdev();
    int len = sr_getevtlen();
    void *ev_datap = sr_getevtdatap();

    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_RetrieveAck( crn_buf ) < 0 )
          procdevfail(ldev);
```

```
    return( 0 );
}


int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_RetrieveAck( )** function include the following:

Error Code	Description
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

■ See Also

- **cc_HoldCall( )**
- **cc_RetrieveCall( )**
- **cc_RetrieveRej( )**

Name:	int cc_RetrieveCall(crn, mode)	
Inputs:	CRN crn	• call reference number
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success	
	<0 on failure	
Includes:	cclib.h	
Category:	Hold and Retrieve	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (DPNSS and Q.SIG only)	

■ Description

The **cc_RetrieveCall( )** function allows the application to retrieve a call from the Hold state. The call must be in the Hold state to use this function.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
mode:	Specifies synchronous (EV_SYNC) or asynchronous (EV_ASYNC) mode.

■ Termination Events

- CCEV_RETRIEVEACK - indicates that the call has been retrieved from Hold state
- CCEV_RETRIEVEREJ - indicates that the retrieve request was rejected by the firmware or remote equipment

■ Cautions

The **cc_RetrieveCall( )** function can only be used when the call is in Hold state.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>
```

```
void main()
{
   LINEDEV   ldev;
   CRN       crn_buf = 0;
   char     *devname = "dtiB1T1";


   if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(ldev, &crn_buf, NULL, -1, EV_SYNC) < 0 )
     {
    procdevfail(ldev);
     .
     }

   if ( cc_AnswerCall(crn_buf, 0, EV_SYNC) < 0 )
     {
      procdevfail(ldev);
     .
     }

    .
    .
    .

   if ( cc_HoldCall(crn_buf, EV_SYNC) < 0 )
     {
      procdevfail(ldev);
    .
     }
    .
    .
   if ( cc_RetrieveCall(crn_buf, EV_SYNC) < 0 )
     {
      procdevfail(ldev);
    .
     }
    .
    .
   /* Drop the call */
   if ( cc_DropCall(crn_buf, NORMAL_CLEARING, EV_SYNC) < 0 )
     {
      procdevfail(ldev);
     .
     }
     .
   if ( cc_ReleaseCall(crn_buf) < 0 )
     {
      procdevfail(ldev);
     .
     }

   /* Close the device */
   if ( cc_Close( ldev ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}
```

```
int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_RetrieveCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ABORTED	Task aborted by **cc_Restart( )** function.
ERR_ISDN_LIB | E_BADSTATE	The **cc_Restart( )** function is in progress.
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state.
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

See Also

- **cc_HoldCall( )**
- **cc_RetrieveAck( )**
- **cc_RetrieveRej( )**

Name:	int cc_RetrieveRej(crn, cause)	
Inputs:	CRN crn	• call reference number
	int cause	• standard ISDN Network cause/error code
Returns:	0 on success <0 on failure	
Includes:	cclib.h	
Category:	Hold and Retrieve	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (Q.SIG only)	

■ Description

The **cc_RetrieveRej( )** function allows the application to reject a request to retrieve a held call from remote equipment. The call must be in Hold state and the CCEV_RETRIEVECALL event must be received before this function is called.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
cause:	A standard ISDN Network cause/error code is returned indicating the reason the hold request was rejected. Possible causes include TEMPORARY_FAILURE (Cause 41), NETWORK_OUT_OF_ORDER (Cause 38), and NETWORK_CONGESTION (Cause 42). For a complete list of ISDN Network cause/error codes, see *Section 7.2.2. Cause/Error Codes from the ISDN Network*.

■ Cautions

- The **cc_RetrieveRej( )** function can only be used after the call is in Hold state and after the CCEV_RETRIEVECALL event is received.
- The BRI protocols and the PRI Q.SIG protocol are the only protocols that allow the rejection of the retrieval of a call in the Hold state. The **cc_RetrieveRej( )** function cannot be used in the PRI DPNSS protocol. If used in unsupported protocols, the **cc_RetrieveRej( )** function call will pass but the retrieve from Hold request will not be rejected and the call will no longer be in the Hold state.

- Not all ISDN Network cause/error codes are universally supported across switch types. Before using a particular cause code, compare its validity with the appropriate switch vendor specifications.

Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <srllib.h>
#include <dtilib.h>
#include <cclib.h>

main()
{

   char *devname = “dtiB1T1”;
   LINEDEV ldev;
   .
   .
   .
   .


   if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }
   if ( sr_enbhdlr( ldev,CCEV_HOLDCALL,HoldCallHdlr ) < 0 )
   {
          printf( "dtiEnable for HoldCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }
   if ( sr_enbhdlr( ldev,CCEV_RETRIEVECALL,RetrieveCallHdlr ) < 0 )
   {
          printf( "dtiEnable for RetrieveCallHdlr failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
          return( 1 );
   }

   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitevt(-1);
   }

   cc_Close(ldev);

}

/*************************************************/
/* HoldCallHdlr - accept the hold call request     */
/*************************************************/
int HoldCallHdlr( )
{
```

```
    CRN crn_buf;
    LINEDEV ldev = sr_getevtdev();
    int len = sr_getevtlen();
    void *ev_datap = sr_getevtdatap();

    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_HoldAck( crn_buf ) < 0 )
      procdevfail(ldev);

    return( 0 );
}
/*********************************************************/
/* RetrieveCallHdlr - Reject the retrieve call request     */
/*********************************************************/
int RetrieveCallHdlr( )
{
    CRN crn_buf;
    LINEDEV ldev = sr_getevtdev();
    int len = sr_getevtlen();
    void *ev_datap = sr_getevtdatap();

    cc_GetCRN(&crn_buf, ev_datap);

    if ( cc_RetrieveRej( crn_buf, TEMPORARY_FAILURE ) < 0 )
      procdevfail(ldev);

    return( 0 );
}


int procdevfail(LINEDEV ldev)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(ldev);
        cc_ResultMsg(ldev,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_RetrieveRej( )** function include the following:

Error Code	Description
ERR_ISDN_FW | ISDN_BADSTATE	Cannot accept event in current state.
ERR_ISDN_LIB | E_ISNULL_PTR	Null pointer error

See Also

- **cc_HoldCall( )**
- **cc_RetrieveAck( )**
- **cc_RetrieveCall( )**

Name:	int cc_SetBilling(crn, rate_type, ratep, mode)	
Inputs:	CRN crn	• call reference number
	int rate_type	• type of billing data
	CC_RATE_U *ratep	• pointer to call charge rate
	unsigned long mode	• asynchronous or synchronous
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	asynchronous or synchronous	
Technology:	PRI (4ESS only)	

■ Description

The **cc_SetBilling( )** function sets the billing rate for AT&T Vari-A-Bill services associated with the particular call reference number, using cents as the unit. In asynchronous mode, the **cc_SetBilling( )** function must be used after either CCEV_CONNECTED or CCEV_ANSWERED is received (in call state Connected). In synchronous mode, **cc_SetBilling( )** must be used after successful completion of either **cc_MakeCall( )** or **cc_AnswerCall( )**.

NOTE: This function is specific to users of the AT&T Network's Vari-A-Bill service the AT&T network. Check with the service provider to see if this service can be used.

Parameter	Description
crn:	The call reference number. Each call needs a CRN.
rate_type:	The type of billing data. Possible values are: • ISDN_NEW_RATE - change to different per-minute rate • ISDN_FLAT_RATE - change to flat charge • ISDN_PREM_CHANGE - add additional charge to the call • ISDN_PREM_CREDIT - subtract charge from the call
ratep:	Pointer to the billing rate of the current call. The billing rate information is contained in the CC_RATE_U data structure. For a description of this structure, see *Section 6.1. CC_RATE_U*.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

Termination Events

- CCEV_SETBILLING - indicates that the billing information for the call has been acknowledged by the network. This event is returned only when the AT&T Vari-A-Bill feature is used.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

Cautions

- This function is available only on the AT&T network and only for the PRI 4ESS protocol.
- **cc_SetBilling( )** may not function in all service-provider environments. Check whether retrieving billing information is an option with the service provider.

Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   char      dnis_buf[CC_ADDRSIZE];
   CC_RATE_U rate;

   if ( cc_Open(&devhdl, devname, 0) < 0 )
   {
        printf("Error opening device: errno = %d\n", errno);
        exit(1);
   }

   printf("Waiting for call\n");
   if (cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_GetDNIS(crn, dnis_buf) < 0 )
        callfail(crn);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        /*
         * using cc_SetBilling( ) to set the rate of the current call.
```

```
         * This function is used in conjunction with AT&T Vari-A-Bill
         * service only.
         * Do not use it in other protocol applications
         */
    rate.ATT.cents = 99;
    if ( cc_SetBilling(crn, ISDN_FLAT_RATE, &rate, EV_SYNC) < 0 )
        callfail(crn);
        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}


int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle, reason, &msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_SetBilling( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_FB_UNAVAIL	Flex billing unavailable
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCALLID	Bad call identifier
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

NOTE: See *Section 7.2.1. Cause/Error Codes from the ISDN Firmware* for information on firmware return codes that are specific to the **cc_SetBilling( )** function.

See Also

- **cc_SetParm( )**

Name:	int cc_SetCallingNum(linedev, callingnum)	
Inputs:	LINEDEV linedev	• line device handle
	char *callingnum	• calling phone number string
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetCallingNum( )** function sets the default calling party number (caller ID) associated with the specific channel (line device) handle. The calling party number is contained in a NULL terminated ASCII string.

The calling party number may also be set in the **cc_MakeCall( )** function. When the calling party number is specified in the MAKECALL_BLK structure, then the **cc_MakeCall( )** function uses the number in the structure for the current call. Subsequent calls return to the default calling party number set by the **cc_SetCallingNum( )** function when the MAKECALL_BLK structure is not used.

Parameter	Description
linedev:	The line device handle.
callingnum:	The phone number of the calling party.

■ Caution

This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
```

```
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “dtiB1T1”;

   if ( cc_Open(&devhdl, devname, 0) < 0 )
   {
      printf(“Error opening device: errno = %d\n”, errno);
      exit(1);
   }
        /*
         * Using cc_SetCallingNum(devhndl, “9933000”) to
         * set default calling party number
         */

   if ( cc_SetCallingNum(devhdl, “9933000”) < 0 )
        procdevfail(devhdl);

   if ( cc_MakeCall(devhdl, &crn, ”9933000”, NULL, 30, EV_SYNC) < 0 )
        procdevfail(devhdl);
        .
        .
        .
        .
        .
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close(devhdl) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);

}

int callfail(CRN crn)
{
       LINEDEV ld;
        cc_CRN2LineDev(crn, &ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
       int reason;
       char *msg;
       reason = cc_CauseValue(handle);
       cc_ResultMsg(handle, reason, &msg);
       printf(“reason = %x - %s\n”,reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetCallingNum( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADIF	Bad interface number

See Also

- **cc_MakeCall( )**

Name:	int cc_SetChanState(linedev, chanstate, mode)	
Inputs:	LINEDEV linedev	• line device handle
	int chanstate	• channel service state
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous or asynchronous	
Technology:	PRI (all protocols)	

■ Description

The **cc_SetChanState( )** function is used to change the maintenance state of a specified B channel. When power is first turned on, all channels are placed in the IN_SERVICE state. However, in some protocols, the D channel may need to be activated.

NOTE: This feature may not be available in some countries.

Parameter	Description
linedev:	The line device handle of the B channel.
chanstate:	The channel service state. Possible values for **chanstate** are: • IN_SERVICE - Informs the board that the host is ready to receive and send a message. • MAINTENANCE - Informs the host that normal outgoing traffic is not allowed. Only an incoming test call is permitted. • OUT_OF_SERVICE - Informs the board that the host is not ready to receive or send a message. For some protocols, the firmware will reject all incoming and outgoing requests.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous (EV_SYNC) mode.

■ Termination Events

- CCEV_SETCHANSTATE - indicates that the B channel has been placed in the requested state.

- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. Typically, this event is triggered by an incorrect function call during the call.

■ Cautions

- The **cc_SetChanState( )** function should only be invoked in the Null state. The Null state occurs immediately after a call to either the **cc_Open( )** or **cc_ReleaseCall( )** function.
- The **cc_SetChanState( )** function affects only the link between the calling process and the device. Other processes and devices are not affected.
- **cc_SetChanState( )** is not supported for E-1 ISDN or NTT PRI protocols, or for any BRI protocols.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “dtiB1T1”;

   if ( cc_Open(&devhdl, devname, 0) < 0 )
   {
      printf(“Error opening device: errno = %d\n”, errno);
      exit(1);
   }
        /*
         * using cc_SetChanState(devhdl, IN_SERVICE,
         * EV_SYNC) to set B channel to “in service” state.
         * Recommended for all supported protocols.
         */

   if ( cc_SetChanState(devhdl, IN_SERVICE, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);
        .
        .
        .
        .
        .
```

```
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
       LINEDEV ld;
        cc_CRN2LineDev(crn,&ld);
        procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
       int reason;
       char *msg;
       reason = cc_CauseValue(handle);
       cc_ResultMsg(handle, reason, &msg);
       printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetChanState( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

■ See Also

- **cc_WaitCall( )**

Name:	int cc_SetDChanCfg(boarddev, dchan_cfgptr)	
Inputs:	LINEDEV boarddev	• line device handle of the D channel board
	DCHAN_CFG dchan_cfgptr	• pointer to DCHAN_CFG structure
Returns:	0 on success < 0 on failure	
Includes:	isdncmd.h isdnlib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetDChanCfg( )** function sets the configuration of the Digital Subscriber Loop (DSL) for the D channel and causes the activation of links if the switch type specified is valid. This function specifies the DSL-specific and logical Data Link-specific parameters. These parameters include switch type, switch side (Network or User) and terminal assignment (fixed Terminal Endpoint Identifier or auto-initializing Terminal Endpoint Identifier). Each station interface is configured separately, which allows different protocols to be run on different stations simultaneously.

When the switch is operating as the User side in North American protocols, the **cc_SetDChanCfg( )** function is used to program the Service Profile Identifier (SPID). The SPID must be transmitted and acknowledged by the switch (see the **cc_TermRegisterResponse( )** function for more information).

The **cc_SetDChanCfg( )** function is also used to define Layer 3 timer values, specify Layer 2 Access and set firmware features such as firmware-applied call progress tones. For a complete listing of the values that can be specified using the **cc_SetDChanCfg( )** function, see *Section* 6.3. DCHAN_CFG.

NOTE: Although this function is supported for BRI/2 and PRI protocols, it can be used only to define Layer 3 timer values. All of the other values in the DCHAN_CFG structure are applicable only to BRI/SC.

Parameter	Description
boarddev:	The line device handle of the D channel board.
dchan_cfgptr:	The pointer to the D channel configuration (DCHAN_CFG) block. See *Section* 6.3. DCHAN_CFG for a description of the DCHAN_CFG data structure and for possible field values.

■ Cautions

- The **cc_SetDChanCfg( )** function must be issued prior to any call control commands.
- All components of the DCHAN_CFG structure that pertain to the configuration must be set. There are no default values.
- Issuing a subsequent **cc_SetDChanCfg( )** command reinitializes all layers of the protocol stack. As a result, all existing calls for that particular D channel are lost. The application receives CCEV_DISCONNECTED events for any calls that existed when the **cc_SetDChanCfg( )** function was called. The application should follow normal call teardown procedures after receiving the CCEV_DISCONNECTED event.

■ Example

```
#include <windows.h>   /* for Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
#include “isdncmd.h”
#include “isdnlib.h”


/* Global variables */

LINEDEV          boarddev;
DCHAN_CFG        dchan_cfg;
main ()
{

dchan_cfg.layer2_access         = FULL_ISDN_STACK;     /* full protocol */
dchan_cfg.switch_type           = ISDN_BRI_NTT;        /* NTT switch */
dchan_cfg.switch_side           = USER_SIDE;           /* User Terminal */
dchan_cfg.number_of_endpoints   = 1;                   /* one terminal */
dchan_cfg.user.tei_assignment   = FIXED_TEI_TERMINAL;  /* Fixed TEI terminal */
dchan_cfg.user.fixed_tei_value  = 23;                  /* TEI assigned to terminal */
dchan_cfg.tmr.te.T303 = TMR_DEFAULT; /* NOTE: the values chosen are arbitrary. */
dchan_cfg.tmr.te.T304 = TMR_DEFAULT;
```

```
dchan_cfg.tmr.te.T305 = TMR_DEFAULT;
dchan_cfg.tmr.te.T308 = TMR_DEFAULT;
dchan_cfg.tmr.te.T310 = TMR_DEFAULT;
dchan_cfg.tmr.te.T313 = TMR_DEFAULT;
dchan_cfg.tmr.te.T318 = TMR_DEFAULT;
dchan_cfg.tmr.te.T319 = TMR_DEFAULT;

.
.
.
if (cc_Open(&boarddev, "briS1", 0) != 0)
{
     printf("cc_open: error\n");
}

if (cc_SetDChanCfg(boarddev, &dchan_cfg) == 0)
{
    printf("Configuration is set\n");
}
else
    printf("Configuration could not be set\n");


/*
 * Wait for Link Activation confirmation
 * After which call processing can be started.
 */
.
.
.
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetDChanCfg( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISNOMEM	Cannot map or allocate memory
ISDN_INVALID_SWITCH_TYPE	Switch type requested is not supported
ISDN_MISSING_FIXED_TEI	Fixed Terminal Endpoint Identifier (TEI) value not provided for non-initializing terminal
ISDN_MISSING_DN	Directory number not specified for terminal
ISDN_MISSING_SPID	Service Profile Identifier (SPID) not provided for North American Terminal

See Also

- **cc_GetDLinkState( )**

Name:	int cc_SetDLinkCfg(bdev, dlinkptr, dlinkcfgptr)	
Inputs:	LINEDEV bdev	• device handle
	DLINK *dlinkptr	• pointer to data link information block
	DLINK_CFG *dlinkcfgptr	• pointer to location of D channel logical link configuration block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC	

■ Description

The **cc_SetDLinkCfg( )** function configures a logical link. This function initializes the required firmware structures to allow the logical link to be used.

Parameter	Description
bdev:	Station device handle.
dlinkptr:	Pointer to the data link information block. See *Section 6.4. DLINK* for a description of the elements of this data structure.
dlinkcfgptr:	Pointer to the buffer containing the data link logical link configuration block. See *Section 6.5. DLINK_CFG* for a description of the elements of this data structure.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
```

```
#include "cclib.h"

/* Global variables */

LINEDEV      ldev;   /* Board device handle */


main()
{
   DLINK dlink;
   DLINK_CFG cfg;

   .
   .
   .

   dlink.sapi = 0;
   dlink.ces = 1;

   cfg.tei = AUTO_TEI;
   cfg.state = DATA_LINK_UP;
   cfg.protocol = DATA_LINK_PROTOCOL_Q931;

   if ( cc_SetDLinkCfg(ldev, &dlink, &cfg) < 0) {
      printf("error");
   } else {
       .
       .
       .
   }
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetDLinkCfg( )** function include the following:

Error Code	Description
E_BADDEV | ERR_ISDN_LIB	Bad Device Descriptor
E_INVNDIINTERFACE | ERR_ISDN_LIB	Invalid NDI Interface
E_INVNRB | ERR_ISDN_LIB	Invalid NRB

■ See Also

- **cc_GetDLinkCfg( )**

Name:	int cc_SetDLinkState(bdev, dlinkptr, state_buf)	
Inputs:	LINEDEV bdev	• device handle
	DLINK *dlinkptr	• pointer to data link information block
	int *state_buf	• pointer to location of D channel state
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetDLinkState( )** function asks the firmware to set the logical data link state to support specific events in the application.

Upon successful completion of the **cc_SetDLinkState( )** function, the request to change the state of the logical link is accepted by the firmware. Subsequently, when the logical data link state changes, the unsolicited event CCEV_D_CHAN_STATUS will be received, indicating that the state has changed.

Parameter	Description
bdev:	Board device handle for PRI, station device handle for BRI.
dlinkptr:	Pointer to the data link information block. See *Section 6.4. DLINK* for a description of the elements of this data structure.
state_buf:	Pointer to the buffer containing the requested data link state value. Possible data link states are: • DATA_LINK_UP - the firmware attempts to activate the logical link if it is not already activated, and allows the Network side to establish the logical link if requested. • DATA_LINK_DOWN - the firmware attempts to release the logical link if it is currently established, and allows the Network side to establish the logical link if requested. • DATA_LINK_DISABLED - the firmware attempts to release the logical link if it is currently established. The firmware does not allow the Network side to establish the logical link if requested.

■ Cautions

- There needs to be a sufficient amount of time between bring down the data link layer and bringing it up. This is necessary to allow time for the network side to release its resources and declare the data link down before the network side tries to reestablish the connection.
- Although the **cc_SetDLinkState( )** function can be used for PRI, it is somewhat limited in scope. In PRI, after layer 2 is brought down (DATA_LINK_DOWN state), the firmware will try to reestablish the link after the timer expires.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/* Global variables */
```

```
LINEDEV          ldev;

main()
{

DLINK dlink;
int state;
.
.
.
        /* Establish the data link on SAPI 0, CES 1 */
       dlink.sapi = 0;
       dlink.ces  = 0;
       state = DATA_LINK_UP;
       if(cc_SetDLinkState(ldev, &dlink, &state) < 0) {
              printf("error");
       } else {
              .
              .
              .
       }
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetDLinkState( )** function include the following:

Error Code	Description
E_BADDEV | ERR_ISDN_LIB	Bad Device Descriptor
E_INVNDIINTERFACE | ERR_ISDN_LIB	Invalid NDI Interface
E_INVNRB | ERR_ISDN_LIB	Invalid NRB

■ See Also

- **cc_GetDLinkState( )**

Name: int cc_SetEvtMsk(linedev, mask, action)
Inputs: LINEDEV linedev • line device handle
unsigned long mask • bitmask or events
int action • action to be taken on the mask bit
Returns: 0 on success
< 0 on failure
Includes: cclib.h
Category: System tools
Mode: synchronous
Technology: BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_SetEvtMsk( )** function sets the event mask associated with the specified line device to support specific events in the application.

NOTE: ISDN does not support disabling GCEV_BLOCKED or GCEV_UNBLOCKED in the **cc_SetEvtMsk( )** function

Parameter	Description
linedev:	The line device handle.
mask:	The event to be enabled or disabled by setting the bitmask for that event (see *Table 24*). Multiple transition events may be enabled or disabled with one function call if the bitmask values are bitwise OR'ed.
action:	Specifies whether to set, add, or subtract the mask bit(s) as specified in the bitmask (see *Table 25*).

The possible **bitmask** values and the actions that they control are described in *Table 24* below.

Table 24. Bitmask Values

Bitmask Type	Action	Default
CCMSK_ALERT	Receiving CCEV_ALERTING	Enabled

Bitmask Type	Action	Default
CCMSK_CALLACK_SEND	Application sends first response to SETUP CCMSK_CALLPROC_SEND message, either CALL_SETUP_ACK or CALL_PROCEEDING	Firmware sends first response to SETUP message
CCMSK_PROCEEDING	Receiving CCEV_PROCEEDING	Enabled
CCMSK_PROGRESS	Receiving CCEV_PROGRESSING	Enabled
CCMSK_SERVICE	Receiving CCEV_SERVICE. When this event arrives, the application should respond with a status message using **cc_SndMsg( )**. The firmware will not automatically respond to this message.	Not enabled
CCMSK_SERVICE_ACK	Receiving CCEV_SETCHANSTATE. When this event is masked, **cc_SetChanState( )** may be blocked.	Not enabled
CCMSK_SETUP_ACK	Receiving CCEV_SETUP_ACK	Not enabled
CCMSK_STATUS	Receiving CCEV_STATUS	Not enabled
CCMSK_STATUS_ ENQUIRY	Receiving CCEV_STATUS_ ENQUIRY. When this event arrives, the application should respond with a status message using **cc_SndMsg( )**. The firmware will not auto respond to this message.	Not enabled

Bitmask Type	Action	Default
CCMSK_TMREXPEVENT	Receiving the CCEV_TIMER event. This event is generated when some timer expires at the firmware in Layer 3. Timer ID, Call ID and the value of the timer are returned.	Not enabled

The **action** parameter may either set or reset the mask bit(s) as specified in the bitmask. Possible actions are shown in *Table 25* below.

Table 25. Bitmask Actions

Action	Description
CCACT_SETMSK:	Enables notification of events specified in the bitmask and disables notification of previously set events.
CCACT_ADDMSK:	Enables notification of events specified in the bitmask in addition to previously set events.
CCACT_SUBMSK:	Disables notification of events specified in the bitmask.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
{
  short devhdl;  /* device handle for D channel */
  .
  .
  .
  if ( cc_Open(devhdl,”dtiB1”, 0 ) < 0 )
     exit(1);
```

```
  /*
   * using cc_ SetEvtMsk (devhdl,CCMSK_PROGRESS
   * | CCMSK_ALERT, CCACT_ADDMSK) to block the incoming ISDN
   * message ALERTING and PROGRESSING. Note that the devhdl is a
   * board level device.
   */
  if ( cc_SetEvtMsk (devhdl, CCMSK_PROGRESS | CCMSK_ALERT,
         CCACT_ADDMSK) < 0 )
                procdevfail(devhdl);

  /* continue the program. */
  .
  .
  .
  .
  .
  if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);
     procdevfail(devhdl);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle, reason, &msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetEvtMask( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

See Also

- **cc_SetParm( )**
- **cc_GetEvtMsk( )**

Name:	int cc_SetInfoElem(linedev, iep)	
Reference:	LINEDEV linedev	• line device handle of the B channel board
Inputs:	IE_BLK *iep	• pointer to the information element buffer
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetInfoElem( )** function sets additional information elements, allowing the application to include application-specific ISDN information elements in the next outbound message. This function is used for rapid deployment of an application that "interworks" with the network to take advantage of ISDN's capabilities. A typical application is user-to-user information elements in each outgoing message.

NOTE: See *Appendix C* for descriptions of ISDN IEs that are specific to the DPNSS protocol.

Parameter	Description
linedev:	The B channel board line device handle.
iep:	The starting address of the information element block (IE_BLK). For a description of the IE_BLK data structure, see *Section 6.6. IE_BLK*. See the Example code for details.

■ Cautions

- This function must be used immediately before calling a function that sends an ISDN message. The information elements specified by this function are applicable only to the next outgoing ISDN message.
- The line device handle in the parameter must be the same as the one used in the function call that sends the ISDN message.
- The IE data length must not exceed MAXLEN_IEDATA of 254 bytes.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

typedef struct uui_type {
        unsigned char uui_id;
        unsigned char length;
        char data[128];
} USR_USR_INFO;      /* This structure complies with CCITT Q.931
                        USER-USER information element standard */

int build_struui(USR_USR_INFO *uuip,char *str)
{
        uuip->uui_id = 0x7E;       /* CCITT UUI */
        strcpy(uuip->data, str);
        uuip->length = strlen(uuip->data);
        return(uuip->length+2);
}

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “dtiB1T1”;
 IE_BLK  icp;
   USR_USR_INFO uui;

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf(“Error opening device: errno = %d\n”, errno);
      exit(1);
   }

        /*
         * Using cc_SetInfoElem()to include additional
         * information elements in the next ISDN message.
         */

   icp.length = build_struui(&uui, “Dialogic ISDN”);
   memcpy(icp.info_elem_str, &uui, icp.length);
   if ( cc_SetInfoElem(devhdl, &icp))
        procdevfail(devhdl);

   if ( cc_MakeCall(devhdl, &crn, ”9933000”, NULL, 30, EV_SYNC) < 0 )
        procdevfail(devhdl);
        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);
```

```
}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetInfoElem( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISNOINFOBUF	Information buffer not ready

■ See Also

- **cc_SetParm( )**

Name:	int cc_SetMinDigits(linedev, mindigits)	
Inputs:	LINEDEV linedev	• device handle
	int mindigits	• the minimum number of digits to be collected
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetMinDigits( )** function sets the minimum number of digits to be collected prior to receiving a CCEV_OFFERED event, in asynchronous mode, or prior to executing the **cc_WaitCall( )** function, in synchronous mode. For example, if the minimum number of digits is set to 10, the firmware will not generate CCEV_OFFERED or the **cc_WaitCall( )** function will not complete until at least 10 digits are collected.

Parameter	Description
linedev:	The line device handle or, for applications using the call waiting feature, the board device handle (BRI and Windows only).
mindigits:	Specifies the minimum number of digits to be collected prior to receiving a CCEV_OFFERED event (asynchronous) or prior to executing the **cc_WaitCall( )** function (synchronous).

■ Cautions

Using **cc_SetMinDigits( )** on a board device will overwrite any prior setting used on an individual channel device belonging to that board.

■ Example

NOTE: See the code sample in the **cc_MakeCall( )** function description for an example of how to use the **cc_SetMinDigits( )** function with the call waiting feature (BRI and Windows only).

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   char      dnis_buf[CC_ADDRSIZE];

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }
  /*
   * using cc_SetMinDigits(linedev, mindigits) to set minimum
   * digits to be collected before terminating the WaitCall().
   * Only needed when overlap receiving is needed.
   */
   if ( cc_SetMinDigits(linedev, mindigits) < 0 )
        procdevfail(devhdl);

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   printf("Retrieving DNIS\n");
   if ( cc_GetDNIS(crn,dnis_buf) < 0 )
        callfail(crn);
   else
        printf("cc_GetDNIS succeeded: %s\n",dnis_buf);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

   /* Voice process after this. */
        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}
```

```
int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetMinDigits( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

- **cc_WaitCall( )**
- **cc_MakeCall( )**

Name:	int cc_SetParm(linedev, parm_id, value)	
Inputs:	LINEDEV linedev	• line device handle
	int parm_id	• parameter identifier
	long value	• parameter value
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetParm( )** function sets the default channel parameters of the line device. The channel parameters are included in the call setup message to describe the characteristics of the channel and tell the network the path for routing the call.

Parameter	Description
linedev:	The line device handle.
parm_id:	The parameter identification. See *Table 26* below for a list of possible parameter ID definitions.
value:	The address of the buffer where the default parameters/information will be stored.

The following table describes the possible parameter ID definitions for the **cc_SetParm( )** function. The parameters are the same as those listed for the **cc_GetParm( )** function.

Table 26. cc_SetParm( ) Parameter ID Definitions

Define	Description	Possible return values
BC_XFER_CAP	Bearer channel information transfer capability	BEAR_CAP_SPEECH - speech BEAR_CAP_UNREST_DIG - unrestricted data BEAR_REST_DIG - restricted data
BC_XFER_MODE	Bearer channel Information Transfer Mode	ISDN_ITM_CIRCUIT - circuit switch
BC_XFER_RATE	Bearer channel, Information Transfer Rate	BEAR_RATE_64KBPS - 64Kbps transfer rate
USRINFO_LAYER1_ PROTOCOL	Layer 1 protocol to use on bearer channel	ISDN_UIL1_CCITTV.110 - CCITT standardized rate adaptation ISDN_UIL1_G711uLAW - Recommendation G.117 u-Law ISDN_UIL1_G711ALAW - Recommendation G.117 a-Law ISDN_UIL1_G721ADCPM - Recommendation G.721 32 kbits/s ADCPM and Recommendation I.460 ISDN_UIL1_G722G725 - Recommendation G.722 and G.725 - 7kHz audio ISDN_UIL1_H261 - Recommendation H.261 - 384 kbits/s video ISDN_UIL1_NONCCITT - Non-CCITT standardized rate adaptation ISDN_UIL1_CCITTV120 - CCITT standardized rate adaptation V.120 ISDN_UIL1_CCITTX31 - CCITT standardized rate adaptation X.31 HDLC

Define	Description	Possible return values
USR_RATE	User rate to use on bearer channel (layer 1 rate)	ISDN_UR_EINI460 - determined by E bits in I.460 ISDN_UR_56000 - 56 kbits, V.6 ISDN_UR_64000 - 64 kbits, X.1 ISDN_UR_134 - 134.5 bits, X.1 ISDN_UR_12000 - 12 kbits, V.6
CALLED_NUM_TYPE	Called number type	EN_BLOC_NUMBER - number is sent en-block (in whole, not overlap sending) INTL_NUMBER - international number for international call. (Verify availability with service provider.) NAT_NUMBER - national number for call within national numbering plan (accepted by most networks) LOC_NUMBER - subscriber number for a local call. (Verify availability with service provider.) OVERLAP_NUMBER - overlap sending; number is not sent in whole (not available on all networks)
CALLED_NUM_PLAN	Called number plan	UNKNOWN_NUMB_PLAN - unknown number plan ISDN_NUMB_PLAN - ISDN/telephony (E.164/E.163) (accepted by most networks) TELEPHONY_NUMB_PLAN - telephony numbering plan PRIVATE_NUMB_PLAN - private numbering plan
CALLING_NUM_TYPE	Calling number type	Same values as CALLED_NUM_TYPE

Define	Description	Possible return values
CALLING_NUM_PLAN	Calling number plan	Same values as CALLED_NUM_PLAN
CALLING_PRESENTATION	Calling presentation indicator	PRESENTATION_ALLOWED - allows the display of the calling number at the remote end
CALLING_SCREENING	Calling screening indicator field	USER_PROVIDED - user provided, not screened (passes through)
RECEIVE_INFO_BUF	Multiple IE buffer; sets the size of the buffer, that is, the number of messages that can be stored in the information queue	Any number in the range of 1 to MAX_RECEIVE_INFO_BUF (The **cc_SetParm( )** function returns -1 (error) or 0 (success).)

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “dtiB1T1”;
   long   value;

   if ( cc_Open( &devhdl, devname, 0 ) < 0  )
   {
      printf(“Error opening device: errno = %d\n”, errno);
      exit(1);
   }
```

```
   /*
    * using cc_ SetParm(devhdl, CALLED_NUM_TYPE,EN_NAT_NUMBER)
    * to set the default value for "called party number type" to
    * 'enbloc national' type
    */
   if ( cc_SetParm(devhdl, CALLED_NUM_TYPE, NAT_NUMBER) < 0 )
        procdevfail(devhdl);
   else
      printf("Parameter CALLED_NUM_TYPE has value: 0x%x\n",value);

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);


}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetParm( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

See Also

- **cc_GetParm( )**
- **cc_SetParmEx( )**

Name:	int cc_SetParmEx(linedev, parm_id, *parminfoptr)	
Inputs:	LINEDEV linedev	• line device handle
	int parm_id	• parameter identifier
	PARM_INFO *parminfoptr	• pointer to the PARM_INFO block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SetParmEx( )** function is an extension of the **cc_SetParm( )** function that allows the application to set parameters requiring variable data to be passed down to the firmware, without reconfiguration. These parameters include the Service Protocol Identifier (SPID) number, the directory number, and the subaddress of a User-side line. The SPID must first be specified using the **cc_SetDChanCfg( )** function and then can be changed using the **cc_SetParmEx( )** function. The directory number and subaddress, which are used to restrict incoming calls to those whose parameters match the specified values, can be set and/or changed at any time using the **cc_SetParmEx( )** function.

Parameter	Description
linedev:	The line device handle.
parm_id:	The parameter identification. This function supports all of the parameters listed in Table 26 in the **cc_SetParm( )** function description. In addition, for BRI/SC only, the following parameters are also supported: • SPID_NUMBER - Service Protocol Identifier Number (applicable to BRI North American Protocols only) • SUBADDR_NUMBER - Subaddress Number (applicable to BRI User Side switches only) • DIRECTORY_NUMBER - Directory Number (applicable to BRI User Side switches only) The values for **parm_id** are defined in *cclib.h*.
parminfoptr:	A pointer to the PARM_INFO structure, which contains variable data to set in the firmware on a call-by-call basis. For a description of the PARM_INFO data structure, see *6.10. PARM_INFO*.

■ Cautions

- In order to use the **cc_SetParmEx( )** function to change the SPID, the D channel must first be configured as an initializing terminal using the **cc_SetDChanCfg( )** function.
- When the link goes down, the last SPID negotiated with the network will be the only SPID to be renegotiated when the link comes back up. This means that both B channels will end up with the same renegotiated SPID. Use the **cc_SetParmEx( )** function to reset the SPIDs for the B channels.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
```

```
{
   LINEDEV   ldev = 0;
   CRN       crn = 0;
   char     *devname = "briS1T1";  /* device name for BRI station 1 timeslot 1 */
   PARM_INFO *parminfo;   /* variable data to be set */
   char *DN = "99330008080";
   .
   .
   .
  /* open the ISDN board device */
  if ( cc_Open( &ldev, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

  /* initialize PARM_INFO structure */
   parminfo.parmdatalen = strlen(DN);
   strcpy (parminfo.parmdata, DN); /* directory number */

  /* Specify the Directory Number */
   if cc_SetParmEx(ldev, DIRECTORY_NUMBER, parminfo) < 0)
   {
      printf("Error in cc_SetParmEx(): %d\n", cc_CauseValue(ldev);
   }

   /* initialize D channel with cc_SetDChanCfg */
        .
        .
        .
   /* continue with call processing */
        .
        .
        .
        .
} /* end main */

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SetParmEx( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter

See Also

- **cc_GetParmEx( )**
- **cc_SetParm( )**

Name:	int cc_SetUsrAttr(linedev, usrattr)	
Inputs:	LINEDEV linedev	• line device handle
	long usrattr	• user attribute information
Returns:	0 on success	
	< 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI	

■ Description

The **cc_SetUsrAttr( )** function sets the user attribute for a line device for later retrieval. The user attribute value can be a memory pointer used to identify a board and a channel on a board. The value can also be a pointer to a user-defined structure such as the current state or the active call reference number.

For example, the application can use the first two digits to identify a board and the last two digits to identify a channel on a board, with a '0' inserted between the numbers to separate them. The number '12024' would indicate the 24th channel on board 12.

Parameter	Description
linedev:	The line device handle.
usrattr:	The user-defined attribute. Applications can recall this number by calling the **cc_GetUsrAttr( )** function.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"
```

```
main()
{
   int chan = 1;
   char devname[16];
   .
   .
   .
   .
   if ( sr_enbhdlr( devhdl, CCEV_DISCONNECTED, discCallHdlr ) < 0 )
   {
           printf( "dtiEnable for DISCONNECT failed:  %s\n",
                                ATDV_ERRMSGP( SRL_DEVICE ) );
           return( 1 );
   }

   printf(devname,"dtiB1T%d",chan);
   if ( cc_Open(&devhdl, devname, 0) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   if ( cc_SetUsrAttr(devhdl, chan))
        procdevfail(devhdl);
   .
   .
   .
   .
   while (1)
   {
      /* wait for network event */
        sr_waitcvt(-1);
   }

   cc_Close(devhdl);

}

/*************************************************/
/* discCallHdlr - disconnect the active call       */
/*************************************************/
int discCallHdlr( )
{
    int devindx;
    int dev;
    int len;
    void *datap;
    CRN crn;
    long chan;

    dev = sr_getevtdev();
    len = sr_getevtlen();
    datap = sr_getevtdatap();

    cc_GetCRN(&crn, datap);
    if ( cc_GetUsrAttr(dev,&chan))
        procdevfail(dev);
    else
        printf("Call disconnected at chan %ld\n",chan);

    if ( cc_DropCall(crn, NORMAL_CLEARING, EV_ASYNC) < 0 )
        procdevfail(dev);

    return( 0 );
}
```

```
int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle, reason, &msg);
        printf(""reason = %x - %s\n"",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Typically, a < 0 return code for the **cc_SetUsrAttr( )** function indicates that the function reference (the device handle) is not valid for the function call.

■ See Also

- **cc_GetUsrAttr( )**
- **cc_Close( )**

Name:	int cc_SndFrame(linedev, sndfrmptr)	
Inputs:	LINEDEV linedev	• line device handle for the D channel
	L2_BLK *sndfrmptr	• pointer to the transmit frame buffer
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Data link layer handling	
Mode:	synchronous	
Technology:	BRI/SC; PRI (excluding Q.SIG)	

■ Description

The **cc_SndFrame( )** function sends a frame to the data link layer. When the data link layer is successfully established, the application will receive a CCEV_D_CHAN_STATUS event. If the data link layer is not established before the function is called, the function will be returned with a value < 0 indicating function failure.

NOTE: To enable Layer 2 access, set parameter number 24 to 01 in the firmware parameter file. When Layer 2 access is enabled, only the **cc_GetFrame( )** and **cc_SndFrame( )** functions can be used (no calls can be made).

Parameter	Description
linedev:	The line device handle for the D channel board.
sndfrmptr:	Pointer to the buffer containing the requested transmit frame. The transmit frame is stored in the L2_BLK data structure. For a description of the structure, see *Section 6.7. L2_BLK*. See the Example code for details.

■ Cautions

- The data link layer must be successfully established before the **cc_SndFrame( )** function is called.
- This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"


/* Global variables */


typedef long int (*EVTHDLRTYP)( );
.
.
.
void InitSndFrameBlk (L2_BLK *l2_blk_ptr)
          {
           l2_blk_ptr -> length = 13;         /* 13 bytes of data */
           l2_blk_ptr -> data [0] = 0x08;     /* Protocol discriminator */
           l2_blk_ptr -> data [1] = 0x02;     /* CRN length - 2 bytes */
           l2_blk_ptr -> data [2] = 0x03;     /* CRN = 8003 */
           l2_blk_ptr -> data [3] = 0x80;
           l2_blk_ptr -> data [4] = 0x6e;     /* msg type = NOTIFY */

           /* The first IE */
           l2_blk_ptr -> data [5] = 0x27;     /* IE type = 27 (NOTIFY) */
           l2_blk_ptr -> data [6] = 0x01;     /* The length of NOTIFY */
           l2_blk_ptr -> data [7] = 0xF1;     /* Notify indication */

           /* The second IE */
           l2_blk_ptr -> data [8] = 0x76;     /* IE type = 76 (REDIRECTION) */
           l2_blk_ptr -> data [9] = 0x03;     /* length of redirection */
           l2_blk_ptr -> data [10] = 0x01;    /* unknown type and E164 plan */
           l2_blk_ptr -> data [11] = 0x03;    /* network provides presentation */
           l2_blk_ptr -> data [12] = 0x8D;    /* reason = transfer */
          }

int evt_hdlr( )
{
   int ldev = sr_getevtdev( );
   unsigned long *ev_datap = (unsigned long *)sr_getevtdatap( );
   int len = sr_getevtlen( );
   int rc = 0;
   L2_BLK l2_blk;

      InitSndFrameBlk (&l2_blk);  /* prepare for the message */

      if ((rc = cc_SndFrame(ldev, &l2_blk) ) ! = 0)
         {
          printf("Error in cc_SndFrame, rc = %x\n", rc);
         }
      else
         .
         .
         .
.
         .
}
```

Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_SndFrame( )** function include the following:

Error Code	Description
ERR_ISDN_LIB|E_ISBADIF	Bad interface number
ERR_ISDN_LIB|E_ISNOMEM	Cannot map or allocate memory

See Also

- **cc_GetFrame( )**

Name:	int cc_SndMsg(crn, msg_type, sndmsgptr)	
Inputs:	CRN crn	• call reference number
	int msg_type	• ISDN message type
	IE_BLK *sndmsgptr	• pointer to the information element (IE) block
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SndMsg( )** function sends a non-Call State related ISDN message to the network over the D channel, while a call exists. The data is sent transparently over the D channel data link using the LAPD (Layer 2) protocol.

For BRI, the **cc_SndMsg( )** function is used to invoke supplemental services, such as Called/Calling Party Identification, Call Transfer, and Message Waiting. The services are invoked by sending Facility Messages or Notify Messages (see Table 27) to the switch. Upon receipt of the message, the network may return a NOTIFY message to the user. The NOTIFY messages can be retrieved by calling the **cc_GetCallInfo( )** function. For more information on invoking supplemental services, see *Appendix D*.

NOTE: The message must be sent over a channel that has a call reference number assigned to it.

Parameter	Description
crn:	The call reference number. Each call needs a valid CRN.
msg_type:	Specifies one of the ISDN message types listed in *Table 27* below. The values for **msg_type** are defined in *cclib.h*. Descriptions of the message types for DPNSS are provided in *Appendix C*.
sndmsgptr:	Points to the buffer (IE_BLK) that contains the information element(s) to be sent in the message. For a description of the data structure used to send the IEs, see *Section 6.6. IE_BLK*. See the Example code below for details.

Table 27. ISDN Message Types for cc_SndMsg( )

All Protocols	Custom BRI 5ESS only	DPNSS only
SndMsg_Congestion	SndMsg_Drop	SndMsg_Divert
SndMsg_Facility	SndMsg_DropAck	SndMsg_Intrude
SndMsg_FacilityAck	SndMsg_DropRej	SndMsg_NSI
SndMsg_FacilityRej	SndMsg_Redirect	SndMsg_Transfer
SndMsg_Information		SndMsg_Transit
SndMsg_Notify		
SndMsg_Status		
SndMsg_StatusEnquiry		
SndMsg_UsrInformation		

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/* FUNCTION PROTOTYPES */
void InitSndMsgBlk();   /* fills in the information element to send */

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";
   IE_BLK    ie_blk;   /* Info elem block pointer for SndMsg() */
   .
   .
   .

   /* open the ISDN line device */
   if ( cc_Open(&devhdl, devname, 0) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   /* wait for the incoming call */
```

```
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   /* answer the call */
   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0)
        callfail(crn);

   /*initialize the SndMsg Block to send */
   InitSndMsgBlk(&ie_blk);

/* send INFORMATION IE data */
   if ( cc_SndMsg(crn, SndMsg_Information, &ie_blk) == -1> { < 0 )
        callfail(crn);
   }
   /* drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   /* release the CRN */
   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);

   /* close the line device */
   if ( cc_Close(devhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}  /* end of main */

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}

void InitSndMsgBlk (IE_BLK * IE_BLK_PTR)
   {

   IE_BLK_PTR-> length  = 10;    /* 10 bytes of data */

   IE_BLK_PTR-> data[0] = 0x28;  /* DISPLAY IE (0x28) */
   IE_BLK_PTR-> data[1] = 0x08;  /* IE length:  8 bytes */
   IE_BLK_PTR-> data[2] = 0x46;
   IE_BLK_PTR-> data[3] = 0x52;
   IE_BLK_PTR-> data[4] = 0x2E;
   IE_BLK_PTR-> data[5] = 0x20;
   IE_BLK_PTR-> data[6] = 0x30;
   IE_BLK_PTR-> data[7] = 0x2E;
   IE_BLK_PTR-> data[8] = 0x32;
   IE_BLK_PTR-> data[9] = 0x30;
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_SndMsg( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_BADSTATE	Bad state
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

■ See Also

- **cc_GetCallInfo( )**
- **cc_GetSigInfo( )**
- **cc_SndNonCallMsg( )**

Name:	int cc_SndNonCallMsg(boarddev, crn_type, msg_type, sndmsgptr)	
Inputs:	LINEDEV boarddev	• board device
	int crn_type	• ISDN call reference number type
	int msg_type	• ISDN message type
	NONCRN_BLK *sndmsgptr	• pointer to the message block containing non-call related ISDN message data
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Optional call handling	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_SndNonCallMsg( )** function sends a non-call related ISDN message to the network over the D channel. The **cc_SndNonCallMsg( )** function specifies the ISDN CRN type as either:

- GLOBAL CRN - pertaining to all calls or channels on a trunk
- NULL CRN - not related to any particular call

Unlike the **cc_SndMsg( )** function, the **cc_SndNonCallMsg( )** function does not require a call reference number (CRN) to transmit the outgoing message.

Parameter	Description
boarddev:	The board device handle.
crn_type:	Specifies one of the following ISDN CRN types: • GLOBAL_CRN • NULL_CRN
msg_type:	Specifies one of the following ISDN message types: • SndMsg_Facility • SndMsg_Notify • SndMsg_FacilityACK • SndMsg_ServiceACK • SndMsg_FacilityREJ • SndMsg_Status • SndMsg_Information • SndMsg_StatusEnquiry The values for **msg_type** are defined in *cclib.h*.
sndmsgptr:	Pointer to the NONCRN_BLK data structure that contains the information related to the GLOBAL or NULL CRN event. For a description of the data structure, see *Section 6.9. NONCRN_BLK*. See the Example code below for details.

■ Cautions

Some IEs may require a Call Reference Value (CRV) to be part of the contents. The Call Reference, in this case, must be the Call Reference value assigned by the network, not the Call Reference Number (CRN) that is assigned by Dialogic and retrieved using the **cc_GetCRN( )** function. It is up to the application to correctly format and order the IEs. Refer to the ISDN Recommendation Q.931 or the switch specification of the application's ISDN protocol for the relevant CCITT format. See the Example code for details.

■ Example

```
/* This Example uses data specific to a 5ESS (non-Euro ISDN) switch */

#include <windows.h>    /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
```

```
{
   LINEDEV       devhdl    = 0;
   char          *devname = "briS1";  /* device name for BRI board 1 station 1 */
   NONCRN_BLK    ie_blk;              /* IE block to transmit custom data */
   .
   .
   .
  /* open the ISDN board device */
  if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

  /* Initialize the values to transmit.  For this example, the ISDN INFORMATION message
     will be transmitted with a Keypad IE.
  */

   ie_blk->sapi     = 0;     /* Call Control SAPI */
   ie_blk->ces      = 1;     /* Always one for User side */
   ie_blk->length   = 3;     /* 3 bytes of data */
   ie_blk->data[0]  = 0x2c;  /* IE ID = Keypad */
   ie_blk->data[1]  = 0x01;  /* Length = 1 */
   ie_blk->data[2]  = 0x35;  /* Keypad value = 0x35 (digit 5) */

   /* send the specified NULL/Dummy CRN ISDN INFORMATION message */

   if ( cc_SndNonCallMsg(devhdl, NULL_CRN, SndMsg_Information,(NONCRN_BLK *)&ie_blk) < 0
)
        procdevfail(devhdl);

   /* continue with call processing */
        .
        .
        .
} /* end main */


int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Possible error codes from the **cc_SndNonCallMsg( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_BADSTATE	Bad state
ERR_ISDN_LIB | E_ISBADTS	Bad time slot

See Also

- **cc_GetNonCallMsg( )**
- **cc_SndMsg( )**

Name:	int cc_StartTrace(boarddev, FileName)	
Inputs:	LINEDEV boarddev	• board device handle controlling the D channel
	char *FileName	• file name for the trace
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	System tools	
Mode:	synchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_StartTrace( )** function instructs the firmware to start the capture of all D channel information into a specified log file. The firmware traces the D channel communications between the Dialogic board and the network, places the results in the shared RAM, and opens a file under the FileName parameter. The results are then placed in the specified file, which is stored on the Dialogic board.

The **cc_StartTrace( )** function allows the application to trace ISDN messages on the specified D channel. The saved trace file is interpreted off line by the *isdtrace* utility program supplied with the release package. (For more on the *isdtrace* utility program, see *Section 8.5.2. ISDTRACE Utility*.) The trace continues until **cc_StopTrace( )** is issued. Complete information on the trace is not available until the **cc_StopTrace( )** function is executed.

Parameter	Description
boarddev:	The board device handle of the ISDN span that contains the D channel.
FileName:	The file name for the trace.

■ Cautions

- The device handle must use the line device handle for the D channel board.
- In order to process trace information, a board-level handle must be used.
- Only one board may be traced at a time.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include “srllib.h”
#include “dtilib.h”
#include “cclib.h”

void main()
{
   LINEDEV   brddevhdl;  /* device handle for D channel */
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = “dtiB1”;
   .
   .
   .
   if ( cc_Open(brddevhdl, devname, 0 ) < 0 )
       exit(1);

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf(“Error opening device: errno = %d\n”, errno);
      exit(1);
   }
   /*
    using cc_StartTrace() to begin
    the trace function and save it on file “istrace.log”.
    Note that the brddevhdl is a board level device.
    */

   if ( cc_StartTrace(brddevhdl, “istrace.log”) < 0 )
        procdevfail(brddevhdl);

   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);
   /*
     using cc_StopTrace() to stop
     the trace function and close the file.
     Note that the brddevhdl is a board level device.
    */
   if ( cc_StopTrace(brddevhdl) < 0 )
        procdevfail(brddevhdl);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);
```

```
   if ( cc_Close(brddevhdl) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_StartTrace( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISTNACT	Trace information is not/is already activated
ERR_ISDN_LIB | E_ISBADIF	Bad interface number

■ See Also

- **cc_StopTrace( )**

Name:	int cc_StopTone(devHdl, mode)	
Inputs:	int devHdl	• channel device handle
	int mode	• asynchronous/synchronous
Returns:	0 on success	
	<0 on failure	
Includes:	cclib.h	
Category:	Global Tone Generation	
Mode:	asynchronous/synchronous	
Technology:	BRI/SC	

■ Description

The **cc_StopTone( )** function forces the termination of a tone that is currently playing on a channel. The function forces a channel that is in the playing state to become idle. Running the **cc_StopTone( )** function asynchronously initiates the function without affecting processes on other channels. Running this function synchronously within a process does not block other processing, allowing other processes to continue to be serviced.

The **cc_StopTone( )** function allows the application to stop the playing of user-defined tones only. This command cannot be used to stop the playing of the firmware-applied call progress tones. The firmware-applied call progress tones and user-defined tones operate independently, except that when the firmware is playing a call progress tone, the application may not play a user-defined tone on the same channel at the same time.

Parameter	Description
devHdl:	Specifies the channel device handle that was obtained when the channel was opened using **cc_Open( )**.
mode:	Specifies whether to run the function asynchronously (EV_ASYNC) or synchronously (EV_SYNC).

■ Termination Events

- CCEV_STOPTONE - indicates that the tone was successfully stopped and the channel was returned to the idle state.
- CCEV_STOPTONEFAIL - indicates that the request to stop a tone and return the channel to the idle state failed.

Use the SRL Event Management functions to handle the termination event.

■ Cautions

- If an I/O function terminates due to another reason before the **cc_StopTone( )** function is issued, the reason for termination will not indicate that **cc_StopTone( )** was called.
- In asynchronous mode, if the application tries to stop a tone that is already stopped, the CCEV_STOPTONEFAIL termination event will be received. Using the **cc_ResultMsg( )** function will retrieve the error code ERR_TONESTOP.
- In synchronous mode, if the application tries to stop a tone that is already stopped, the function will fail. Using the **cc_ResultMsg( )** function will retrieve the error code ERR_TONESTOP.
- When calling **cc_StopTone( )** from a signal handler, the **mode** parameter must be set to EV_ASYNC.
- This function is not supported for the BRI/2 board or PRI protocols.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <srllib.h>
#include <dxxxlib.h>
#include <cclib.h>

main()
{
   toneParm     ToneParm;
   DV_TPT       tpt;
   int          devHdl;
   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( cc_Open(&devHdl, "briS1T1", 0) ) < 0 ) {
      printf( "Error opening device : errno < %d\n", errno );
      exit( 1 );
   }
    ToneParm.freq1 = 330;
    ToneParm.freq2 = 460;
    ToneParm.amp1 = -10;
    ToneParm.amp2 = -10;
    ToneParm.toneOn1 = 400;
    ToneParm.toneOff1 = 0;
    ToneParm.duration = -1;

tpt.tp_type = IO_EOT;
tpt.tp_termno = DX_MAXTIME;
tpt.tp_length = 6000;
tpt.tp_flags = TF_MAXTIME;
```

```
   /*
    * Play a Tone with its
    * Frequency1 = 330, Frequency2 = 460
    * Amplitude at -10dB for both
    * Duration of infinity
 * This is a Panasonic Local Dial Tone.
    *
    */


    if (cc_PlayTone( devHdl, &ToneParm, &tpt, EV_SYNC ) == -1 ){
      printf( "Unable to Play the Tone\n" );
      printf( "Lasterror = %d  Err Msg = %s\n",
          cc_CauseValue( devHdl ), cc_ResultMsg( devHdl ) );
      cc_Close( devHdl);
      exit( 1 );
   }
       .
       .
       .
       .

   /* Force the channel idle.  The I/O function that the channel is
    * executing will be terminated.
    * In the asynchronous mode, cc_StopTone() returns immediately,
    * without waiting for the channel to go idle.
    */
   if ( cc_StopTone(devHdl, EV_ASYNC) == -1) {
      /* process error */
   }
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_StopTone( )** function include the following:

Error Code	Description
ERR_TONEINVALIDMSG	Invalid message type
ERR_TONEBUSY	Busy executing previous command
ERR_TONECP	System error with CP
ERR_TONEDSP	System error with DSP
ERR_TONECHANNELID	Invalid channel ID
ERR_TONESTOP	Tone has already stopped playing on this channel

See Also

- **cc_PlayTone( )**
- **cc_ToneRedefine( )**

Name:	int cc_StopTrace(boarddev)
Inputs:	LINEDEV boarddev • board device handle
Returns:	0 on success < 0 on failure
Includes:	cclib.h
Category:	System control
Mode:	asynchronous
Technology:	BRI/2; BRI/SC; PRI (all protocols)

■ Description

The **cc_StopTrace( )** function stops the trace that was started using the **cc_StartTrace( )** function, discards the trace information, and closes the file.

Parameter	Description
boarddev:	The line device handle of the D channel board.

■ Cautions

None

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   brddevhdl;  /* device handle for D channel */
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1";
   .
   .
   .
   if ( cc_Open(brddevhdl,devname, 0 ) < 0 )
       exit(1);

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
```

```
   }
   /*
      using cc_StartTrace() to begin
      the trace function and save it on file “istrace.log”.
      Note that the brddevhdl is a board level device.
    */

   if ( cc_StartTrace(brddevhdl,”istrace.log”) < 0 )
        procdevfail(brddevhdl);

   printf(“Waiting for call\n”);
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);
   /*
      using cc_StopTrace() to stop
      the trace function and close the file.
      Note that the brddevhdl is a board level device.
    */
   if ( cc_StopTrace(brddevhdl) < 0 )
        procdevfail(brddevhdl);

   /* Close the device */
   if ( cc_Close( devhdl ) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);

   if ( cc_Close(brddevhdl) < 0 )
      printf(“Error closing device, errno = %d\n”, errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf(“reason = %x - %s\n”,reason,msg);
}
```

Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_StopTrace( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISTNACT	Trace information is not/is already activated
ERR_ISDN_LIB | E_ISBADIF	Bad interface number

See Also

- **cc_StartTrace( )**

Name:	int cc_TermRegisterResponse(linedev, data_blk_ptr)	
Inputs:	LINEDEV linedev	• board device handle
	TERM_BLK data_blk_ptr	• pointer to the terminal initialization block data to be sent to the firmware
Returns:	0 on success < 0 on failure	
Includes:	isdncmd.h	
Category:	System tools	
Mode:	asynchronous	
Technology:	BRI/SC	

■ Description

The **cc_TermRegisterResponse( )** function sends a response for CCEV_TERM_REGISTER events. The application, when used as the Network side, receives this event as notification of a TE registration request. On receiving the CCEV_TERM_REGISTER event, the application evaluates the Service Profile Interface ID (SPID) received and either rejects or accepts the registration request. The application then conveys its result to the network using the **cc_TermRegisterResponse( )** function to send either the CCEV_RCVTERMREG_ACK event, if the request is accepted, or the CCEV_RCVTERMREG_NACK event, if the request is rejected. If the request is accepted, the terminal is then fully initialized.

Refer to the Call Scenarios in *Appendix A* for the sequence of events and function calls required for BRI North American terminal initialization. For additional information, refer to the *North American BRI Terminal Initialization with Dialogic Products* application note; the note can be downloaded from the Application Notes section of the Dialogic FirstCall Info Server website: *http://support.dialogic.com*

Parameter	Description
linedev:	The line device handle of the D channel Board.
data_blk_ptr:	Pointer to the data block TERM_BLK to be sent to the firmware. For a description of the TERM_BLK structure, see *Section 6.12. TERM_BLK*.

The data associated with the terminal initialization events can be retrieved using **sr_getevtdatap( )** and casting the value to data types associated with the received event. The types of data structures that are used to cast the event-associated data are provided in *Table 28* below.

Table 28. Terminal Initialization Events and Data Structures

Event	Type of Data Structure
CCEV_TERM_REGISTER	SPID_BLK (See *Section 6.11. SPID_BLK* for a description of this structure.)
CCEV_RCVTERMREG_ACK	USPID_BLK (See *Section 6.15. USPID_BLK* for a description of this structure.)
CCEV_RCVTERMREG_NACK	TERM_NACK_BLK (See *Section 6.13. TERM_NACK_BLK* for a description of this structure.)

NOTE: The North American terminal initialization events all occur on a board level device, not a channel device. The handlers for these events must be registered on the board level device. Refer to the *Voice Software Reference – Standard Runtime Library* for information on the **sr_enblhdlr( )** function.

■ Cautions

- The **cc_TermRegisterResponse( )** function applies only to BRI North American terminal protocols used as Network side.
- This function is not supported for the BRI/2 board.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

/* Global variables */
```

```
LINEDEV          lbuf;
DCHAN_CFG        dchan_cfg;
main ()
{

   dchan_cfg.layer2_access         = FULL_ISDN_STACK;     /* full protocol */
   dchan_cfg.switch_type           = ISDN_BRI_NI1;        /* NI1 switch */
   dchan_cfg.switch_side           = NETWORK_SIDE;        /* Network Terminal */
   dchan_cfg.number_of_endpoints   = 1;                   /* one terminal */
   dchan_cfg.user.tei_assignment   = FIXED_TEI_TERMINAL;  /* Fixed TEI terminal */
   dchan_cfg.user.fixed_tei_value  = 23;                  /* TEI assigned to terminal */
   dchan_cfg.tmr.te.T303 = TMR_DEFAULT; /* NOTE: the values chosen are arbitrary. */
   dchan_cfg.tmr.te.T304 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T305 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T308 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T310 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T313 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T318 = TMR_DEFAULT;
   dchan_cfg.tmr.te.T319 = TMR_DEFAULT;


   if (cc_Open(&lbuf, briS1,0) != SUCCESS)
   {
      printf("cc_open: error\n");
   }

   if (cc_SetDChanCfg(lbuf, &dchan_cfg) == SUCCESS)
   {
      printf("Configuration is set\n");
   }
   else
      printf("Configuration could not be set\n");
.
   .
   /* Initialize SRL */
   .
   .
   .
   /* enable termRegisterHdlr() to handler the CCEV_TERM_REGISTER event */

    if ( sr_enbhdlr( devhdl, CCEV_TERM_REGISTER, termRegisterHdlr ) < 0 )
    {
        printf( "Handler enable for CCEV_TERM_REGISTER event failed:  %s\n",
                          ATDV_ERRMSGP( lbuf ) );
        return( 1 );
    }
   .
   .
   .

   /*
      Wait for Link Activation confirmation
      After which call processing can be started.
    */
   .
   .
   .
   sr_waitevt(-1)

}
```

```
/*********************************************/
/* termRegisterHdlr - Term Register Handler    */
/*********************************************/
int termRegisterHdlr( )
{
            int devindx;
            int dev = sr_getevtdev();
            int len = sr_getevtlen();
            void *datap = sr_getevtdatap();
            char *message;
            int  dchstate;
            int x, y;
            TERM_BLK  termBlk;
            SPID_BLK *spidBlk;

/* Extract the event data associated with the CCEV_TERM_RSTER
 * Event.  Note that the data in the event data pointer is directlast
 * to type SPID_BLK in order to extract the SPID number,PI,
 * and CES values.
 */
            spidBlk = (SPID_BLK *) datap;
            printf("SPID: %S", spidBlk->SPID);

            termBlk.data_link.sapi = spidBlk->data_link.sapi;
            termBlk.data_link.ces  = spidBlk->data_link.ces;

/*          ......
 *  at this point, the application reads the SPID value and detnes
 *  whether or not the value of the SPID is valid for a designated sce.
 *  If the SPID is valid, then the application will send a PIVE
 *  acknowledgement (ISDN_OK).  Otherwise a negative acknowledgement
 *  (ISDN_ERROR) is sent.  This example shows POSITIVE acknowledgement.
 *
 */

/* if sending a positive acknowledgement, the set ack_type = ISDN_OK */
            termBlk.ack_type = ISDN_OK;
            termBlk.ack_info.uspid.usid = 0xA;
            termBlk.ack_info.uspid.tid = 0x0;
            termBlk.ack_info.uspid.interpreter = 1;

/* send out the TERMINAL REGISTRATION acknowledgement */
            if (cc_TermRegisterResponse(dtidev, &termBlk) != 0)
                        printf("Term Reg Ack Error\n");
            else printf("TermRegisterResponse successful\n");
```

■ Errors

If the function returns a value < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_TermRegisterResponse( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISNOMEM	Cannot map or allocate memory

See Also

- **cc_SetEvtMsk( )**

Name:	int cc_ToneRedefine(devHdl, sigType, pToneParm, mode)	
Inputs:	LINEDEV devHdl	• line device handle
	unsigned char sigType	• Tone Signal Type
	toneParm *pToneParm	• pointer to Tone Parameter structure
	int mode	• asynchronous/synchronous
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Global Tone Generation	
Mode:	asynchronous/synchronous	
Technology:	BRI/SC	

■ Description

The **cc_ToneRedefine( )** function redefines a call progress tone's attributes in the tone template table. The tone template table resides in the firmware and is used during call establishment. The template contains common call progress tone types and is preset to default values at initialization (see *Table 29. Tone Template Table*). The current template has a total of eight entries, of which only four are defined. The other four are reserved for future use.

The **cc_ToneRedefine( )** function allows the existing tone template to be redefined, but not the functional meanings of the call progress tones.

Parameter	Description
devHdl:	Specifies the valid channel device handle obtained when the channel was opened using **cc_Open( )**. Each channel has an internal tone template.
sigType:	Indicates the type of call progress tone, such as dial tone, busy tone, ringback, etc. Note that each **sigType** has its own meaning and cannot be changed, for example, **sigType** 0x01 always means a dial tone.
pToneParm:	Pointer to the tone parameter structure. For a description of the toneParm data structure, see *Section 6.14. ToneParm*.
mode:	Specifies whether to run this function asynchronously or synchronously. Set to either EV_ASYNC or EV_SYNC.

The following table shows the tone template table that resides in the firmware.

Table 29. Tone Template Table

Sig Type	Meaning	Default values (in ms)								
		duration	**freq 1**	**amp 1**	**freq 2**	**amp 2**	**tone On1**	**tone Off1**	**no use**	**no use**
0x01	Dial tone	-1	350	-14	440	-14	-1	0	n/a	n/a
0x02	Busy tone	-1	480	-14	620	-14	500	500	n/a	n/a
0x03	Re-order	-1	480	-14	620	-14	300	200	n/a	n/a
0x04	Ring-back	-1	440	-14	480	-14	2000	4000	n/a	n/a
0x05	future use									
0x06	future use									
0x07	future use									
0x08	future use									

The default call progress tones are Dial Tone and Ringback Tone; however the Busy Tone or Reorder Tone will be played instead if the application provides the CCITT compatible SIGNAL IE by sending either the SETUP_ACK or ALERTING message (see the **cc_SetInfoElem( )** function description for information on setting the SIGNAL IE). The SIGNAL IE must be programmed according to the ITU specifications. The firmware will correlate the specified signal, in the SIGNAL IE, with the appropriate **sigType** from the template table.

■ Termination Events

- CCEV_TONEREDEFINE - indicates that the tone was successfully redefined
- CCEV_TONEREDEFINEFAIL - indicates that the function failed

Use the SRL Event Management functions to handle the termination event.

■ Cautions

This function is not supported for the BRI/2 board or for PRI protocols.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <cclib.h>
#include <srllib.h>
#include <dxxxlib.h>

main()
{
    unsigned short sigType = 0x01;
    int            devHdl;
    toneParm       ToneParm;
   /*
    * Open the Voice Channel Device and Enable a Handler
    */
   if ( ( cc_Open(&devHdl, "briS1T1", 0) ) < 0 ) {
      printf( "Error opening device : errno < %d\n", errno );
      exit( 1 );
   }
    ToneParm.freq1 = 330;
    ToneParm.freq2 = 460;
    ToneParm.amp1 = -10;
    ToneParm.amp2 = -10;
    ToneParm.toneOn1 = 400;
    ToneParm.toneOff1 = 0;
    ToneParm.duration = -1;

   /*
    * Redefine a dial tone to the internal template.
    * This template has Frequency1 = 330,
    * Frequency2 = 460, amplitude at -10dB for
    * both frequencies and duration of infinity
* This is a Panasonic Local Dial Tone.
*/
   cc_ToneRedefine(devHdl, sigType, &ToneParm, EV_SYNC );
   /*
    * Continue Processing
    *   .
    *   .
    *   .
    */
   /*
    * Close the opened Voice Channel Device
```

```
    */
   if ( cc_Close( devhdl ) != 0 ) {
      printf( "Error closing devicd, errno= %d\n", errno );
   }
   /* Terminate the Program */
   exit( 0 );
}
```

The following example illustrates how to override the default tone that is played with an outgoing message. The example shows how the application uses the ALERTING message to direct the firmware to play the Busy tone instead of the default Ringback tone.

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include <cclib.h>
#include <srllib.h>
#include <dxxxlib.h>
main()
{
    LINEDEV      boardDevHdl, tsDevHdl;
    DCHAN_CFG    dchan_cfg;
    IE_BLK       ie_buf;
   /*
    * Open the BRI Station Device
    */
   if ( ( cc_Open(&boardDevHdl, "briS1", 0) ) < 0 ) {
      printf( "Error opening board device : errno < %d\n", errno );
      exit( 1 );
   }
   /*
    * Open the BRI Channel Device
    */
   if ( ( cc_Open(&tsDevHdl, "briS1T1", 0) ) < 0 ) {
      printf("Error opening timeslot device : errno < %d\n",errno);
      exit( 1 );
   }

   /*
    * Configure the BRI station.
    */
   dchan_cfg.layer2_access       = FULL_ISDN_STACK;
   dchan_cfg.switch_type         = ISDN_BRI_NI1;
   dchan_cfg.switch_side         = NETWORK_SIDE;
   dchan_cfg.number_of_endpoints = 1;
   dchan_cfg.feature_controlA = DEFAULT_PCM_TONE |
                                /* Want firmware to apply tones */
                                HOST_CONTROLLED_RELEASE;
            /* App. controls when B channel is fully released. */
   /*
    * Continue Processing
    *   .
    *   .
    */

   /*
    * Use cc_SetInfoElem() to specify the SIGNAL IE,
    * that is to be sent with the
    * Alerting message. Firmware will provide the tone,
    * corresponding to what is in the
    * SIGNAL IE, providing it is available in the tone template
```

```
    * table.
    */
   ie_buf.length = 3;
   ie_buf.data[0] = 0x34;    /* Signal IE ID */
   ie_buf.data[1] = 0x01;    /* Length of data in Signal IE */
   ie_buf.data[2] = 0x04;    /* Q.931 definition for Busy Tone On
                                signal value */

   if (cc_SetInfoElem( tsDevHdl, &ie_buf) < 0) {
      printf( "Error setting IE data : errno < %d\n", errno );
      exit( 1 );
   }

   if (cc_AcceptCall( tsDevHdl, 0, EV_ASYNC) < 0) {
      printf( "Error sending Alerting message : errno < %d\n",
              errno );
      exit( 1 );
   }
   /* Alerting message will generate a Busy tone, rather than a
      Ringback tone. */

   /*
    * Continue Processing
    *   .
    *   .
    */

   /*
    * Close the opened timeslot Device
    */
   if ( cc_Close( tsDevhdl ) != 0 ) {
      printf( "Error closing devicd, errno= %d\n", errno );
   }
   /*
    * Close the opened board Device
    */
   if ( cc_Close( boardDevhdl ) != 0 ) {
      printf( "Error closing devicd, errno= %d\n", errno );
   }
   /* Terminate the Program */
   exit( 0 );
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_ToneRedefine( )** function include the following:

Error Code	Description
ERR_TONEINVALIDMSG	Invalid message type
ERR_TONESIGNALTYPE	Invalid signal type

ERR_TONEFREQ1	Invalid value specified in parameter freq 1
ERR_TONEFREQ2	Invalid value specified in parameter freq 2
ERR_TONEAMP1	Invalid value specified in parameter amp1
ERR_TONEAMP2	Invalid value specified in parameter amp2
ERR_TONEON1	Invalid value specified in parameter toneOn1
ERR_TONEOFF1	Invalid value specified in parameter toneOff1
ERR_DURATION	Invalid value specified in parameter duration

■ See Also

- **cc_PlayTone( )**

Name:	int cc_WaitCall(linedev, crnptr, waitcall_blkp, timeout, mode)	
Inputs:	LINEDEV linedev	• device handle
	CRN *crnptr	• pointer to call reference number
	WAITCALL_BLK *waitcall_blkp	• for future use
	int timeout	• amount of time to wait for calls
	unsigned long mode	• synchronous or asynchronous
Returns:	0 on success < 0 on failure	
Includes:	cclib.h	
Category:	Call control	
Mode:	synchronous or asynchronous	
Technology:	BRI/2; BRI/SC; PRI (all protocols)	

■ Description

The **cc_WaitCall( )** function sets up conditions for processing an incoming call. This function also sets up the buffer location where the incoming call's information is stored and unblocks the time slot to allow for notification of other incoming calls

Parameter	Description
linedev:	The line device handle or, for the call waiting feature, the board device handle (BRI and Windows only). When a board device is specified, all subsequent CCEV_OFFERED events will be for that board device.
crnptr:	Pointer to where the call reference number will be stored. (The **crnptr** parameter is used in synchronous mode only.)
waitcall_blkp:	For future use. Set to NULL.
timeout:	The time, in seconds, that the application will wait for the call. If the timeout is 0, a value < 0 will return unless the incoming call is pending. The **timeout** parameter is used in synchronous mode only.
mode:	Specifies asynchronous (EV_ASYNC) or synchronous

Parameter	Description
	(EV_SYNC) mode.

When and how the **cc_WaitCall( )** function is issued depends on whether the function is called in synchronous or asynchronous mode.

In synchronous mode, the **cc_WaitCall( )** function cannot be stopped until **timeout** expires or until **cc_Restart( )** is called from another process. The parameter **crnptr** will be assigned when **cc_WaitCall( )** is terminated by the event CCEV_OFFERED. If the **cc_WaitCall( )** function fails, the call (and the call reference number) will be released automatically.

When **cc_WaitCall( )** is called in asynchronous mode, **cc_ReleaseCall( )** will **not** block the incoming call notification. The application needs to issue **cc_WaitCall( )** only once per line device. However, if **cc_Restart( )** is called, the application must reissue **cc_WaitCall( )**.

The call reference parameter (**crnptr**) is not used in the **cc_WaitCall( )** function in asynchronous mode. The application must retrieve the call reference number by using the **cc_GetCRN( )** function when the call notification event, CCEV_OFFERED, is received.

■ Termination Events

- CCEV_OFFERED - indicates that the setup message has been received by the application.
- CCEV_TASKFAIL - indicates that a request/message was rejected by the firmware. The application can use **cc_Restart( )** after the event is received to reset the channel.

■ Cautions

- Call Waiting Feature (BRI and Windows only):
    - The **cc_WaitCall( )** function must be used in asynchronous mode to support the call waiting feature.
    - A board device must be specified in **linedev** to allow the application to receive incoming waiting calls.

  - To receive incoming waiting calls, applications using the NTT protocol must use the **cc_SetInfoElem( )** function to override the default Channel ID IE content. This ensures that the application complies with the NTT protocol specification requiring the first reply to the SETUP message to specify the B channel on which the call will proceed.

- The application should always release the call reference number after the termination of a connection by calling **cc_ReleaseCall( )**. Failure to do so may cause memory allocation errors.
- In asynchronous mode, the CRN will not be available until an incoming call has arrived. In synchronous mode, the CRN will not be available until the **cc_WaitCall( )** function completes successfully.
- The structures **crnptr** and **waitcall_blkp** should be globally allocated.

■ Example

```
#include <windows.h>   /* For Windows applications only */
#include <stdio.h>
#include <errno.h>
#include "srllib.h"
#include "dtilib.h"
#include "cclib.h"

void main()
{
   LINEDEV   devhdl    = 0;
   CRN       crn = 0;
   char     *devname = "dtiB1T1";

   if ( cc_Open( &devhdl, devname, 0 ) < 0 )
   {
      printf("Error opening device: errno = %d\n", errno);
      exit(1);
   }

   printf("Waiting for call\n");
   if ( cc_WaitCall(devhdl, &crn, NULL, -1, EV_SYNC) < 0 )
        procdevfail(devhdl);

   if ( cc_AnswerCall(crn, 0, EV_SYNC) < 0 )
        callfail(crn);

        .
        .
        .
        .
        .
   /* Drop the call */
   if ( cc_DropCall(crn, NORMAL_CLEARING, EV_SYNC) < 0 )
        callfail(crn);

   if ( cc_ReleaseCall(crn) < 0 )
        callfail(crn);
```

```
   if ( cc_Close( devhdl ) < 0 )
      printf("Error closing device, errno = %d\n", errno);

}

int callfail(CRN crn)
{
        LINEDEV ld;
         cc_CRN2LineDev(crn,&ld);
         procdevfail(ld);
}

int procdevfail(LINEDEV handle)
{
        int reason;
        char *msg;
        reason = cc_CauseValue(handle);
        cc_ResultMsg(handle,reason,&msg);
        printf("reason = %x - %s\n",reason,msg);
}
```

■ Errors

If the function returns < 0 to indicate failure, use the **cc_CauseValue( )** function to retrieve the reason code for the failure. The **cc_ResultMsg( )** function can be used to interpret the reason code. Error codes are defined in the files *ccerr.h*, *isdnerr.h*, and *isdncmd.h*.

Error codes from the **cc_WaitCall( )** function include the following:

Error Code	Description
ERR_ISDN_LIB | E_ISBADPAR	Bad input parameter
ERR_ISDN_LIB | E_ISBADTS	Bad time slot
ERR_ISDN_LIB | E_ISBADIF	Bad interface number
ERR_ISDN_LIB | E_ISBADCRN	Bad call reference number
ERR_ISDN_LIB | E_ISNULLPTR	Null pointer error

■ See Also

- **cc_DropCall( )**
- **cc_MakeCall( )**
- **cc_ReleaseCall( )**

6. Data Structure Reference

The data structures used by selected ISDN functions are described in this chapter. These structures are used to control the operation of functions and to return information. The data structures defined include:

- CC_RATE_U
- channel_id
- DCHAN_CFG
- DLINK
- DLINK_CFG
- IE_BLK
- L2_BLK
- MAKECALL_BLK
- NONCRN_BLK
- PARM_INFO
- SPID_BLK
- TERM_BLK
- TERM_NACK_BLK
- ToneParm
- USPID_BLK
- USRINFO_ELEM
- WAITCALL_BLK

The data structure definition is followed by a table providing a detailed description of the fields in the data structure. These fields are listed in the sequence in which they are defined in the data structure.

6.1. CC_RATE_U

The CC_RATE_U data structure contains billing rate information for Vari-A-Bill services, which is set by the **cc_SetBilling( )** function. The current structure, defined for AT&T only, is shown below:

```
typedef union {
          struct  ATT {long cents}
} CC_RATE_U;
```

Table 30. CC_RATE_U Field Descriptions

Field	Description
ATT	The billing rate for the current call.

6.2. channel_id

The channel_id structure is used within the MAKECALL_BLK structure to specify the channel indicator and the channel indicator mode for the call waiting feature. The channel indicator specifies the Channel resource preference (NO_BCHAN, ANY_BCHAN, DCHAN_IND, or a specific B-Channel number). To initiate a waiting call, the channel indicator must be set to NO_BCHAN. The channel indicator mode (PREFERRED or EXCLUSIVE) should be selected if a B channel is specified.

NOTE: The channel indicator and the channel indicator mode are only used if a board device handle is specified in the function call. If a line device handle is specified, these indicators will be ignored.

The structure is defined as follows:

```
struct {
    byte channel;
    byte channel_mode;
    short rfu;
} channel_id;
```

Table 31. channel_id Descriptions and Values

Field	Description	Values
channel	Specifies the channel to be used.	NO_BCHAN – No B channel ANY_BCHAN – Any B channel DCHAN_IND – Non circuit switched
channel mode	Specifies the channel mode to be used for the B channel, if a B channel is specified in the channel field.	PREFERRED – B channel preferred EXCLUSIVE – B channel exclusive
rfu	Reserved for future use	Not used

6.3. DCHAN_CFG

The DCHAN_CFG data structure contains D-channel configuration block information. The D-channel configuration block sets the configuration of the Digital Subscriber Loop (DSL) for BRI applications. The D-channel is configured using the **cc_SetDChanCfg( )** function.

The structure is defined as follows:

```
typedef struct {
   byte           layer2_access;          /* Layer 2 or full stack */
   byte           switch_type;            /* Layer 3 switch type */
   byte           switch_side;            /* Network or User side */
   byte           number_of_endpoints;    /* # of logical data links */
   byte           feature_controlA;       /* Firmware feature mask A */
   byte           feature_controlB;       /* Firmware feature mask B */
   byte           rfu_1;                  /* Reserved for future use */
   byte           rfu_2;                  /* Reserved for future use */
   struct {

      byte        tei_assignment;         /* Auto assignment or Fixed TEI terminal */
      byte        fixed_tei_value;        /* TEI value if Fixed TEI terminal */
      union {
         struct {
            byte  auto_init_flag;         /* Auto initializing term or not */
            byte  SPID[MAX_SPID_SIZE];    /* SPID for terminal, NULL
```

```
                                                terminated string. */
            byte  rfu_1;
            byte  rfu_2;
         } no_am; /* North America */
    } protocol_specific;
   } user;
  #define RFU_COUNT 8 /* # of reserve for future use bytes */
   byte rfu[RFU_COUNT];

   union {
      struct {
         long     T302;
         long     T303;
         long     T304;
         long     T305;
         long     T306;
         long     T308;
         long     T309;
         long     T310;
         long     T312;
         long     T322;
      } nt;
      struct {
         long     T303;
         long     T304;
         long     T305;
         long     T308;
         long     T310;
         long     T312;
         long     T313;
         long     T318;
         long     T319;
      } te;
   } tmr;
} DCHAN_CFG, *DCHAN_CFG_PTR;
```

The possible values for the DCHAN_CFG structure are listed below. All components of the DCHAN_CFG structure that pertain to the configuration must be set. There are no default values.

NOTE: The **T3xx** values for defining the Layer 3 timer can be used for BRI/2, BRI/SC and PRI protocols. All of the other values in the structure are applicable only to BRI/SC.

Table 32. DCHAN_CFG Field Descriptions and Values

Type	Description	Values

layer2_access	Boolean value used to configure the DSL for direct layer 2 access or for full stack access.	`#define LAYER_2_ONLY 0`
		`#define FULL_ISDN_STACK 1`
		Where:
		• LAYER_2_ONLY = ISDN access at layer 2 (If LAYER_2_ONLY is selected, no other parameters are required)
		• FULL_ISDN_STACK = ISDN access at L3 call control

switch_type	Basic rate protocol (switch type) for DSL. Multiple run-time selectable switch types are available.	`typedef enum {`
		`ISDN_INVALID_SWITCH=0x80,`
		`ISDN_BRI_5ESS,`
		`ISDN_BRI_DMS100,`
		`ISDN_BRI_NTT,`
		`ISDN_BRI_NET3,`
		`ISDN_BRI_NI1,`
		`ISDN_BRI_NI2`
		`} IsdnSwitchType;`
		Where:
		• ISDN_BRI_5ESS = ATT 5ESS BRI
		• ISDN_BRI_DMS100 = Northern Telecom DMS100 BRI
		• ISDN_BRI_NTT = Japanese INS-Net 64 BRI
		• ISDN_BRI_NET3 = EuroISDN BRI
		• ISDN_BRI_NI1 = National ISDN 1
		• ISDN_BRI_NI2 = National ISDN 2

Type	Description	Values

switch_side	Boolean value defining whether the DSL should be configured as the Network side (NT) or the User side (TE).	`#define USER_SIDE 0`
		`#define NETWORK_SIDE 1`
		Where:
		• USER_SIDE = User side of ISDN protocol
		• NETWORK_SIDE = Network side of ISDN protocol

number_of_ endpoints	Number of logical data links to be supported.	1 to MAX_DLINK, where MAX_DLINK is currently set to 8. This field only has significance when configuring the DSL as the NETWORK side.

Type	Description	Values

feature_controlA	Firmware feature control field A. This is a bit mask field for setting features in the firmware.	The following defines are used to configure the firmware features. The lowest two bits provide a combination of four possible settings for the TONE feature.
		`#define NO_PCM_TONE 0x00`
		`#define ULAW_PCM_TONE 0x01`
		`#define ALAW_PCM_TONE 0x02`
		`#define DEFAULT_PCM_TONE 0x03`
		`#define SENDING_COMPLETE_ATTACH 0x04`
		`#define USER_PERST_L2_ACT 0x08`
		`#define HOST_CONTROLLED_RELEASE 0x10`
		Where:
		• NO_PCM_TONE = Disable firmware from providing tones and set default encoding according to switch type
		• ULAW_PCM_TONE = Provide tones and use ULAW encoding for B channel tones
		• ALAW_PCM_TONE = Provide tones and use ALAW encoding for B channel tones
		• DEFAULT_PCM_TONE = Provide tones and use default encoding for B channel tones according to the switch type setting
		• SENDING_COMPLETE_ATTACH = Add Sending Complete IE to SETUP message
		• USER_PERST_L2_ACT = Persistent L2 activation on User side
		• HOST_CONTROLLED_RELEASE = Delay RELEASE reply until host issues **cc_ReleaseCall( )**

feature_controlB	Firmware feature control field. This is a bit mask field for setting features in the firmware.	Currently not used.

Type	Description	Values
rfu_1 & rfu_2	Reserved for future use.	Currently not used.

tei_assignment	Applies to User Side only. It specifies if the terminal has a fixed TEI or an auto-assigning TEI. If it is fixed, then "fixed_tei_value" must be specified (see below).	`#define AUTO_TEI_TERMINAL 0`
		`#define FIXED_TEI_TERMINAL 1`
		Where:
		• AUTO_TEI_TERMINAL = auto TEI assigning Term
		• FIXED_TEI_TERMINAL = Fixed TEI assigning Term

fixed_tei_value	Defines the TEI to be used for a fixed TEI assigning terminal.	0 to 63 (Required when tei_assignment = FIXED_TEI_TERMINAL)

auto_init_flag	Boolean value defining whether or not the terminal is an auto initializing terminal. This field applies only when configuring the DSL as the User side and only to North American protocols.	`#define AUTO_INIT_TERMINAL 0`
		`#define NON_INIT_TERMINAL 1`
		Where:
		• AUTO_INIT_TERMINAL = auto initializing terminal
		• NON_INIT_TERMINAL = non-auto initializing term

Type	Description	Values
SPID	Defines the assigned Service Provider Identifier (SPID) value for terminal initialization. It is only applicable to User side US switches. **NOTE:** When the SPID is set, it is assigned to both bearer channels associated with the D channel. To subsequently modify SPID assignments, use **cc_SetParmEx( )**.	ASCII digit string limited to the digits 0-9 and limited in length to MAX_SPID_SIZE Where: MAX_SPID_SIZE = (20+1) (Required when auto_init_flag = AUTO_INIT_TERMINAL. Most North American switches require a SPID.)
no_am.rfu_1 & rfu_2	Reserved for future use.	Currently not used.
rfu[RFU_COUNT]	Array of fields reserved for future use.	Currently not used.

Type	Description	Values
T3xx (T302, T303, T304, T305, T306, T308, T309, T310, T312, T313, T318, T319, T322)	Defines the Layer 3 timer values. See Q.931 specification and corresponding switch specifications for exact definitions and default values for these timers. Not all timers are applicable to all of the switches.	Specified values are in 10 millisecond increments. For example, a specified value of 100 is equivalent to 1 second. Possible values are: • 0 = Default value for switch • -1 = Default value for switch • 0 < n < -1 = Timer value in tens of milliseconds **NOTE:** Incorrect or unreasonable timer settings will result in undesirable effects to calls as well as the call control stack. Before overriding the default values, users needs to understand the timer meanings and their interdependencies.

6.4. DLINK

The DLINK data structure contains information about the data link information block. The DLINK structure is passed for the **cc_GetDLinkCfg( )**, **cc_SetDLinkCfg( )**, **cc_GetDLinkState( )** and **cc_SetDLinkState( )** functions.

The DLINK structure is used in the following structures:

- SPID_BLK, see *Section 6.11. SPID_BLK*
- TERM_BLK, see *Section 6.12. TERM_BLK*
- TERM_NACK_BLK, see *Section 6.13. TERM_NACK_BLK*
- USPID_BLK, see *Section 6.15. USPID_BLK*

The DLINK structure is defined as follows:

```
typedef   struct
{
    char  sapi;
    char  ces;
}  DLINK,   *DLINK_PTR;
```

Table 33. DLINK Field Descriptions

Field	Description
sapi	The service access pointer identifier. (This field is zero for PRI.)
ces	The connection endpoint suffix. (This field is zero for PRI.)

6.5. DLINK_CFG

The DLINK_CFG structure contains information about the data link logical link configuration block. The DLINK_CFG structure is retrieved using the **cc_GetDLinkCfg( )** function.

The structure is defined as follows:

```
typedef   struct
{
    char  tei;
    int   state;
    int   protocol;
}  DLINK_CFG,   *DLINK_CFG_PTR;
```

Table 34. DLINK_CFG Field Descriptions

Field	Description
tei	Terminal Endpoint Identifier. Valid values are: • 0 - 63 - for manual TEIs (chosen by the user side) • AUTO_TEI - for automatic TEIs (chosen by the network side)
state	The original state in which the logical link should be configured. Valid values are: • DATA_LINK_UP - the firmware will attempt to activate the logical link if it is not already activated and will allow the network side to establish the logical link if requested. • DATA_LINK_DOWN - the firmware will attempt to release the logical link if it is currently established. The firmware will allow

Field	Description
	the network side to establish the logical link if requested. • DATA_LINK_DISABLED - the firmware will attempt to release the logical link if it is currently established. The firmware will **not** allow the network side to establish the logical link if requested.
protocol	The protocol to be used on this logical link. For instance: • DATA_LINK_PROTOCOL_Q931 - indicates that the link is to be used as an ISDN connection-oriented logical link. • DATA_LINK_PROTOCOL_X25 - indicates that the link is to be used as an X.25 packet-switched link.

6.6. IE_BLK

The IE_BLK structure contains information elements (IEs) to be sent to the network using the **cc_SndMsg( )** or **cc_SetInfoElem( )** function, or the information elements to be retrieved using the **cc_GetInfoElem( )** **cc_GetSigInfo( )** function.

The structure is defined as follows:

```
#define MAXLEN_IEDATA  254
typedef struct
  {
    short length;
    char  data[MAXLEN_IEDATA];
  } IE_BLK, *IE_BLK_PTR;
```

Table 35. IE_BLK Field Descriptions

Field	Description
length	The total bytes in the data field.
data	This field contains the information element(s) to be sent.

6.7. L2_BLK

The L2_BLK structure is used to send and receive layer 2 information to an ISDN interface using the **cc_GetFrame( )** and **cc_SndFrame( )** functions.

The structure is defined as follows:

```
#define MAXLEN_IEDATA     254
typedef struct
  {
   char   sapi;
   char   ces;
   short  length;
   char   data[MAXLEN_DATA];
   }  L2_BLK, *L2_BLK_PTR;
```

Table 36. L2_BLK Field Descriptions

Field	Description
sapi	The Service Access Point Identifier (applies to BRI only, set to 0 for PRI).
ces	The Connection Endpoint Suffix (applies to BRI only, set to 0 for PRI).
length	The total bytes of the following data field.
data	This field contains the received frame.

6.8. MAKECALL_BLK

The MAKECALL_BLK structure contains the parameters used to specify an outgoing call. Outgoing calls are made using the **cc_MakeCall( )** function. The parameters for outgoing calls are set when the MAKECALL Block function is initialized. For information on initializing the MAKECALL Block, see *Section 6.8.1. MAKECALL_BLK Initialization*.

The structure is defined as follows:

```
struct is {
    unsigned char BC_xfer_cap;
    unsigned char BC_xfer_mode;
```

```
    unsigned char BC_xfer_rate;
    unsigned char usrinfo_layer1_protocol;
    unsigned char usr_rate;
    unsigned char destination_number_type;
    unsigned char destination_number_plan;
    unsigned char destination_sub_number_type;
    unsigned char destination_sub_number_plan;
    char destination_sub_phone_number[MAXPHONENUM];
    unsigned char origination_number_type;
    unsigned char origination_number_plan;
    char origination_phone_number[MAXPHONENUM];
    unsigned char origination_sub_number_type;
    unsigned char origination_sub_number_plan;
    char origination_sub_phone_number[MAXPHONENUM];
    unsigned char facility_feature_service;
    unsigned char facility_coding_value;
    int completion_point;

    union {
      struct {
        int rfu;
      } att;
      struct {
        CHAN_ID channel_id;
      } bri;
      struct {
        int rfu;
      } vn3;
      struct {
        int rfu;
      } dass;
    }u;          /* switch system dependent parameters */
    USRINFO_ELEM *usrinfo_bufp;
    NFACILITY_ELEM *nsfc_bufp;
  }isdn;
} MAKECALL_BLK,  *MAKECALL_BLK_PTR;
```

The following table provides definitions of the MAKECALL block parameters. All elements in the MAKECALL_BLK structure must be specified. To use the default setting, specify ISDN_NOTUSED.

Table 37. MAKECALL_BLK Parameter ID Definitions

Parameter	Description	Supported values
BC_xfer_cap	Bearer channel information transfer capability.	BEAR_CAP_SPEECH - speech BEAR_CAP_UNREST_DIG - unrestricted digital data BEAR_REST_DIG - restricted digital data **NOTE:** The value specified by **cc_SetParm( )** is used if this element is set to ISDN_NOTUSED.
BC_xfer_mode	Bearer channel information transfer mode	ISDN_ITM_CIRCUIT - circuit switch mode **NOTE:** The value specified by **cc_SetParm( )** is used if this element is set to ISDN_NOTUSED.
BC_xfer_rate	Bearer channel information transfer rate	BEAR_RATE_64KBPS - 64K bps transfer rate **NOTE:** The value specified by c**c_SetParm( )** is used if this element is set to ISDN_NOTUSED.

Parameter	Description	Supported values
usrinfo_layer1_protocol	Layer 1 protocol to use on bearer channel	ISDN_UIL1_CCITTV.110 - CCITT standardized rate adaptation V.110) ISDN_UIL1_G711uLAW - recommendation G.711 u-Law ISDN_UIL1_G711ALAW - recommendation G.711 a-Law ISDN_UIL1_G721ADCPM - recommendation G.721 32 kbits/s ADCPM and recommendation I.460 ISDN_UIL1_G722G725 - recommendation G.772 and G.725 - 7kHz ISDN_UIL1_H261 - recommendation H.262 - 384 kbits/s video ISDN_UIL1_NONCCITT - Non-CCITT standardized rate adaptation ISDN_UIL1_CCITTV120 - CCITT standardize rate adaptation V.120 ISDN_UIL1_CCITTX31 - CCITT standardized rate adaptation X.31HDLC **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.

Parameter	Description	Supported values
usr_rate	User rate to use on bearer channel (layer 1 rate)	ISDN_UR_EINI460 - determined by E bits in I.460 ISDN_UR_56000 - 56 kbits, V.6 ISDN_UR_64000 - 64 kbits, X.1 ISDN_UR_134 - 1345 kbits, X.1 ISDN_UR_12000 - 12 kbits, V.6 **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
destination_number_type	Called party number type	EN_BLOC_NUMBER - number is sent en-block (in whole, not overlap sending) INTL_NUMBER - international number for international call. (verify availability with service provider) NAT_NUMBER - national number for call within national numbering plan (accepted by most networks) LOC_NUMBER - subscriber number for a local call (verify availability with service provider) OVERLAP_NUMBER - overlap sending; number is not sent in whole (not available on all networks) **NOTE:** The value specified by **cc_SetParm( )** is used if this element is set to ISDN_NOTUSED.

Parameter	Description	Supported values
destination_number_plan	Called party number plan	UNKNOWN_NUMB_PLAN unknown new plan ISDN_NUMB_PLAN - ISDN/telephony - E.164/E.163 (accepted by most networks) TELEPHONY_NUMB_PLAN telephony numbering plan - E.164 PRIVATE_NUMB_PLAN - private numbering plan **NOTE:** The value specified by **cc_SetParm( )** is used if this element is set to ISDN_NOTUSED.
destination_sub_number_type	Called party sub-number type	OSI_SUB_ADDR - NSAP - (X.213/ISO 8348 AD2) (not accepted by all networks) USER_SPECIFIC_SUB_ADDR - user specified (not accepted by all networks) IA_5_FORMAT IA5 - sub-address digit format; telephony numbering plan (accepted by most networks) **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
destination_sub_number_plan	Called party sub-number plan	ISDN_NOTUSED (reserved for future use)

Parameter	Description	Supported values
destination_sub_phone_ number [MAXPHONENUM]	Called party sub-phone number	ASCII digit string of MAXPHONENUM. If the first character is set to ISDN_NOTUSED, the number plan and type are ignored and the destination_sub_phone_number IE will not be sent. **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
origination_number_type	Calling party number type	Same values as destination_number_type
origination_number_plan	Calling party number plan	Same values as destination_number_plan
origination_phone_number [MAXPHONENUM]	Calling party phone number - sets the calling party number for this call only.	ASCII digit string of MAXPHONENUM. If the first character is set to ISDN_NOTUSED, the number plan and type are ignored and the origination_phone_number IE will not be sent. Use **cc_SetCallingNum( )** to set the number permanently. **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
origination_sub_number_type	Calling party sub-number type	Same values as destination_sub_number_type
origination_sub_number_plan	Calling party sub-number plan	Same values as destination_sub_number_plan

Parameter	Description	Supported values
origination_sub_phone_ number [MAXPHONENUM]	Calling party sub-phone number	ASCII digit string of MAXPHONENUM. If the first character is set to ISDN_NOTUSED, the number plan and type are ignored and the origination_sub_phone_number IE will not be sent. **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
facility_feature_service	Identifies facility request as a feature or a service (see Note below)	ISDN_FEATURE - request is a facility feature. Features are normally used in the facility message after a call is initiated. Features can also be used in the setup message. ISDN_SERVICE - requested facility is a service. Service can be used at any time in the NSF IE. Service is often used in the setup message to select a specific network service. **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.

Parameter	Description	Supported values
facility_coding_value	Facility coding value; identifies the specific feature or service provided (see Note below)	ISDN_CPN_PREF - calling party number preferred ISDN_SDN - AT&T Software Defined Network ISDN_BN_PREF - billing number preferred ISDN_ACCUNET - AT&T ACCUNET service ISDN_LONG_DIS - long distance service ISDN_INT_800 - international 800 service ISDN_CA_TSC - CA TSC service ISDN_ATT_MULTIQ - AT&T Multiquest 900 service **NOTE:** This element must be set to ISDN_NOTUSED if it is **not** to be used in the setup message.
completion_point	Reserved for future use	Set to ISDN_NOTUSED (reserved for future use)
channel_id	Channel ID structure	See *Section* 6.2. channel_id for a description of the structure and the supported values.
USRINFO_ELEM *usrinfo_bufp	User information element	Not used, set to NULL. See the **cc_SetInfoElem( )** function to add custom IEs to the MAKECALL Block. **NOTE:** This element must be set to NULL if it is **not** to be used in the setup message.

Parameter	Description	Supported values
NFACILITY_ELEM *nsfc_bufp	Network specific facility element	Not used, set to NULL. See the **cc_SetInfoElem( )** function to add custom IEs to the MAKECALL Block. **NOTE:** This element must be set to NULL if it is **not** to be used in the setup message.
NOTE: The facility_feature_service and facility_coding_value data elements must be paired to support the specific feature or service requested from the network. The specific feature/service that is being used must be identified before entering a value for facility_feature_service.		

6.8.1. MAKECALL_BLK Initialization

Because ISDN services vary with switches and provisioning plans, a set of default standards cannot be set for the MAKECALL_BLK. Therefore, it is up to the user application to fill in the applicable MAKECALL_BLK values that apply to the particular provisioning.

All of the bearer capability elements in the MAKECALL_BLK structure must be specified. Specify ISDN_NOTUSED will cause the default values to be sent. The application should first initialize the MAKECALL_BLK structure with a set of defaults prior to filling in settings pertaining to the particular ISDN service.

A sample MAKECALL_BLK initialization is shown below:

```
/* Initialize the MAKECALL block */
            makecall_blk.isdn.BC_xfer_cap = BEAR_CAP_SPEECH;
            makecall_blk.isdn.BC_xfer_mode = ISDN_ITM_CIRCUIT;
            makecall_blk.isdn.BC_xfer_rate = BEAR_RATE_64KBPS;
            makecall_blk.isdn.usrinfo_layer1_protocol = 0xFF;
            makecall_blk.isdn.usr_rate = 0xFF;
            makecall_blk.isdn.destination_number_type = 0xFF;
            makecall_blk.isdn.destination_number_plan =0xFF;
            makecall_blk.isdn.destination_sub_number_type = 0xFF;
            makecall_blk.isdn.destination_sub_phone_number[0] = NULL;
            makecall_blk.isdn.origination_number_type = 0xFF;
            makecall_blk.isdn.origination_number_plan = 0xFF;
            makecall_blk.isdn.origination_sub_number_type = 0xFF;
            makecall_blk.isdn.origination_sub_phone_number[0] = NULL;
            makecall_blk.isdn.origination_phone_number[0] = NULL;
            makecall_blk.isdn.facility_feature_service = 0xFF;
```

```
makecall_blk.isdn.facility_coding_value = 0xFF;
makecall_blk.isdn.usrinfo_bufp = NULL;
makecall_blk.isdn.nsfc_bufp = NULL;
```

For more on the MAKECALL_BLK structure, see the **cc_MakeCall( )** function description in *Chapter 5. ISDN Function Reference*.

6.9. NONCRN_BLK

The NONCRN_BLK structure contains information related to a GLOBAL or NULL call reference number (CRN). The messages are sent to the network using the **SndNonCallMsg( )** function.

The structure is defined as follows:

```
#define MAXLEN_IEDATA  254
typedef struct
  {
    char    sapi;
    char    ces;
    short   length;
    char  data[MAXLEN_IEDATA];
  } NONCRN_BLK, *NONCRN_BLK_PTR;
```

Table 38. NONCRN_BLK Field Descriptions

Field	Description
sapi	The Service Access Point Identifier. For call control procedures, this value is always zero.
ces	The Connection Endpoint Suffix. For the User side, the value is always one. For the Network side, this value is between one and eight.
length	The total bytes in the data field.
data	This field contains the information element(s) to be sent.

6.10. PARM_INFO

The PARM_INFO structure contains parameters that contain variable data that is passed from the firmware. The variable data is retrieved using the **cc_GetParmEx( )** function and set using the **cc_SetParmEx( )** function.

The structure is defined as follows:

```
typedef   struct
   {
   byte parmdatalen;
   unsigned char parmdata [256];  /* maximum length of 256 bytes for ISDN information */
} PARM_INFO, *PARM_INFO_PTR;
```

Table 39. PARM_INFO Field Descriptions

Field	Description
parmdatalen	The total bytes in the data field.
Parmdata	This field contains the variable data to be sent.

6.11. SPID_BLK

The SPID_BLK data structure is used to cast terminal initialization event data after a CCEV_TERM_REGISTER event is received. SPID_BLK contains the value of the Service Profile Interface ID, which is used to determine whether the value is valid for a designated service.

The data structure is defined as follows:

```
/*
 * Structure for CCEV_TERM_REGISTER Event.
 */
typedef struct
{
   DLINK data_link;
   byte  initializing_term;
   byte  SPID[MAX_SPID_SIZE];
} SPID_BLK;
```

Table 40. SPID_BLK Field Descriptions

Field	Description
data_link	The DLINK data structure; see *Section 6.4. DLINK*.
initializing_term	The type of initializing terminal.
SPID	The Service Profile Interface ID

6.12. TERM_BLK

The TERM_BLK data structure contains information regarding the application's response to the CCEV_TERM_REGISTER event. The response is sent using the **cc_TermRegisterResponse( )** function.

The structure is defined as follows:

```
typedef struct
{
   DLINK data_link;
   byte  ack_type;
   union
   {
      byte cause_value; /* Cause Value if ack type is ISDN_ERROR */
      struct
      {
         byte usid;
         byte tid;
         byte interpreter;
      } uspid;
   } ack_info;
} TERM_BLK, *TERM_BLK_PTR;

/* where DATA_LINK contains the following structure */

typedef struct
{
   byte  sapi;
   byte  ces;
} DLINK, *DLINK_PTR;
```

Table 41. TERM_BLK Field Descriptions

Field	Description
data_link	The DLINK data structure; see *Section 6.4. DLINK*.
ack_type	The type of acknowledgement to be passed to the firmware. The settings are: • ISDN_OK - to send a Positive acknowledgment • ISDN_ERROR - to send a Negative acknowledgment
cause_value	The Cause Value defined in *isdncmd.h*. (For a listing of Cause Values, see *Section 7.2.1. Cause/Error Codes from the ISDN Firmware*.)
usid	User Service Identifier. The range is 01 – FF. 00 signifies default.
tid	Terminal Identifier. The range is 01 – 63. 00 signifies that the firmware is to provide a default.
interpreter	Specifies how the **usid** and **tid** values are to be interpreted. Possible value settings are: • 0 = terminal is selected when it matches both the USID and TID • 1 = terminal is selected when it matches the USID but not the TID

6.13. TERM_NACK_BLK

The TERM_NACK_BLK data structure is used to cast terminal initialization event data after a CCEV_RCVTERMREG_NACK event is received. TERM_NACK_BLK contains the cause value for the event, indicating why the terminal initialization request was rejected by the network.

The data structure is defined as follows:

```
/*
 * Structure for CCEV_RCVTERMREG_NACK Event.
 */
typedef struct
{
   DLINK data_link;
   byte  cause_value;
} TERM_NACK_BLK;
```

Table 42. TERM_NACK_BLK Field Descriptions

Field	Description
data_link	The DLINK data structure; see *Section 6.4. DLINK*.
cause_value	The Cause Value defined in *isdncmd.h*. (For a listing of Cause Values associated with a CCEV_RCTERMREG_NACK event, see *Table 43* below..)

The following table lists the possible cause values that may be returned in the TERM_NACK_BLK data structure after receiving a CCEV_RCVTERMREG_NACK event. Any values provided by the Network that are not listed in the table will also be passed up to the application.

Table 43. Cause Values Associated with CCEV_RCVTERMREG_NACK

Cause Value	Q.850 Description	Meaning
0x26	Network out of order	Used when the network has removed the TEI, causing the data link to go down.
0x63	Information element/parameter non-existent or not implemented	Switch does not support endpoint initialization.
0x64	Invalid information element contents	SPID was most likely coded incorrectly.
0x66	Recovery on timer expiry	Application tried two attempts at initialization with no response from the network.
0x6F	Protocol error, unspecified	Used when no cause was given for the rejection.

6.14. ToneParm

The ToneParm data structure is used to redefine a firmware-applied tone's attributes using the **cc_ToneRedefine( )** or to play a user-defined tone using the **cc_PlayTone( )** function.

The data structure is defined as follows:

```
Struct toneParm
{
    uint16    duration;    //1 ~ 65535 (in 10 ms, 0xffff - forever)
    uint16    freq1;       //200 ~ 3100 Hz
    int16     amp1;        //-40 ~ +3 dB
    uint16    freq2;       //200 ~ 3100 Hz
    int16     amp2;        //-40 ~ +3 dB
    uint16    toneOn1;     //1 ~ 65535 (in 10 ms)
    uint16    toneOff1;    //0 ~ 65534 (in 10 ms)
    uint16    reserv1;     //reserved for future use
    uint16    reserv2;     //reserved for future use
}
```

Table 44. ToneParm Field Descriptions

Field	Description
duration	specifies the duration of the tone in 10 ms units. The range is 1 - 65535. Set to -1 to play forever.
freq1	specifies the frequency of the tone. The range is 200 - 3100 Hz.
amp1	specifies the amplitude of the tone. The range is -40 - +3 dB.
freq2	specifies the frequency of the tone. The range is 200-3100 Hz.
amp2	specifies the amplitude of the tone. The range is -40 - +3 dB.
toneOn1	specifies the tone interval, in 10 ms units. The range is 1 - 65535 ms. Set to 1 or greater for continuous tone.
toneOff1	specifies the tone interval, in 10 ms units. The range is 0 - 65534 ms. Set to 0 to play a continuous tone.
reserv1	Reserved for future use
reserv2	Reserved for future use

6.15. USPID_BLK

The USPID_BLK data structure is used to cast terminal initialization event data after a CCEV_RCVTERMREG_ACK event is received. USPID_BLK contains the value of a valid User Service Profile Interface.

The data structure is defined as follows:

```
/* Structure for CCEV_RCVTERMREG_ACK Event /

typedef struct
{
   DLINK data_link;
   struct
   {
      byte usid;
      byte tid;
      byte interpreter;
   } uspid;
} USPID_BLK;
```

Table 45. USPID_BLK Field Descriptions

Field	Description
data_link	The DLINK data structure; see *Section 6.4. DLINK*.
usid	User Service Identifier. The range is 01 – FF. 00 signifies default.
tid	Terminal Identifier. The range is 01 – 63. 00 signifies that the firmware is to provide a default.
interpreter	Specifies how the **usid** and **tid** values are to be interpreted. Possible value settings are: • 0 = terminal is selected when it matches both the USID and TID • 1 = terminal is selected when it matches the USID but not the TID

6.16. USRINFO_ELEM

The USRINFO_ELEM data structure contains the user-to-user information (UUI) data that is retrieved by the **cc_GetCallInfo( )** and **cc_GetSigInfo( )** functions. The content of the user information is application-dependent.

The structure is defined as follows:

```
typedef struct  {
    unsigned char  length;   /* protocol_discriminator + user information length */
    unsigned char  protocol_discriminator;
    char  usrinformation[256];
  } USRINFO_ELEM, *USRINFO_ELEM_PTR;
```

Table 46. USRINFO_ELEM Field Descriptions

Field	Description
length	The length of the user information element
protocol discriminator	The protocol discriminator
usrinformation	The user-to-user information (UUI) data

6.17. WAITCALL_BLK

This structure is reserved for future use. The pointer to the WAITCALL_BLK structure in the argument list for the **cc_WaitCall( )** function must be set to NULL.

7. ISDN Events and Errors

This chapter describes the events and error/cause codes that are returned by the Dialogic ISDN library functions. The function descriptions in *Chapter 5. ISDN Function Reference* list the possible error codes and, for asynchronous functions, the termination events returned by the function.

7.1. Event Categories

There are two types of events returned by the ISDN library functions: those that are returned after the termination of a function call and those that are unsolicited and triggered by external events.

7.1.1. Termination Events

Table 47 lists the events returned at the termination of asynchronous function calls. The events are categorized by call reference number (CRN) referenced functions, line device handle referenced functions, and CRN or line device handle referenced functions. (See *Section 5.3. Function References: CRNs, CRVs, and Line Device* Handles for more on function references.)

NOTE: Termination events are solicited events that are returned by asynchronous functions only.

Table 47. Termination Events

Termination Event	Returned After:	Indicates:	Reference:
CCEV_ACCEPT	**cc_AcceptCall( )**	ALERTING message has been sent to the network.	CRN
CCEV_ANSWERED	**cc_AnswerCall( )**	CONNECT message has been sent to the network.	CRN
CCEV_CONNECTED	**cc_MakeCall( )**	CONNECT message has been received from the network.	CRN
CCEV_DROPCALL	**cc_DropCall( )**	DISCONNECT message has been sent to the network.	CRN
CCEV_HOLDACK	**cc_HoldCall( )**	A HOLD CALL COMPLETED acknowledge message was received by the application (that is, a hold call request was acknowledged by remote equipment, and the call is in HOLD state).	CRN
CCEV_HOLDREJ	**cc_HoldCall( )**	A HOLD CALL REJECT message was received from remote equipment (that is, a call from remote equipment did not enter HOLD state). Use the **cc_GetCallInfo( )** function to retrieve	CRN

Termination Event	Returned After:	Indicates:	Refer-ence:
		information about why the request was rejected.	
CCEV_MOREDIGITS	**cc_GetMoreDigits( )**	Requested number of digits has been received. Use **cc_GetDNIS( )** to retrieve digits.	CRN
CCEV_OFFERED	**cc_WaitCall( )**	SETUP message has been received. Use **cc_GetCallInfo( )** to retrieve information about the event.	CRN
CCEV_PLAYTONE	**cc_PlayTone( )**	User-defined tone successfully played.	Line Device Handle
CCEV_PLAYTONEFAIL	Failure of **cc_PlayTone( )**	Request to play user-defined tone failed.	Line Device Handle
CCEV_RELEASECALL	**cc_ReleaseCallEx( )**	Dialogic ISDN resources were successfully released for the call.	CRN
CCEV_RELEASECALL FAIL	Failure of **cc_ReleaseCallEx( )**	The request to release the resources for the call failed.	CRN
CCEV_REQANI	**cc_ReqANI( )**	ANI has been received from the network. Used only with the AT&T ANI-on-demand feature.	CRN
CCEV_RESTART	**cc_Restart( )**	Restart operation has been completed. The channel is in NULL state.	Line Device Handle

Termination Event	Returned After:	Indicates:	Reference:
CCEV_RESTARTFAIL	Failure of **cc_Restart( )**	Typically, this event is triggered by an incorrect state of a line device. The application may use **cc_ResultValue( )** after this event is received to verify the reason for failure.	Line Device Handle
CCEV_RETRIEVEACK	**cc_RetrieveCall( )**	A RETRIEVE CALL COMPLETE acknowledge message was received from the network. The indicated network has reconnected a held call to a B channel.	CRN
CCEV_RETRIEVEREJ	**cc_RetrieveCall( )**	A RETRIEVE CALL REJECT message was received from the network. The indicated network has rejected a request to reconnect a held call to a B channel. This event can be received only when the application acts as terminal equipment. Use the **cc_GetCallInfo( )** function to retrieve information about why the request was rejected.	CRN
CCEV_SETBILLING	**cc_SetBilling( )**	Billing information	CRN

Termination Event	Returned After:	Indicates:	Refer-ence:
		for the call has been acknowledged by the network. This event is used only with the AT&T Vari-A-Bill feature.	
CCEV_ SETCHANSTATE	**cc_SetChanState( )**	The B channel is placed in the requested state.	Line Device Handle
CCEV_STOPTONE	**cc_StopTone( )**	The tone operation was terminated.	Line Device Handle
CCEV_STOPTONEFAIL	Failure of **cc_StopTone( )**	The request to terminate the playing of a tone failed.	Line Device Handle
CCEV_TASKFAIL	A request or message from the application is rejected by the firmware.	Rejection of a request or message. Normally, this event is triggered by an incorrect function call during the call. This event can also be received due to "race conditions," meaning that the application and the network requested different actions from the firmware at the same time. The application may use **cc_Restart( )** after this event is received to reset the channel.	CRN or Line Device Handle

Termination Event	Returned After:	Indicates:	Refer-ence:
CCEV_TONEREDEFINE	**cc_ToneRedefine( )**	The tone(s) in the firmware tone template table were successfully redefined.	Line Device Handle
CCEV_ TONEREDEFINEFAIL	Failure of **cc_ToneRedefine( )**	The request to redefine tone(s) in the firmware tone template table failed.	Line Device Handle

7.1.2. Unsolicited Events

Table 48 lists the events that are triggered by external signals. Events triggered by external signals are unsolicited events that are returned by both asynchronous and synchronous functions.

Table 48. Unsolicited Events

External Event	Indicates:
CCEV_ALERTING	An ALERTING message has been received by the application, indicating that the connection request has been accepted by the network. By default, the firmware will report this event. The application may clear the CCMSK_ALERT bit to block this event.
CCEV_CALLINFO	An INFORMATION message has been received by the application, for example, in response to a **cc_SndMsg( )** function call in which the **msg_type** specified is SndMsg_Information. Use the **cc_GetCallInfo( )** function to retrieve information about the event.
CCEV_CONFDROP	A DROP request has been received; the request was made by sending the

External Event	Indicates:
	SndMsg_Drop message type via the **cc_SndMsg( )** function. The CCEV_CONFDROP event has two different meanings that depend upon the type of call: Two-party call - the event is a request to disconnect the call. The application should respond by issuing a **cc_DropCall( )**. Conference call - the event is a request to remove the last party that was added to the conference. The application needs to respond to this request with either a SndMsg_DropAck or SndMsg_DropRej message to indicate the acceptance or rejection of the request. If the request is accepted, the party is dropped from the conference. The CCEV_CONFDROP event only pertains to a Custom BRI 5ESS switch type.
CCEV_CONGESTION	A CONGESTION message has been received by the application, indicating that the remote end is not ready to accept incoming user information. Use the **cc_GetCallInfo( )** function to retrieve additional information about the event.
CCEV_D_CHAN_STATUS	The D channel status was changed as a result of the event on the D channel. Refer to the sample code in the **cc_GetSAPI( )** and **cc_GetCES( )** function descriptions for more information.
CCEV_DISCONNECTED	A DISCONNECT, RELEASE COMPLETE,

External Event	Indicates:
	or RELEASE message has been received by the application. This event may be received in any state except IDLE, DISCONNECTED, or NULL. The application must send a **cc_DropCall( )** after this event is received. Use the **cc_GetCallInfo( )** function to retrieve additional information about the event.
CCEV_DIVERTED	NAM with divert information has been received by the application. An outgoing call has been successfully diverted to another station.
CCEV_DROPACK	The network has honored a DROP request for a conference call; the request was made by sending the SndMsg_Drop message type via the **cc_SndMsg( )** function. The event is sent on the corresponding line device. The CCEV_DROPACK event pertains only to a Custom BRI 5ESS switch type.
CCEV_DROPREJ	The network has **not** honored a DROP request for a conference call. The event is sent on the corresponding line device. The CCEV_DROPREJ event pertains only to a Custom BRI 5ESS switch type.
CCEV_ FACILITY	A FACILITY REQUEST message has been received by the application.
CCEV_FACILITYACK	A FACILITY_ACK message has been received by the application.
CCEV_FACILITYGLOBAL	An ISDN_FACILITY message containing a Global CRN value was received. This event is sent on the board level device, as the event

External Event	Indicates:
	is associated with all calls on the device. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_FACILITYNULL	An ISDN_FACILITY message was received containing a Dummy (NULL) CRN. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_FACILITYREJ	A FACILITY_REJ message has been received by the application.
CCEV_HOLDCALL	A HOLD message was received from remote equipment. The application may send **cc_HoldAck( )** or **cc_HoldRej( )** as a response.
CCEV_INFOGLOBAL	An ISDN_INFORMATION message containing a Global CRN value was received. This event is sent on the board level device, as the event is associated with all calls on the device. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_INFONULL	An ISDN_INFORMATION message was received containing a NULL CRN. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_L2FRAME	A data link layer frame has been received by the application. The application should use the **cc_GetFrame( )** function to retrieve the

External Event	Indicates:
	received frame. It is the application's responsibility to analyze the contents of the frame.
CCEV_L2NOBFFR	There are no buffers available to save the incoming frame.
CCEV_NOFACILITYBUF	There are no buffers available to store the Network Facility Information Element (IE). This event can be ignored. The event is received for every incoming ISDN message that contains the Network Facility IE. The IE can be retrieved using the **cc_GetInfoElem( )** function or the **cc_GetCallInfo(U_IES)** function.
CCEV_NOTIFY	A NOTIFY message has been received by the application. Use the **cc_GetCallInfo( )** function to retrieve additional information about the event.
CCEV_NOTIFYGLOBAL	An ISDN_NOTIFY message containing a Global CRN value was received. This event is sent on the board level device, as the event is associated with all calls on the device. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_NOTIFYNULL	An ISDN_NOTIFY message was received containing a Dummy (NULL) CRN. Upon receipt of this event, the application may issue a **cc_GetNonCallMsg( )** function to retrieve the data into its local structure.
CCEV_NSI (DPNSS and Q.SIG only)	A Network Specific Indication (NSI) message was received from the network. The

External Event	Indicates:
	application should use **cc_GetCallInfo( )** to retrieve the NSI string(s).
CCEV_ISDNMSG	An undefined ISDN message has been received by the application.
CCEV_PROCEEDING	An ISDN message has been received by the application. By default, the firmware will send this event to the application. The application may clear the CCMSK_PROCEEDING bit to block this event.
CCEV_PROGRESSING	A PROGRESS message has been received by the application. By default, the firmware will send this event to the application. The application may block this event by clearing the CCMSK_PROGRESS bit. Use the **cc_GetCallInfo( )** function to retrieve additional information about the event.
CCEV_RCVTERMREG_ACK	The acceptance, by the switch, of a terminal initialization request from a station that has been configured as a North American User side (TE) station. This event occurs on a board level device, not a channel device. Use the USPID_BLK data structure to retrieve information about the event.
CCEV_RCVTERMREG_ NACK	The rejection, by the switch, of a terminal initialization request from a station that has been configured as a North American User side (TE) station. This event occurs on a board level device, not a channel device. Use the TERM_NACK_BLK data structure to retrieve information about the event.

External Event	Indicates:
CCEV_REDIRECT	The firmware has reset its call state to Overlap Sending. The request to redirect was made by the switch to solicit more out-of-band digits. The CCEV_REDIRECT event pertains only to a Custom BRI 5ESS switch type configured as the TE (User side). A station configured as an NT (Network side) may generate a message, via the **cc_SndMsg( )** function, causing this event.
CCEV_RETRIEVECALL	A RETRIEVE CALL acknowledge message was received from terminal equipment. The indicated terminal equipment requests reconnection of a call on hold. This event can only be received by an application when the application acts as a network.
CCEV_SERVICE	A SERVICE message has been received from the network.
CCEV_SERVICEACK	A SERVICE ACKNOWLEDGE message has been received from the network.
CCEV_SETCHANSTATE	The B channel status was changed as a result of **cc_SetChanState( )** or a network request. This event occurs on a board level device, not a channel device.
CCEV_SETUP_ACK	A SETUP_ACK message has been received by the application.
CCEV_STATUS	A STATUS message has been received from the network.
CCEV_STATUS_ENQUIRY	A STATUS_ENQ message has been received from the network.

External Event	Indicates:
CCEV_TERM_REGISTER	An unsolicited event indicating that some action is required by the switch for the North American terminal initialization procedure. The application must respond to the event using the **cc_TermRegisterResponse( )** function to either accept or reject the request to initialize the terminal. A Service Profile Identifier (SPID) value is associated with the event. A SPID is assigned by a phone company to a Fully Initializing Terminal (FIT) on an ISDN Basic Rate Interface (BRI) line. This event occurs on a board level device, not on a channel device. Use the SPID_BLK data structure to retrieve information about the event.
CCEV_TIMER	An unsolicited event indicating that a timer has expired.
CCEV_TRANSFERACK	A TRANSFER ACKNOWLEDGE MESSAGE was received from the network. The indicated network has accepted a request to transfer a call.
CCEV_TRANSFERREJ	A TRANSFER REJECT message was received from the network. The indicated network has rejected a request to transfer a call.
CCEV_TRANSIT (DPNSS and Q.SIG only)	After a transfer is established, transit messages are used for relating messages between the originating end and the terminating end.
CCEV_USRINFO	A USER INFORMATION message has been received by the application, indicating that a user-to-user information (UUI) event is coming. For example, this event is received

External Event	Indicates:
	in response to a **cc_SndMsg( )** function call, from the far end, in which the **msg_type** is SndMsg_UsrInformation. Use the **cc_GetCallInfo( )** function to retrieve the UUI.

7.2. Error Handling

The ISDN library functions return a value indicating success (0) or failure (<0) of the function call. **cc_CauseValue( )** is used to retrieve the reason for the failure. The application may use the **cc_ResultMsg( )** function to interpret the returned value.

ISDN error/cause codes consist of two parts: error location and reason. The error location is the upper byte and the reason is the lower byte. For example, the error code (ERR_ISDN_FW | ISDN_CHRST_ERR) indicates that the error is located in the firmware and the reason for the failure is a channel restart error.

There are three cause/error locations, as described in *Table 49* below.

Table 49. Cause/Error Locations

Cause/Error Location	Return Conditions
Firmware (ERR_ISDN_FW)	Returned when there is a firmware-related cause/error. Firmware errors are listed in the *isdncmd.h* file. (See *Section 7.2.1. Cause/Error Codes from the ISDN Firmware*.)
Network (ERR_ISDN_CAUSE)	Returned with a CCEV_DISCONNECTED event. The application uses **cc_ResultValue( )** to retrieve the cause value. Network cause values are listed in the *isdncmd.h* file. (See *Section 7.2.2. Cause/Error Codes from the ISDN Network*.)
ISDN Library (ERR_ISDN_LIB)	Returned when there is a library-related cause/error. Library errors are listed in the *isdnerr.h* file. (See *Section 7.2.3. Cause/Error Codes from the ISDN Library*.)

The following sections provide the cause values and return codes from each of the error/cause locations. Refer to the files *ccerr.h* and *isdncmd.h* for a comprehensive list of error values.

7.2.1. Cause/Error Codes from the ISDN Firmware

The following table provides the error/cause codes located in the ISDN firmware. Error values include the hex followed by the decimal equivalent in parentheses.

Table 50. ISDN Firmware Error Codes

Error Name	Value	Description
ISDN_OK	0x00 (0)	Normal returned code.
ISDN_BADDSL	0x101 (257)	Wrong DSL (Digital Subscriber Line) number. Will not occur in non-NFAS environment.
ISDN_BADTS	0x102 (258)	Bad time slot. Will occur when a second call is placed on an already active channel.
ISDN_BADARGU	0x103 (259)	Bad internal firmware command argument(s), possibly caused by a bad function parameter.
ISDN_BADSTR	0x105 (261)	Bad phone number string. Phone digits string contains invalid phone digit number.
ISDN_BADIF	0x106 (262)	Bad ISDN interface ID. Will not occur in non-NFAS environment.
ISDN_MISSIE	0x107 (263)	Missing mandatory IE.
ISDN_CFGERR	0x108 (264)	Configuration error .
ISDN_CHRST_ERR	0x10A (266)	Channel restart error.
ISDN_BADSERVICE	0x10D (269)	The requested network service, such as **cc_ReqANI( )** or **cc_SndMsg( )**, is not supported by the network and has been rejected.

Error Name	Value	Description
ISDN_BADCALLID	0x10E (270)	Bad call ID. No call record exists for specified call ID.
ISDN_BADSTATE	0x10F (271)	Cannot accept the event in the current state.
ISDN_BADSS	0x110 (272)	Unspecified service state was requested.
ISDN_TSBUSY	0x111 (273)	Time slot already in use.
ISDN_NOAVAIL	0x112 (274)	No more memory available to accept a new call request.
ISDN_LINKFAIL	0x113 (275)	Layer 2 data link failed. This code is returned when the firmware cannot send a message due to L2 data link failure.
ISDN_BADMSG	0x11D (285)	Unsupported messages for DASS2: ALERTING, CONGESTION, FACILITY, FACILITY_ACK, FACILITY_REJECT, UUI, NOTIFY, and RELEASE.
ISDN_INVALID_EVENT	0x11E (286)	Invalid event for the switch.
ISDN_INVALID_SWITCH_ TYPE	0x124 (292)	Switch type not supported.
ISDN_MISSING_FIXED_TEI	0x126 (294)	Fixed Terminal Equipment Identifier (TEI) value not provided for non-initializing terminal.
ISDN_MISSING_DN	0x127 (295)	Directory number not specified for terminal.

Error Name	Value	Description
ISDN_MISSING_SPID	0x128 (296)	Service Profile Interface ID (SPID) not provided for North American terminal.

Cause/Error Codes from the ISDN Firmware for cc_SetBilling( )

The following table provides error codes from the firmware that apply only to the **cc_SetBilling( )** function. The error values include the hex followed by the decimal in parentheses.

Table 51. ISDN Firmware Error Codes for cc_SetBilling( )

Error Name	Value	Description
ISDN_FB_UNAVAIL	0x115 (277)	Flexible billing feature is not available.
ISDN_FB_BAD_OPER	0x117 (279)	Bad operation.
ISDN_FB_BAD_ARG	0x118 (280)	Bad argument.
ISDN_FB_RET_ERR	0x119 (281)	Return error component value.
ISDN_FB_IE_ERR	0x11A (282)	Bad information element.
ISDN_NO_FB_INFO	0x11B (283)	No flexible billing information.

NOTE: The **cc_SetBilling( )** function and the associated firmware return codes apply only to users who have access to AT&T's Vari-A-Bill service.

7.2.2. Cause/Error Codes from the ISDN Network

The following table provides the cause/error codes located in the ISDN Network. Error names are listed in alphabetical order. The values listed in the table include the hex followed by the decimal in parentheses. The Q.931 codes and their descriptions refer to International Telecommunications Union (ITU) Q.931

standards. Not all cause codes are universally supported across switch types. Before using a particular cause code, compare its validity with the appropriate switch vendor specifications.

Table 52. ISDN Network Error Codes

Error Name	Hex Value (Decimal)	Q.931 Code and Description
UNASSIGNED _NUMBER	0x201 (513)	Cause 01 Unassigned (unallocated) number
NO_ROUTE	0x202 (514)	Cause 02 No route to specified transit network
CHANNEL_UNACCEPTABLE	0x206 (518)	Cause 06 Channel unacceptable
NORMAL_CLEARING	0x210 (528)	Cause 16 Normal call clearing
USER_BUSY	0x211 (529)	Cause 17 User busy
NO_USER_RESPONDING	0x212 (530)	Cause 18 No user responding
CALL_REJECTED	0x215 (533)	Cause 21 Call rejected
NUMBER_CHANGED	0x216 (534)	Cause 22 Number changed
DEST_OUT_OF_ORDER	0x21B (539)	Cause 27 Destination out of order

Error Name	**Hex Value (Decimal)**	**Q.931 Code and Description**
INVALID_NUMBER_FORMAT	0x21C (540)	Cause 28 Invalid number format (incomplete number)
FACILITY_REJECTED	0x21D (541)	Cause 29 Facility rejected
RESP_TO_STAT_ENQ	0x21E (542)	Cause 30 Response to STATUS ENQUIRY
UNSPECIFIED_CAUSE	0x21F (543)	Cause 31 Normal, unspecified cause
NO_CIRCUIT_AVAILABLE	0x222 (546)	Cause 34 No circuit/channel available
NETWORK_OUT_OF_ORDER	0x226 (550)	Cause 38 Network out of order
TEMPORARY_FAILURE	0x229 (553)	Cause 41 Temporary failure
NETWORK_CONGESTION	0x22A (554)	Cause 42 Switching equipment congestion
REQ_CHANNEL_NOT_AVAIL	0x22C (556)	Cause 44 Requested channel/circuit not available
PRE_EMPTED	0x22C (557)	Cause 45 Call preempted
FACILITY_NOT_SUBSCRIBED	0x232 (562)	Cause 50 Requested facility not subscribed

Error Name	Hex Value (Decimal)	Q.931 Code and Description
OUTGOING_CALL_BARRED	0x234 (564)	Cause 52 Outbound call barred
INCOMING_CALL_BARRED	0x236 (566)	Cause 54 Incoming call barred
BEAR_CAP_NOT_AVAIL	0x23A (570)	Cause 58 Bearer capability not presently available
SERVICE_NOT_AVAIL	0x23F (575)	Cause 63 Service or option not available, unspecified
CAP_NOT_IMPLEMENTED	0x241 (577)	Cause 65 Bearer capability not implemented
CHAN_NOT_IMPLEMENTED	0x242 (578)	Cause 66 Channel type not implemented
FACILITY_NOT_IMPLEMENT	0x245 (581)	Cause 69 Requested facility not implemented
INVALID_CALL_REF	0x251 (593)	Cause 81 Invalid call reference value
CHAN_DOES_NOT_EXIST	0x252 (594)	Cause 82 Identified channel does not exist
INCOMPATIBLE_DEST	0x258 (600)	Cause 88 Incompatible destination
INVALID_MSG_UNSPEC	0x25F (607)	Cause 95 Invalid message, unspecified
MANDATORY_IE_MISSING	0x260 (608)	Cause 96 Mandatory IE missing

Error Name	Hex Value (Decimal)	Q.931 Code and Description
NONEXISTENT_MSG	0x261 (609)	Cause 97 Message type non-existent or not implemented
WRONG_MESSAGE	0x262 (610)	Cause 98 Message not compatible with call state or message type non-existent or not implemented
BAD_INFO_ELEM	0x263 (611)	Cause 99 Information element non-existent or not implemented
INVALID_ELEM_CONTENTS	0x264 (612)	Cause 100 Invalid information element contents
WRONG_MSG_FOR_STATE	0x265 (613)	Cause 101 Message not compatible with call state
TIMER_EXPIRY	0x266 (614)	Cause 102 Recovery on timer expiry
MANDATORY_IE_LEN_ERR	0x267 (615)	Cause 103 Mandatory IE length error
PROTOCOL_ERROR	0x26F (623)	Cause 111 Protocol error, unspecified
INTERWORKING_UNSPEC	0x27F (639)	Cause 127 Interworking, unspecified

7.2.3. Cause/Error Codes from the ISDN Library

The following table provides the error/cause codes located in the ISDN library. Error values include the hex followed by the decimal in parentheses.

Table 53. ISDN Library Error Codes

Error Name	Value	Description
E_ISSUCC	0x00 (0)	Message acknowledged.
E_ISREADY	0x301 (769)	Board not ready.
E_ISCONFIG	0x302 (770)	Configuration error.
E_ISNOINFO	0x303 (771)	Information not available.
E_ISNOFACILITYBUF	0x305 (773)	Network facility buffer not ready.
E_ISBADBUFADDR	0x306 (774)	Bad buffer address.
E_ISBADTS	0x307 (775)	Bad time slot.
E_ISMAXLEN	0x385 (901)	Exceeds maximum length allowed.
E_ISNULLPTR	0x386 (902)	Null pointer error.
E_ISNOMEM	0x387 (903)	Out of memory.
E_ISFILEOPENFAIL	0x388 (904)	Failed to open a file.
E_ISTNACT	0x389	Trace is not activated. This error

Error Name	Value	Description
	(905)	returns when the application either tries to stop a non-existent trace function or to start the trace function twice on the same D channel.
E_ISBADPAR	0x38A (906)	Bad input parameter(s).
E_ISBADCALLID	0x3C1 (961)	Bad call identifier.
E_ISBADCRN	0x3C2 (962)	Bad call reference number.
E_ISNOINFOBUF	0x3C3 (963)	The information requested in the **cc_GetCallInfo( )** function call is not available.
E_ISINVNETWORK	0x3C4 (964)	Invalid network type. Applies only to the **cc_ReqANI( )** function.
E_FB_UNAVAIL	0x3C8 (968)	Flexible billing unavailable. Applies only to the **cc_SetBilling( )** function.
E_ISBADIF	0x3C9 (969)	Bad interface number.
E_TRACEFAIL	0x3CA (970)	Failed to get trace information.
E_UNKNOWNRESULT	0x3CB (971)	Unknown result code.
E_BADSTATE	0x3CD (973)	Bad state.
E_ABORTED	0x3CE (974)	Previous task was aborted by **cc_Restart( )**.

8. Application Guidelines

This chapter offers advice and suggestions for programmers designing and coding Dialogic ISDN applications in a Windows or a LINUX environment. Specific guidelines for developing ISDN applications are provided.

Topics include the following:

- general guidelines
- handling events, errors and alarms
- programming considerations
- diagnostic tools

NOTE: These guidelines are not intended as a comprehensive guide to developing or debugging Dialogic ISDN applications.

8.1. General Guidelines

This section provides general guidelines for writing applications including explanations of:

- symbolic defines
- header files
- aborting and terminating the application

8.1.1. Symbolic Defines

Applications containing numerical values for defines are subject to obsolescence. In general, Dialogic recommends using symbolic defined names rather than numerical values. Defines are found in the *cclib.h* header file.

8.1.2. Header Files

Various header files must be included in an ISDN application. These files provide the equates, structures, and prototypes needed to compile application programs. The following header files are typically used for ISDN call control applications:

- *cclib.h* - ISDN Call Control library defines
- *isdncmd.h* - ISDN command structure defines
- *isdnlib.h* - ISDN library headers
- *isdnerr.h* - ISDN error defines
- *srllib.h* - SRL headers for event handling
- *dtilib.h* - DTI headers for layer 1 board device handling

8.1.3. Aborting and Terminating the Application

Upon aborting a Dialogic ISDN API application, the operating system terminates the current process, but may leave devices in an unknown state. This may result in errors the next time the application is run. To avoid errors of this type, the application should trap the following terminating signals to terminate the application:

1. Disconnect all active calls (in CONNECTED state).
2. Abort all calls in progress, either by dropping and releasing the call or by issuing **cc_Restart( )** on each line device.
3. Set the line state to "Out of Service" or "Maintenance" if that option is available in the protocol being used.

When the process completes, any device claimed with **cc_Open( )** must be released using the **cc_Close( )** function.

8.2. Handling Errors, Events and Alarms

The Dialogic ISDN API library is an extension of the DTI library. Therefore, events, errors, and alarms are handled by the application in the same way as Dialogic DTI applications. Refer to the *Digital Network Interface Software*

Reference and the *Voice Software Reference - Standard Runtime Library* for the appropriate operating system for more information.

8.2.1. Handling Errors

Each Dialogic ISDN API library function returns a value < 0 on failure. Be sure to check any call to a Dialogic ISDN API library function for a return value that indicates an error. The following code samples demonstrate how to handle errors done in asynchronous and synchronous modes.

Asynchronous Mode

```
sr_waitevt(-1);
code = sr_getevttype( );
if (code&DT_CC) == DT_CC) {        /* comments */
     switch(code) {
          case CCEV_DISCONNECTED:
               cc_ResultValue( );
               break;
     }
}
```

Synchronous Mode

```
char *msg;
if (cc_XXX(ldev) == CC_ERROR) {
     error = cc_CauseValue(ldev);
     error = cc_ResultMsg(ldev, &msg);
     printf("error msg = %s\n", msg);
}
```

8.2.2. Handling Events

When an event occurs, the application may use **sr_getevttype( )** to retrieve the event type. For example:

```
switch (sr_getevttype()) {
case CCEV_OFFERED:
    .
    .
    .
case CCEV_CONNECTED:
    .
    .
```

The application may also use the Dialogic function calls to retrieve additional event or error information. For example:

- **ATDV_NAMEP( )**
- **ATDV_LASTERR( )**
- **ATDV_ERRMSGP( )**
- **sr_getevtdatap( )**
- **sr_getevtlen( )**

8.2.3. Handling Alarms

All ISDN trunk alarms are reported to and handled by the application in the same way as DTI trunk alarms. Additionally, when the trunk is lost, all active calls will be disconnected automatically and a CCEV_DISCONNECTED event will be reported for each line device.

8.3. Programming Considerations - PRI and BRI

This section addresses the following programming considerations involved in using board functions:

- resource association (PRI and BRI)
- MAKECALL Block initialization and settings
- information element settings

8.3.1. Resource Association

Each Dialogic ISDN API line device is implemented as an association of a network time slot device. The application uses **cc_Open( )** to indicate to the driver which resources to use to control a specific line.

Each PRI structure is composed of one D channel and 23 (T1) or 30 (E1) B (bearer) channels. A PRI board device, such as dtiB1, is defined as a station and controls the D channel. A PRI time slot device, such dtiB1T1, is defined as a bearer channel under a station.

Each BRI structure is composed of one D channel and two B (bearer) channels. A BRI board device, such as briS1, is defined as a station and controls the D channel the same way as a PRI board device. A BRI time slot device, such as briS1T1, is defined as a bearer channel under a station and is handled the same way as a PRI line device.

NOTE: For BRI, the protocol must be configured at initialization using the **cc_SetDChanCfg( )** function. See *Section 8.4.1. BRI/SC Configuration* below and the **cc_SetDChanCfg( )** function description in *Chapter 5. ISDN Function Reference* for more information.

8.3.2. MAKECALL Block Initialization and Settings

Because ISDN services vary with switches and provisioning plans, a set of default standards cannot be set for the MAKECALL_BLK. Therefore, it is up to the user application to fill in the applicable MAKECALL_BLK values that apply to the particular provisioning.

The current ISDN Call Control library will check for the ISDN_NOTUSED (0xFF) define to determine

- which Information Elements (IEs) to send down to the firmware
- the length of each IE to be sent

All of the bearer capability elements in the MAKECALL_BLK structure must be specified or the ISDN library will not properly fill in the ISDN_CMD messages to pass down to the firmware. It is suggested that the application first initialize the MAKECALL_BLK structure with a set of defaults prior to filling in settings pertaining to the particular ISDN service.

A sample MAKECALL_BLK initialization is shown below:

```
/* Initialize the MAKECALL block */
           makecall_blk.isdn.BC_xfer_cap = BEAR_CAP_SPEECH;
           makecall_blk.isdn.BC_xfer_mode = ISDN_ITM_CIRCUIT;
           makecall_blk.isdn.BC_xfer_rate = BEAR_RATE_64KBPS;
           makecall_blk.isdn.usrinfo_layer1_protocol = 0xFF;
           makecall_blk.isdn.usr_rate = 0xFF;
           makecall_blk.isdn.destination_number_type = 0xFF;
           makecall_blk.isdn.destination_number_plan =0xFF;
           makecall_blk.isdn.destination_sub_number_type = 0xFF;
           makecall_blk.isdn.destination_sub_phone_number[0] = NULL;
           makecall_blk.isdn.origination_number_type = 0xFF;
           makecall_blk.isdn.origination_number_plan = 0xFF;
```

```
makecall_blk.isdn.origination_sub_number_type = 0xFF;
makecall_blk.isdn.origination_sub_phone_number[0] = NULL;
makecall_blk.isdn.origination_phone_number[0] = NULL;
makecall_blk.isdn.facility_feature_service = 0xFF;
makecall_blk.isdn.facility_coding_value = 0xFF;
makecall_blk.isdn.usrinfo_bufp = NULL;
makecall_blk.isdn.nsfc_bufp = NULL;
```

For more on the MAKECALL_BLK structure, see *Section 6.8. MAKECALL_BLK* and the **cc_MakeCall( )** function description in *Chapter5. ISDN Function Reference*.

8.3.3. Information Element Settings

The information elements (IEs) to be passed down to the network need to conform to the switch-specific recommendations. Use the assumptions listed below when setting IEs.

Assumption 1: The customer is responsible for providing variable length IEs in ascending order in the Public part, as shown in the following table.

Table 54. Variable Length IEs

Type of IE	Value
Network Specific Facilities	0x20
Display	0x28
Signal	0x34

Assumption 2: A single byte IE (with the exception of a LOCKING Shift IE) can be placed anywhere in the message. This includes Type 1 (NON-LOCKING Shift) and Type 2 elements. The NON-LOCKING shift should cause the codeshift in the forward direction only. For example, when in codeset "3," the NON-LOCKING shift should add an element in codeset "4." The following tables show the settings for Type 1 and Type 2 IEs.

Table 55. NON-LOCKING Shift IEs - Type 1

Type of IE	Value	Codeset
Network Specific Facilities	0x20	0
Shift	0x9E	6 (NON-LOCKING)
IPU	0x76	6
Display	0x28	0
Signal	0x34	0

Table 56. Single Byte IEs - Type 2

Type of IE	Value	Codeset
Network Specific Facilities	0x20	0
Sending Complete	0xA1	0 (Single Byte IE)
Display	0x28	0
Signal	0x34	0

Assumption 3: A LOCKING Shift IE must be placed after all the IEs when a lower codeset is included. A NON-LOCKING Shift IE or another LOCKING Shift IE of a greater codeset value can follow the IE. The following tables provide two options for setting LOCKING Shift IEs.

Table 57. LOCKING Shift IEs - Option 1

Type of IE	Value	Codeset
Network Specific Facilities	0x20	0
Sending Complete	0xA1	0 (Single Byte IE)
Display	0x28	0
Signal	0x34	0

Type of IE	Value	Codeset
Shift	0x94	4 (LOCKING)
IPU	0x76	4
Shift	0x9E	6 (NON-LOCKING)
DDD	0x55	6
SSS	0x44	4
Shift	0x97	7 (LOCKING)
ABC	0x77	7
DEF	0x77	7

Table 58. LOCKING Shift IEs - Option 2

Type of IE	Value	Codeset
Network Specific Facilities	0x20	0
Sending Complete	0xA1	0 (Single Byte IE)
Display	0x28	0
Keypad Facility	0x2C	0
Shift	0x96	6 (LOCKING)
IPU	0x76	6
Shift	0x90	0 (NON-LOCKING)
Signal	0x34	0
ABC	0x77	6
DEF	0x77	6
Shift	0x97	7 (LOCKING)
ABC	0x77	7

Type of IE	Value	Codeset
DEF	0x77	7

Assumption 4: User-supplied IEs (with the exception of CHANNEL_ID_IE, see below) take precedence over the Firmware-defined IEs, even those that are in private IE parts.

Assumption 5: The CHANNEL_ID_IE will always be taken from the Firmware-defined section.

Assumption 6: When Single Byte IEs and NON-LOCKING Shift IEs occur in both the User-supplied and Firmware-defined sections, the value will be taken from the User-defined section, but it will be inserted at the position defined by the firmware, assuming that there may be specific requirements on the firmware to have the position.

For more information on setting IEs, see the **cc_SetInfoElem( )** function description.

8.4. Programming Considerations - BRI/SC Only

This section provides the following information, which pertains only to BRI/SC protocols:

- BRI/SC Configuration
- BRI/SC Terminal Initialization
- BRI/SC Tone Generation Configuration

8.4.1. BRI/SC Configuration

Unlike the PRI firmware, the BRI firmware requires the application to first configure the desired protocol and features via the **cc_SetDChanCfg( )** function. This is necessary for BRI/SC protocols as there is only one BRI firmware file containing multiple protocols and the firmware needs to know which protocol is to be configured. The protocol also needs to know whether the station is to be configured as a Network side or User side station.

NOTE: North American protocols often require TE devices configured as the User side to transmit a Service Profile Identifier (SPID), which is then acknowledged by the switch. The SPID is programmed using the **cc_SetDChanCfg( )** function. See *Section 8.4.2. BRI/SC Terminal Initialization* below for more information.

After the firmware is downloaded to the board, each station that requires D channel signaling must be configured individually using the **cc_SetDChanCfg( )** function. For example, to configure 16 stations, 16 **cc_SetDChanCfg( )** operations must be performed, with each operation specifying the appropriate station ID (for example, "briS1" to "briS16").

The D channel of each line can be configured at any time and as many times as needed. This includes changing between Network and User sides as well as changing protocols, SPIDs, and other attributes, such as tone generation (see *Section 8.4.3. BRI/SC Tone Generation Configuration* below). If calls are active at the time of reconfiguration, the application will receive CCEV_DISCONNECTED events for any calls that existed when the **cc_SetDChanCfg( )** function was issued. Those calls are disconnected by the application through normal call tear-down procedures. All data links are then severed and reconnected (some configurations do not require reconnection).

NOTE: The SPID, directory number, and subaddress of a User-side line can be changed at any time without reconfiguring the channel. See the **cc_SetParmEx( )** function description in *Chapter 5. ISDN Function Reference* for more information.

It is recommended that the **cc_SetDChanCfg( )** operations be performed by a separate system configuration task and not as part of a call processing task or thread.

8.4.2. BRI/SC Terminal Initialization

The BRI North American protocols may require terminal initialization settings prior to establishing any layer 3 connectivity. Dialogic API support for terminal initialization consists of the **cc_SetDChanCfg( )**, **cc_SetParmEx( )** and **cc_TermRegisterResponse( )** functions, and the following events:

- CCEV_TERM_REGISTER

- CCEV_RCVTERMREG_ACK
- CCEV_RCVTERMREG_NACK

North American protocols often require TE devices to be fully initializing. This means that the Service Profile Identifier (SPID) must be transmitted and acknowledged by the switch. For the User side, the SPID is programmed in the D channel configuration using **cc_SetDChanCfg( )**. When the SPID is accepted or rejected by the switch, the application receives either a CCEV_RCVTERMREG_ACK or a CCEV_RCVTERMREG_NACK, respectively.

NOTE: When attaching to a real switch, the SPID must be requested from the switch operator before a connection is attempted.

For the Network side, the application is notified of a TE registration request via the CCEV_TERM_REGISTER board level event. The application must respond to the event using the **cc_TermRegisterResponse( )** function on the board level device to fully initialize the terminal.

Refer to the function descriptions for **cc_TermRegisterResponse( )** and **cc_SetDChanCfg( )** for more information on terminal initialization. Also see the Call Scenarios in *Appendix A* for the sequence of messages and events required for BRI terminal initialization.

8.4.3. BRI/SC Tone Generation Configuration

Tones can be generated and played on any B channel with the use of the BRI/SC board's on-board DSP chip. In-band tones can be activated either by the host application and/or the firmware. This allows the application to combine in-band call progress-related information with the out-of-band signaling information.

The firmware applies tones only to stations configured as the Network side. The firmware must be configured by the host application to apply the call progress tones as well as to which PCM to use.

Under a Network configuration, and when enabled, the firmware automatically generates a dial tone when a SETUP message is received from a user and more digits are required. A ringback tone is generated when the network sends an ALERT message. After a call is connected, no other tones are generated.

The **cc_SetDChanCfg( )** function is used to configure the firmware tone control. In addition, the application can change the values in the firmware tone template table (see *Table 29. Tone Template Table*) using the **cc_ToneRedefine( )** function. To apply user-defined tones (that is, tones other than those in the firmware tone template table), the application uses the **cc_PlayTone( )** and **cc_StopTone( )** functions. For information on these functions, see the function descriptions in *Chapter 5. ISDN Function Reference*.

8.5. Diagnostic Tools (The DialView Suite)

DialView is a suite of tools to help developers test and debug their ISDN applications. DialView includes:

- ISDN Network Firmware (NT1 and NE1)
- ISDN Diagnostic Program (*isdiag*)
- ISDN Trace Utility (*isdtrace*)

8.5.1. ISDIAG Utility

The ISDN Diagnostic program (*isdiag*) is an interactive tool used to help verify ISDN line operation and to assist in troubleshooting the network trunk. When the application is ready for final installation, running this diagnostic program can help in determining what the network carrier is expecting first.

With the ISDN Diagnostic program running, a trace on the inbound call will detect what the network sent. A trace on a failed outgoing call will show the cause of the failure.

When the ISDN Diagnostic Program is first started, users identify the specific board, channel number (time slot), bus type (SCbus), and board type (T-1 or E-1) on which outgoing calls will be made. Incoming calls may be received on any time slot. For a LINUX application, use the F1 key to bring up the help screens and for a description of the menu items.

NOTE: ISDIAG is not intended as an application tester when installed in a system using NT emulation software (*isnt1.fwl* or *dtint1.fwl*, T-1 only).

To start the *isdiag* program, type:

isdiag <board> <channel> <boardtype> <type> <trace mode> <voice>

Where:

<board> is the board number 1 through 4 (dti1 - dti4)

<channel> is the channel time slot number (1-23 for T-1, 1-30 for E-1, 1-2 for BRI)

<boardtype> is the type of board: “t” for T-1, “e” for E-1, “b” for BRI/SC, “b2” for BRI/2)

<type> is the Bus type; “p” for PEB and “s” for SCbus)

<trace mode> indicates whether waitcall is issued at startup; “r” is for trace mode (no waitcall issued at startup). The default is normal mode (waitcall issued at startup).

<voice> indicates whether voice is supported; “v” is for voice supported (span cards). The default is no voice support (dti cards).

If BRI is chosen as the board type, ISDIAG will prompt for input settings for the BRI D channel configuration. Refer to the function description for **cc_SetDChanCfg( )** for information on the possible settings for BRI D channel configuration.

After the **channel** and **type** are selected, the program will automatically configure the system and display the first level menu. At this level, the following actions can be selected:

- Set outbound call parameters
- Request calling party number (ANI)
- Send maintenance request
- Display information (called party subaddress, user-to-user information, B and D channel status)
- Drop ISDN calls
- Make outbound ISDN calls
- Stop play/record/dial
- Set and get ISDN information elements
- Send ISDN messages
- Start, stop, and browse ISDN trace files

- Restart ISDN line devices and set to WaitCall state for receiving inbound ISDN calls
- Change the current ISDN line device number
- Use a shell to access DOS environment form the ISDIAG application
- Hold/retrieve calls (BRI protocols and DPNSS and Q.SIG PRI protocols)
- Set supplementary DPNSS/Q.SIG services (intrusion, local diversion, remote diversion, virtual calls for inbound/outbound)
- Use an online Help menu that describes the main menu options

8.5.2. ISDTRACE Utility

The ISDN trace utility program (*isdtrace.exe*) translates the recorded binary ISDN trace file (*filename.log*) into a formatted text file (*filename.res*) for easy reading. The binary trace file is generated using the **cc_StartTrace( )** and **cc_StopTrace( )** functions.

NOTE: The *isdtrace* utility is identical to *pritrace*, which was used for ISDN primary rate products only. The *isdtrace* utility is used for both primary rate and basic rate products. The file is located in the C:\dialogic\bin\ directory. Using *pritrace* with BRI/2 or BRI/SC boards will result in a format error.

To start the *isdtrace* program, type:

isdtrace <infilename> [<outfilename>] [-p | -b]

Where:

<infilename> is the saved binary file from the trace functions

<outfilename> is the ASCII readable trace of D channel

[-p | -b] indicates Primary Rate Interface (PRI) or Basic Rate Interface (BRI)

NOTE: The *isdtrace* program creates a temporary file called *isdtemp.log*. The *isdtemp.log* file contains the hex information of the binary input file.

The following table provides an example of a file fragment that shows the translated data:

Table 59. ISDTRACE Example File

Receive	Transmit

```
NET5

RECEIVE
Response=0   SAPI=0x00
TEI=0x00
0x01 0x09  Receive Ready
```

```
TRANSMIT
Command=0    SAPI=0x00
TEI=0x00
0x01 0x0b  Receive Ready

TRANSMIT
Response=1   SAPI=0x00
TEI=0x00
0x08 0x0a  Information
Dest=0   CR=0x0002
SETUP(0x05)
  1:           SENDING COMPLETE(0xa1)
  1:           BEARER CAPABILITY(0x04)
  2:           IE Length(0x02)
  3:  1------- Extension Bit
      -00----- Coding Standard
      ---00000 Info. Transfer Cap.
  4:  1------- Extension Bit
      -00----- Transfer Mode
      ---10000 Info. Transfer Rate
  1:           CHANNEL ID(0x18)
  2:           IE Length(0x03)
  3:  1------- Extension Bit
      -0------ Interface ID Present
      --1----- Interface Type
      ---0---- Spare
      ----1--- Preferred/Exclusive
      -----0-- D-Channel Indicator
      ------01 Info. Channel Sel.
3.2:  1------- Extension Bit
      -00----- Coding Standard
      ---0---- Number Map
      ----0011 Channel/Map Element
  4:  1------- Extension Bit
      -0000010 Channel Number/Slot Map
  1:           CALLED PARTY NUM(0x70)
  2:           IE Length(0x0b)
  3:  1------- Extension Bit
      -010---- Type of Number
      ----0001 Numbering plan ID
      2019933000  Number Digit(s)
  1:           CALLED PARTY SUBADD(0x71)
  2:           IE Length(0x04)
  3:  1------- Extension Bit
      -000---- Type of Subaddress
          0x01 Subaddress Info.
          0x02 Subaddress Info.
          0x03 Subaddress Info.
  1:           USER-USER(0x7e)
  2:           IE Length(0x4)
  3:      0x04 Protocol Discrim.
```

Receive	Transmit
	0x44 User Information 0x69 User Information 0x61 User Information
RECEIVE Command=1 SAPI=0x00 TEI=0x00 0x01 0x0a Receive Ready	
RECEIVE Response=0 SAPI=0x00 TEI=0x00 0x0a 0x0a Information Orig=1 CR=0x8002 CALL PROCEEDING(0x02) 1: CHANNEL ID(0x18) 2: IE Length(0x03) 3: 1------- Extension Bit -0------ Interface ID Present --1----- Interface Type ---0---- Spare ----1--- Preferred/Exclusive -----0-- D-Channel Indicator ------01 Info. Channel Sel. 3.2: 1------- Extension Bit -00----- Coding Standard ---0---- Number Map ----0011 Channel/Map Element 4: 0------- Extension Bit -0000010 Channel Number/Slot Map	
	TRANSMIT Command=0 SAPI=0x00 TEI=0x00 0x01 0x0c Receive Ready
RECEIVE Response=0 SAPI=0x00 TEI=0x00 0x0c 0x0a Information Orig=1 CR=0x8002 CALL CONNECT(0x07)	
	TRANSMIT Command=0 SAPI=0x00 TEI=0x00 0x01 0x0e Receive Ready
	TRANSMIT Response=1 SAPI=0x00 TEI=0x00 0x0a 0x0e Information Dest=0 CR=0x0002 CALL CONNECT ACKNOWLEDGE(0x0f)
RECEIVE Command=1 SAPI=0x00 TEI=0x00 0x01 0x0c Receive Ready	
	TRANSMIT Response=1 SAPI=0x00 TEI=0x00 0x0c 0x0e Information Dest=0 CR=0x0002 CALL DISCONNECT(0x45) 1: CAUSE(0x08)

Receive	Transmit
	2: IE Length(0x02) 3: 1------- Extension Bit -00----- Coding Standard ---0---- Spare ----0010 Location 4: 1------- Extension Bit -0010000 Cause Value
RECEIVE Command=1 SAPI=0x00 TEI=0x00 0x01 0x0e Receive Ready	
RECEIVE Response=0 SAPI=0x00 TEI=0x00 0x0e 0x0e Information Orig=1 CR=0x8002 RELEASE(0x4d)	
	TRANSMIT Command=0 SAPI=0x00 TEI=0x00 0x01 0x10 Receive Ready
	TRANSMIT Response=1 SAPI=0x00 TEI=0x00 0x0e 0x10 Information Dest=0 CR=0x0002 RELEASE COMPLETE(0x5a)
RECEIVE Command=1 SAPI=0x00 TEI=0x00 0x01 0x10 Receive Ready	

Appendix A - Call Control Scenarios

This appendix contains the following ISDN call control scenarios in the order listed:

- BRI Channel Initialization and Start Up (User Side)
- BRI Channel Initialization and Start Up (Network Side)
- PRI Channel Initialization and Start Up
- Normal Call Establishment and Termination
    - Network initiated call
    - Network terminated call
    - Application initiated call
    - Aborting **cc_MakeCall( )**
    - Application terminated call
- Call Rejection
    - Outgoing call rejected by the network
    - Incoming call rejected by the application
    - Glare (call collision)
    - Simultaneous disconnect (any state)
- Initiation of Hold and Retrieve (BRI, PRI DPNSS and PRI Q.SIG protocols only)
    - Local initiated
    - Remote initiated
- Network Facility Request or Service
    - Vari-A-Bill (AT&T service only)
    - ANI-on-demand, incoming call (AT&T service only)
    - Advice of charge, inbound and outbound call (AT&T service only)
    - Two B Channel Transfer (TBCT)
    - Non-call Associated Signaling (NCAS)

NOTE: Dialogic BRI/SC boards can be used as either Network side or User side (Terminal side). Except where noted, the call control scenarios in this appendix provide programming descriptions for the User side only.

BRI Channel Initialization and Start Up (User Side)

Synchronous or Asynchronous Programming

Application	Device Driver	State	Firmware	Network
cc_Open( ) -->		NULL		
	Return with line device <--			
cc_SetDChanCfg( ) -->	Initialize -->		Configures protocol and BRI station D channel settings	
	CCEV_D_CHAN_ STATUS <--			Establish Data Link
(if Terminal = North American)			SPID information -->	
(if Terminal = North American)	CCEV_RCVTERMREG_ACK (if positive) CCEV_RCVTERMREG_NACK (if negative) <--	SME_TERM_ REGISTER <--		Positive or Negative Acknowledge-ment of SPID information
***cc_WaitCall( )** -->	ISDN_Unblock_Ts -->		Incoming call unblocked	
* Required for both synchronous and asynchronous programming model. This process is done only once per download.				

BRI Channel Initialization and Start Up (Network Side)

Synchronous or Asynchronous Programming

Application	Device Driver	State	Firmware	User to-Network
cc_Open( ) -->		NULL		
	Return with line device <--			
cc_SetDChanCfg( ) -->	Initialize -->		Configures protocol and BRI station D channel settings	
	DATA_LINK_UP			Establish Data Link
(if Terminal = North American)	CCEV_TERM_REGISTER <--	SME_TERM_REGISTER <--		SPID informat-ion
cc_TermRegisterResponse( ) (if Terminal = North American) -->	CCEV_RCVTERMREG_ACK (if positive ack) / CCEV_RCVTERMREG_NACK (if negative ack) -->		Positive or negative acknowledge-ment of SPID information -->	
***** Required for both synchronous and asynchronous programming model.				

PRI Channel Initialization and Start Up

Synchronous or Asynchronous Programming

Application	Device Driver	State	Firmware	Network
			OOS at power up F/W place B channel to "IN" service state	*Maintenance -->
				MT_ACK <--
			F/W resets all B channel in to idle state	**Restart -->
			Call blocked	Restart_ACK <--
cc_Open( ) -->		NULL		
	Return with line device <--			
***cc_WaitCall( )** -->	ISDN_Unblock_Ts -->		Incoming call unblocked	

* Optional for TE/Windows implementation.
** An implementation option for custom equipment, mandatory for network emulation side.
*** Required for both synchronous and asynchronous programming model.

Normal Call Establishment and Termination

This section provides scenarios of normal basic call control procedures for call establishment and termination. Both Facility Associated Signaling (FAS) and Non-Facility Associated Signaling (NFAS) cases are illustrated.

Network initiated call (inbound call)

Synchronous Programming: The incoming call terminates the **cc_WaitCall( )** function. **cc_WaitCall( )** must be issued for the next incoming call after the last call is terminated.

Application	Device Driver	State	Firmware	Network
cc_WaitCall( ) -->	ISDN_Unblock_Ts -->	NULL	Incoming call unblocked	
				Set_Up <--
			*B-channel cut-thru CALL_PROCEEDING -->	Proceeding -->
	CRN assigned termination of cc_WaitCall() <--	OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) -->				
	Return immediately with DNIS <--			
cc_GetANI( ) **or** **cc_ReqANI( )** (option) -->				
	Return immediately with ANI <--			
cc_CallProgress( ) (option) -->	Call_Progress -->		P1=8 (in-band information is now available) CALL_PROGRESS -->	Progress -->
	Return immediately <--		No response from F/W to driver	

Application	Device Driver	State	Firmware	Network
cc_AcceptCall( ) (option) -->	Call_Alert -->		CALL_ALERT -->	Alerting -->
	termination of cc_AcceptCall() <--	ACCEPTED	CALL_ALERT_ACK <--	
cc_AnswerCall( ) -->	Call_Connect -->		*B-channel cut-thru CALL_CONNECT -->	Connect-->
	termination of cc_AnswerCall() <--	CONNECTED	CALL_CONNECT_ACK <--	Conn_ACK <--
***** Application may connect a voice resource channel to the B channel.				

Asynchronous Programming: Incoming call notification is received as an event. **cc_WaitCall( )** needs to be issued only once when the system is initialized.

Application	Device Driver	State	Firmware	Network
cc_WaitCall( ) -->	ISDN_Unblock_Ts -->	NULL	Incoming call unblocked	
				<-- Set_Up
			CALL_PROCEEDING -->	Proceeding -->
	CRN assigned CCEV_OFFERED <--	OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) (option) -->			*B-channel cut-thru	
	Return immediately with DNIS <--			
cc_GetANI( ) or cc_ReqANI( ) (option) -->				
	Return immediately with ANI <--			
cc_CallProgress( ) (option) -->	Call_Progress -->		CALL_PROGRESS P1=8 (in-band information is now available) -->	Progress -->
			No response from F/W to driver	
cc_AcceptCall( ) (option) -->	Call_Alert -->		CALL_ALERT -->	Alerting -->
	CCEV_ACCEPTCALL -->	ACCEPTED	CALL_ALERT_ACK -->	
cc_AnswerCall( ) -->	Call_Connect -->		*B-channel cut-thru CALL_CONNECT -->	Connect -->

	CCEV_ANSWERCALL <--	CONNECTED	CALL_CONNECT_ACK <--	Conn_ACK <--
* Application may connect a voice channel to the B channel.				

Network terminated call

Firmware-controlled disconnect: This is the default setting for parameter 24 in the **.prm* file

Synchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
			*B channel disconnected CALL_RELEASE -->	Disconnect <--
	CCEV_DISCONNECTED <--	DISCONNECTED	CALL_DISC <--	
				**Release -->
				Rel_Comp <--
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->	IDLE		
	Termination of cc_DropCall() <--		CALL_CLEARED <--	
	ISDN_Block_Ts -->		Incoming call unblocked	
cc_ReleaseCall( ) -->	Call_Dealloc -->	NULL		
	Driver releases CRN Return		F/W releases CRN CALL_DEALLOC_ACK <--	
*In the firmware-controlled disconnect process, the firmware ensures that the RELEASE is sent out to the network immediately after the DISCONNECT is received. ****cc_WaitCall( )** must be issued again to receive the next incoming call.				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
			*B channel disconnected CALL_RELEASE -->	Disconnect <--

	CCEV_DISCONNECTED <--	DISCONNECTED	CALL_DISCONNECTED <--	
				Release -->
				Rel_Comp <--
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->	IDLE		
	CCEV_DROPCALL <--		CALL_CLEARED <--	
cc_ReleaseCall( ) -->	Call_Dealloc -->	NULL		
	Driver releases CRN Return		F/W releases CRN CALL_DEALLOC_ACK <--	
*In the firmware-controlled disconnect process, the firmware ensures that the RELEASE is sent out to the network immediately after the DISCONNECT is received.				

Host-controlled disconnect: Parameter 24 in the **.prm* file = 01

Synchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
	CCEV_DISCONNECTED <--	DISCONNECTED	CALL_DISC <--	Disconnect <--
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->	IDLE		
	Termination of cc_DropCall() <--		CALL_CLEARED <--	
	ISDN_Block_Ts -->		Incoming call unblocked	
cc_ReleaseCall ( ) -->	Call_Dealloc -->	NULL	CALL_RELEASE -->	*Release -->
	Driver releases CRN Return		F/W releases CRN CALL_DEALLOC_ACK <--	
				Rel_Comp <--
* In the host-controlled disconnect process, the RELEASE message gets sent out only when the Host does a **cc_ReleaseCall( ).** The **cc_WaitCall( )** must be issued again to receive the next incoming call.				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
	CCEV_ DISCONNECTED <--	DISCONNECTED	CALL_DISC <--	Disconnect <--
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->	IDLE		
	CCEV_DROPCALL <--		CALL_CLEARED <--	
cc_ ReleaseCallEx( ) -->	Call_Dealloc -->	NULL	CALL_RELEASE -->	*Release -->
	Driver releases CRN Return		F/W releases CRN CALL_DEALLOC_ACK <--	

				Rel_Comp <--
* In the host-controlled disconnect process, the RELEASE message gets sent out only when the Host does a **cc_ReleaseCall( ).**				

Application initiated call (outbound call)

Synchronous Programming

Application	Device Driver	State	Firmware	Network
		NULL		
cc_MakeCall( ) -->	CRN assigned Call_Outgoing -->			
			CALL_OUTGOING -->	Set Up -->
	CCEV_PROCEEDING (if requested not masked) <--	DIALING	*B channel cut thru CALL_PROCEEDING <--	Proceeding <--
	CCEV_PROGRESSING (if requested not masked) <--		PI=1 (interworking with a non-ISDN has occurred within the network) CALL_PROGRESS <--	Progress <--
	CCEV_PROGRESSING (if requested not masked) <--		PI=2 (the destination user is not ISDN) CALL_PROGRESS <--	Progress <--
Application may assign a voice resource to detect the in-band tone	CCEV_PROGRESSING (if requested not masked) <--		PI=8 (in-band information is now available) CALL_PROGRESS <--	Progress <--
	CCEV_ALERTING (if requested not masked) <--	ALERTING	CALL_ALERT <--	Alerting option <--
	CCEV_CONNECTED <--	CONNECTED	CALL_CONNECT <--	Connect <--
* Application may connect a voice channel to the B channel.				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		NULL		
cc_MakeCall( ) -->	CRN assigned Call_Outgoing -->			
			CALL_OUTGOING -->	Set Up -->

Application	Device Driver	State	Firmware	Network
	CCEV_PROCEEDING (if requested not masked) <--	DIALING	*B channel cut thru CALL_PROCEEDING <--	Proceeding <--
	CCEV_PROGRESSING (if requested not masked) <--		PI=1 (interworking with a non-ISDN has occurred within the network) CALL_PROGRESS <--	Progress <--
	CCEV_PROGRESSING (if requested not masked) <--		PI=2 (the destination user is not ISDN) CALL_PROGRESS <--	Progress <--
Application may assign a voice resource to detect the in-band tone	CCEV_PROGRESSING (if requested not masked) <--		PI=8 (in-band information is now available) CALL_PROGRESS <--	Progress <--
	CCEV_ALERTING (if requested not masked) <--	ALERTING	CALL_ALERT <--	Alerting option <--
	CCEV_CONNECTED <--	CONNECTED	CALL_CONNECT <--	Connect <--
* Application may connect a voice channel to the B channel.				

Aborting cc_MakeCall( )

When a B channel negotiation is used in call setup, the application must select CCEV_PROCEEDING as the termination point for the **cc_MakeCall( )** function or use the asynchronous programming model. The following scenario illustrates a case where the application uses an asynchronous model to abort the **cc_MakeCall( )** attempt.

Asynchronous Programming

Application	Device driver	State	Firmware	Network
		NULL		
cc_MakeCall( ) -->	CRN assigned Call_Outgoing -->			
			CALL_OUTGOING (request for channel "x") -->	Set Up -->
	CCEV_PROCEEDING (indicating the network requested channel) <--	DIALING	CALL_PROCEEDING (network wishes to use a B channel other than "x") <--	Proceeding <--
cc_DropCall( ) -->	Call_Disconnected (cause value = 1) -->	IDLE		
			B channel disconnected CALL_DISC -->	disconnect -->
Continue the "disconnect" process described below in the "Application Terminated Call" table	-----	---	----	---

Application Terminated Call

Firmware-controlled disconnect: This is the default setting for parameter 24 in the *.prm file

Synchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->			
		IDLE	B channel disconnected CALL_DISC -->	Disconnect -->
				Release <--
	termination of cc_DropCall() <--		CALL_CLEARED <-- **RELEASE_DONE -->	**Rel_Comp -->
	ISDN_Block_Ts -->		Incoming call unblocked	
***cc_ReleaseCall ( )** -->	Call_Dealloc -->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	
***cc_WaitCall( )** must be issued again to receive the next incoming call. **Note that for the firmware-controlled disconnect process, the firmware immediately sends a RELEASE_DONE (and therefore a RELEASE COMPLETE) to the network after receiving a RELEASE message.				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->			
		IDLE	B channel disconnected CALL_DISC -->	Disconnect -->

				Release <--
	CCEV_DROPCALL <--		CALL_CLEARED <-- RELEASE_DONE<br-->	 Rel_Comp<br-->
cc_ReleaseCall ( )<br-->	Call_Dealloc<br-->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	

Host-controlled disconnect: This is the default setting for parameter 24 in the **.prm* file

Synchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->			
		IDLE	B channel disconnected CALL_DISC -->	Disconnect -->
				Release <--
	termination of cc_DropCall() <--		CALL_CLEARED <--	
	ISDN_Block_Ts -->		Incoming call unblocked	
***cc_ReleaseCall ( )** -->	Call_Dealloc -->		**RELEASE_DONE -->	**Rel_Comp -->
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	
***cc_WaitCall( )** must be issued again to receive the next incoming call. **Note that for the host-controlled disconnect process, the firmware sends a RELEASE_DONE (and therefore a RELEASE COMPLETE) to the network only when the host does a **cc_ReleaseCall( )** .				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->			
		IDLE	B channel disconnected CALL_DISC -->	Disconnect -->
				Release <--
	CCEV_DROPCALL <--		CALL_CLEARED <--	

cc_ReleaseCall ( ) -->	Call_Dealloc -->		RELEASE_DONE -->	Rel_Comp -->
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	

Call Rejection

Outgoing call rejected by the network

Synchronous Programming

Application	Device Driver	State	Firmware	Network
cc_MakeCall( ) -->	CRN assigned Call_Outgoing -->	DIALING		
			CALL_OUTGOING -->	Set up -->
	CCEV_ DISCONNECTED <--	*DISCONNECTED	CALL_REJECTION <--	Rel_Comp <--
	ISDN_Block_Ts (sync mode only) -->		Incoming call blocked	
cc_DropCall( )		IDLE	B channel disconnected CALL_DISC	
	CCEV_DROPCALL		CALL_CLEARED <--	
cc_ReleaseCall ( ) -->	Call_Dealloc -->	NULL		
	Driver releases CRN Return <--		F/W releases CRN CALL_DEALLOC_ACK <--	
*Application can use cc_ResultValue( ) to determine the cause value for the disconnect (ERR_ISDN_CAUSE).				

Incoming call rejected by the application

Synchronous Programming

Application	Device Driver	State	Firmware	Network
cc_WaitCall( ) -->	ISDN_Unblock_Ts -->	NULL	Incoming call unblocked	
				Set_Up <--
			*B channel cut-thru CALL_PROCEEDING -->	Proceeding -->
	CRN assigned termination of cc_WaitCall( ) <--	OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) (option) -->				
	Return immediately with DNIS <--			
cc_DropCall( ) -->	Call_Disconnect (cause value ≠ 0) -->			
		IDLE	B channel disconnected CALL_DISC -->	disconnect -->
				Release <--
	termination of cc_DropCall( ) <--		CALL_CLEARED <--	Rel_Comp -->
	ISDN_Block_Ts		Incoming call unblocked	
cc_ReleaseCall( ) -->	Returned immediately <-- Call_Dealloc -->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	
				Set_Up <--
		OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) (option) -->				

Application	Device Driver	State	Firmware	Network
cc_CallAck( ) -->				Proceeding -->
***Application may control CALL_PROCEEDING by adding CCMSK_CALL_PROC and using **cc_CallAck( )** to send event mask, proceeding toward network.				

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
cc_WaitCall( ) -->	ISDN_Unblock_Ts -->	NULL	Incoming call unblocked	
				Set_Up <--
			*CALL_PROCEEDING -->	Proceeding -->
	CRN assigned CCEV_OFFERED <--	OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) (option) -->				
	Return immediately with DNIS <--			
cc_DropCall( ) -->	Call_Disconnect (cause value ≠ 0) -->			
		IDLE	B channel disconnected CALL_DISC -->	disconnect -->
				Release <--
	CCEV_DROPCALL <--		CALL_CLEARED <--	Rel_Comp -->
cc_ReleaseCall( ) -->	Call_Dealloc -->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	
				Set_Up <--
		OFFERED	CALL_INCOMING <--	
cc_GetDNIS( ) (option) -->				
cc_CallAck( )				Proceeding -->
* Application may control CALL_PROCEEDING by adding CCMSK_CALL_PROC and using **cc_CallAck( )** to send event mask, proceeding toward network.				

Glare Condition 1: Call collision occurs after the SETUP message is sent to the network

A glare condition occurs when both an incoming and outgoing call request the same time slot. When glare occurs, the incoming call is assigned the time slot. In this scenario, the firmware detects an incoming SETUP message after transmitting the outgoing SETUP message to the network. In this case, the firmware contains a call reference number for both the incoming and outgoing calls. Therefore, the application must issue **cc_DropCall()** and **cc_ReleaseCall()** to release the outgoing call prior to processing the incoming call. This scenario applies to an exclusive service setting with a firmware-controlled release configuration.

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
cc_MakeCall()-->	Host CRN # 1 assigned Call_Outgoing -->	NULL		
			Firmware CRN #1 is assigned CALL_OUTGOING -->	Set up -->
				Set_Up <--
			*B channel cut-thru CALL_PROCEEDING -->	Proceeding -->
	Host CRN # 2 assigned CCEV_OFFERED <--	OFFERED	Firmware CRN #2 is assigned CALL_INCOMING <--	
cc_AcceptCall() (option) -->	Call_Alert -->		CALL_ALERT -->	Alerting -->
	CCEV_ACCEPTCALL <--	ACCEPTED	CALL_ALERT_ACK <--	
cc_AnswerCall() -->	Call_Connect -->		*B channel cut-thru CALL_CONNECT -->	Connect -->
	CCEV_ANSWERCALL <--	CONNECTED	CALL_CONNECT_ACK <--	Conn_ACK <--
	CCEV_DISCONNECTED <--	**DISCONNECTED	CALL_REJECTION on CRN # 1 <--	Rel_Comp <--

Application	Device Driver	State	Firmware	Network
cc_MakeCall()-->	Host CRN # 1 assigned Call_Outgoing -->	NULL		
			Firmware CRN #1 is assigned CALL_OUTGOING -->	Set up -->
cc_DropCall() -->	Call_Dis -->			
CCEV_DROPCALL <--		IDLE	CALL_CLEARED <--	
cc_ReleaseCall( )	Call_Dealloc -->	NULL		
	Host CRN #1 released		Firmware CRN #1 released CALL_DEALLOC_ACK <--	
*Application may connect a voice channel to the B channel. **Application can use cc_ResultValue( ) to determine cause value for disconnect (ERR_ISDN_CAUSE).				

Glare Condition 2: Call collision occurs before the SETUP message is sent to the network

A glare condition occurs when both an incoming and outgoing call request the same time slot. When glare occurs, the incoming call is assigned the time slot. In this scenario, the firmware detects an incoming SETUP message prior to transmitting the outgoing SETUP message to the network. The application receives a failure to process the request to transmit the SETUP message (CCEV_TASKFAIL event). In this case, the application does not need to issue **cc_DropCall()** and **cc_ReleaseCall()**, and continues to process the incoming call. This scenario applies to an exclusive service setting with a firmware-controlled release configuration.

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
cc_MakeCall()-->	Host CRN # 1 assigned Call_Outgoing -->	NULL		
				Set_Up <--
			*B channel cut-thru CALL_PROCEEDING -->	Proceeding -->
ISDN Application does not need cc_DropCall() or cc_ReleaseCall() and continues processing the incoming call.)	**Library receives ISDN_ERROR and releases host CRN #1 CCEV_TASKFAIL <--	NULL	Firmware detects the collision prior to sending out the SETUP message to the network. Precedence is given to the *incoming* call. Firmware CRN #1 is released. ISDN_ERROR <--	
	Host CRN # 2 assigned CCEV_OFFERED <--	OFFERED	CALL_INCOMING <--	
cc_AcceptCall() (option) -->	Call_Alert -->		CALL_ALERT -->	Alerting -->
	CCEV_ACCEPTCALL <--	ACCEPTED	CALL_ALERT_ACK <--	

cc_AnswerCall() -->	Call_Connected -->		*B channel cut-thru CALL_CONNECT -->	Connect -->
	CCEV_ANSWERCALL <--	CONNECTED	CALL_CONNECT_ACK <--	Conn_ACK <--
*Application may connect a voice channel to the B channel. **Note that the CCEV_TASKFAIL event may occur anytime between calling cc_MakeCall() and the receipt of the CCEV_ANSWERCALL event.				

Simultaneous disconnect (any state)

A simultaneous disconnect condition occurs when both the application and the network attempt to disconnect the call. The following scenarios are written for the asynchronous programming model. For synchronous programming, CCEV_DROPCALL will terminate **cc_DropCall( )**.

The first simultaneous disconnect scenario covers the following conditions:

- **Glare at firmware** - the firmware sees DISCONNECT first.
- **cc_DropCall( ) arrives before release command is sent** - the network disconnects first while **cc_DropCall( )** arrives at the firmware *before* a release command is sent to the network.

NOTE: This scenario assumes the default firmware-controlled disconnect process.

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
	CCEV_ DISCONNECTED <--	DISCONNECTED	CALL_DISC <--	Disconnect <--
***cc_DropCall( )**	Call_Disconnected (cause value =0) -->	IDLE	Firmware does nothing here until Release is sent	Release -->
	****CCEV_DISCONNECTED <--			Release -->
	CCEV_DROPCALL <--		CALL_CLEARED <--	Rel_Comp <--
*****cc_ReleaseCall( )** -->	Call_Dealloc -->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	

*Application should set a "drop call" flag.
**Application should ignore CCEV_DISCONNECTED if "drop call" flag is set.
***cc_ReleaseCall( )** always clears "drop call" flag.

The next scenario covers the following simultaneous disconnect conditions:

- **cc_DropCall( ) arrives after Release command is sent** - the network disconnects first while **cc_DropCall( )** arrives at the firmware after a Release command is sent to the network.
- **Glare happens on the wire** - the firmware sees the **cc_DropCall( )** function call first.

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
		CONNECTED		
	CCEV_DISCONNECTED <--	DISCONNECTED	CALL_DISC <--	Disconnect <--
				Release -->
				Rel_Comp <--
cc_DropCall( ) -->	Call_Disconnected (cause value =0) -->	IDLE		
	CCEV_DROPCALL <--		CALL_CLEARED <--	
	ISDN_Block_Ts (sync model only)		Incoming call blocked	
cc_ReleaseCall( ) -->	Call_Dealloc -->			
	Driver releases CRN Return <--	NULL	F/W releases CRN CALL_DEALLOC_ACK <--	

Initiation of Hold and Retrieve (BRI and PRI DPNSS/Q.SIG Protocols Only)

Hold and retrieve - local initiated

Step	Dialogic API	Action/Result	Dialogic Event
1	--- **cc_HoldCall( )**	CALL CONNECTED -->	---
2	---	CALL HELD <--	--- CCEV_HOLDACK
3	**Unroute SCbus time slot for held call**	: -->	
4	**cc_RetrieveCall( )**		
5		<--	CCEV_RETRIEVEACK
6	Reroute SCbus time slot for retrieved call		
7	---	CALL NOT HELD <--	--- CCEV_HOLDREJ
8	**Take no action**		

1. Place a connected call on hold (**cc_HoldCall( )**).
2. When call is held, application will receive hold acknowledge (CCEV_HOLDACK) event.
3. Application should unroute SCbus time slot for held call.
4. Retrieve a held call (**cc_RetrieveCall( )**).
5. When call is retrieved, application will receive retrieve acknowledge (CCEV_RETRIEVEACK) event.
6. Application should reroute SCbus time slot for retrieved call.
7. When call is not held, application will receive hold reject (CCEV_HOLDREJ) event.
8. Application should take no action on call's SCbus time slot.

NOTE: A request to retrieve a held call cannot be rejected in the DPNSS protocol.

Hold and retrieve - remote initiated

Step	Dialogic API	Action/Result	Dialogic Event
1	---	CALL CONNECTED <--	--- CCEV_HOLDCALL
2	--- **Unroute SCbus time slot for held call**	CALL HELD	---
3	**cc_HoldAck( )**	--> :	
4		<--	CCEV_RETRIEVECALL
5	**Reroute SCbus time slot for retrieved call**		
6	--- **Take no action**	CALL NOT HELD	---
7	**cc_HoldRej( )**	-->	

1. Receives a request to place a connected call on hold (CCEV_HOLDCALL).
2. Application accepts hold request; should unroute SCbus time slot for requested call.
3. Accepts hold request (**cc_HoldAck( )**).
4. Receives request to retrieve a held call (CCEV_RETRIEVECALL).
5. Application receives retrieve request; should reroute SCbus time slot for requested call.
6. Application rejects hold request; should take no action on call's SCbus time slot.
7. Rejects hold request (**cc_HoldRej( )**).

NOTE: A request to retrieve a held call cannot be rejected in the DPNSS protocol.

Network Facility Request or Service

Vari-A-Bill (AT&T Service Only)

Vari-A-Bill is a service option provided only by AT&T at the press time of this document.

Asynchronous Programming

Application	Device Driver	State	Firmware	Network
			CALL_ALERT -->	Alerting -->
			CALL_CONNECT -->	Connect -->
	CCEV_CONNECTED <--	CONNECTED	CALL_CONNECT_ACK <--	CONN_ACK <--
cc_SetBilling( ) -->	ISDN_SETBILLING -->		Request for billing change CALL_FACILITY -->	FAC -->
	CCEV_SetBilling or termination of cc_SetBilling( ) <--		Response from the network; a positive confirm or a reject for "a" reason, etc. ISDN_SETBILLING <--	FAC <--

ANI-on-demand - incoming call (AT&T Service Only)

The following scenario uses **cc_ReqANI( )** to acquire the caller's ID. It differs from the **cc_GetANI( )** function in the way the function is returned.

Asynchronous or Synchronous Programming

Application	Device Driver	State	Firmware	Network
		NULL		
			CRN assigned	Set_Up <--
			*B channel cut-thru CALL_PROCEEDING -->	Proceeding -->
	CCEV_OFFERED <--	OFFERED	CALL_INCOMING <--	
cc_ReqANI( ) -->	ISDN_GetANI -->		CALL_FACILITY -->	FAC -->
	CCEV_GetANI for async programming <-- **or** termination of cc_ReqANI for sync programming <--		CPN/BN information is contained in FAC_ACK message ISDN_RETANI <--	FAC_ACK <-- **or** FAC_REJ <--
* Application may connect a voice channel to the B channel.				

Advice of charge - inbound and outbound call (AT&T Service Only)

Asynchronous Programming: Call disconnected by network

Application	Device Driver	State	Firmware	Network
		CONNECTED		
	CCEV_DISCONNECTED <--	DISCONNECTED	Charge information is part of the DISC message CALL_DISC <--	disconnect <--
				Release -->
cc_GetBilling( ) -->				REL_COMP <--
	billing info (return immediately) <--			
cc_ReleaseCall( ) -->	Call_Dealloc -->			
	Return <--	NULL	CALL_DEALLOC_ACK <--	

Synchronous Programming: Call disconnected by application

Application	Device Driver	State	Firmware	Network
		CONNECTED		
cc_DropCall( ) -->	Call_Disconnected -->			
			B channel disconnected CALL_DISC -->	disconnected -->
				Release <--
	CCEV_DROPCALL <--	IDLE	Charge information is part of the Release message CALL_CLEARED <--	Rel_Comp -->
cc_GetBilling( ) -->				

	Billing info (return immediately.) <--			
cc_ReleaseCall( ) -->	Call_Dealloc -->			
	Return <--	NULL	CALL_DEALLOC_ACK <--	

Two B Channel Transfer (TBCT)

TBCT enables an ISDN PRI user to request the switch to connect together two independent calls on the user's interface. The two calls can be served by the same PRI trunk or by two different PRI trunks that both serve the user.

NOTE: For more on TBCT, refer to the Bellcore Generic Requirements GR-2865-CORE, ISSUE 2, May 1997: Generic Requirements for ISDN PRI Two B Channel Transfer.

If the switch accepts the request, the user is released from the calls and the two other users are connected directly. Billing for the two original calls continues in the same manner as if the transfer had not occurred. As an option, TBCT also allows for transfer notification to the transferred users.

TBCT works only when all of the following conditions are met:

- The user subscribes to TBCT (this feature is supported for 5ESS, 4ESS, and NI2 protocols)
- The two calls are of compatible bearer capabilities
- At least one of the two calls is answered. If the other call is outgoing from the user, it may be either answered or alerting; if the other call is incoming to the user, it must be answered.

The TBCT feature is invoked by sending a FACILITY message to the Network containing, among other things, the Call Reference Values (CRVs) of the two calls to be transferred. The **cc_GetNetCRV( )** function allows applications to query the Dialogic firmware directly for the Network Call Reference Value. (See the **cc_GetNetCRV( )** function description in *Chapter 5. ISDN Function Reference* for more information.)

When a transferred call is disconnected, the network informs the TBCT controller by sending a NOTIFY message with the Network Call Reference Value. The application receives the GCEV_EXTENSION event (with ext_id = GCIS_EXEV_NOTIFY) event.

The following figures provide line diagrams that illustrate the operation of this feature. *Figure 7* shows the invocation of TBCT with notification in which both calls answered.

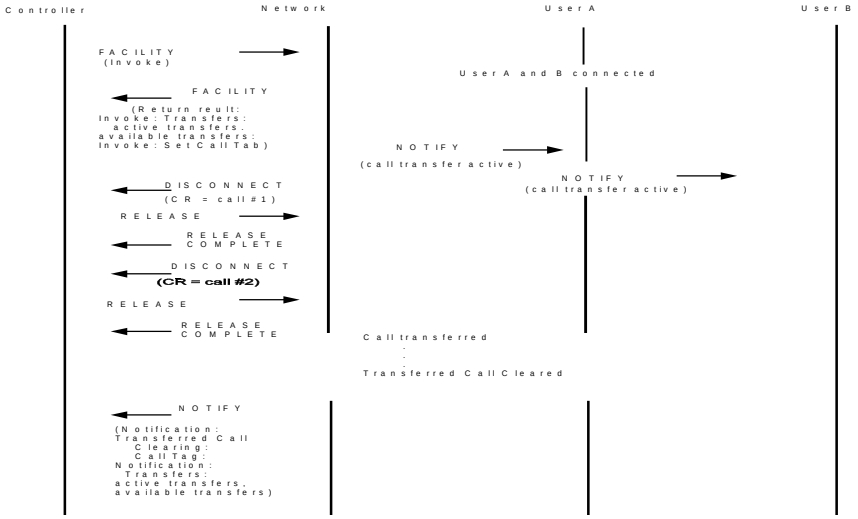

Figure 7. TBCT Invocation with Notification (Both Calls Answered)

Figure 8 shows the invocation of TBCT with notification where one call is answered and the other call is alerting.

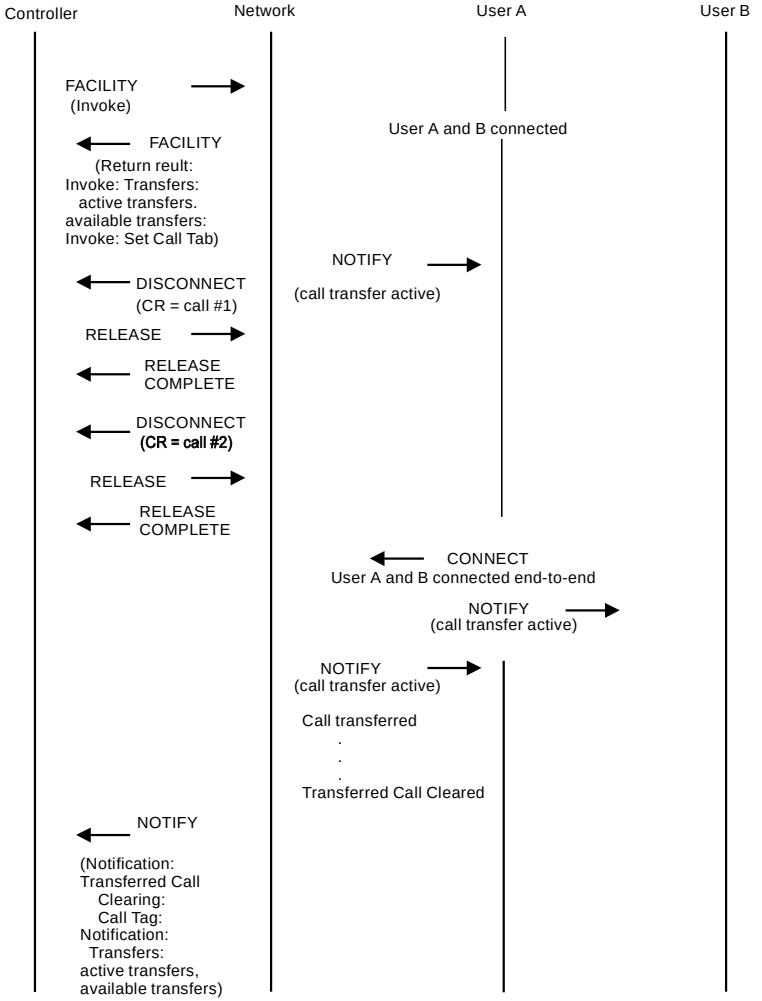

Figure 8. TBCT Invocation with Notification (Call 1 Answered/Call 2 Alerting)

The following call scenario describes the procedures for initiating a TBCT. The scenario is followed by code samples that demonstrate the use of Dialogic API in initiating a TBCT.

Synchronous Programming: Initiating TBCT

Application	Device Driver	State	Firmware	Network
cc_Open( ) Board level device -->		NULL		
	Return with board level device <--			
cc_Open( ) B-channel line devices -->		NULL		
	Return with line device <--			
Users A & B connected				
cc_GetNetCRV (Call #2) -->	ISDN_GETNETCRV -->	CONNECTED (Calls #1 and #2)		
			ISDN_GETNETCRV <--	
	Return with network CRV <--			
cc_SndMsg (FACILITY, Call #1) -->				
	CALL_FACILITY (Call #1) -->		Facility -->	
	CCEV_FACILITY (Call #1) <--			Facility <--
cc_GetCallInfo( ) (Call #1)				
		DISCONNECTED (Call #1)	CALL_DISC (Call #1) <--	Disconnect (Call #1) <--
	CCEV_DISCONNECT (Call #1) <--		Release (Call #1) -->	
cc_DropCall( ) (Call #1) -->	CALL_DISCONNECTED (Call #1) -->	IDLE (Call #1)		Rel_Comp (Call #1) <--

Application	Device Driver	State	Firmware	Network
	CCEV_DROPCALL (Call #1) <--		CALL_CLEARED (Call #1) <--	
cc_ReleaseCall( ) (Call #1) -->	CALL_DEALLOC (Call #1) -->			
	Driver releases CRN return	NULL (Call #1)	CALL_DEALLOC_ACK (Call #1) <--	
		DISCONNECTED (Call #2)	CALL_DISC (Call #2) <--	Disconnect (Call #2) <--
	CCEV_DISCONNECT (Call #2) <--		Release (Call #2) -->	
cc_DropCall( ) (Call #2) -->	CALL_DISCONNECTED (Call #2) -->	IDLE (Call #2)		Rel_Comp (Call #2) <--
	CCEV_DROPCALL (Call #2) <--		CALL_CLEARED (Call #2) <--	
cc_ReleaseCall( ) (Call #2) -->	CALL_DEALLOC (Call #2)			
	Driver releases CRN return	NULL (Call #2)	CALL_DEALLOC_ACK (Call #2) <--	
Users A & B disconnected				
				Notify (CRV = 0) <--
			CALL_NOTIFY (boarddev = dtiB#, CRN = 0) <--	
	CCEV_NOTIFYNULL (boarddev = dtiB#, CRN = 0) <--			
cc_GetNonCallMsg( ) Board level device -->				
	Return with NOTIFY message <--			

The following code samples demonstrate the use of the Dialogic API at various stages of the TBCT call scenario.

1. Opening a board level device:

```
LINEDEV     dti_dev1_hdl;
      .
      .
 rc = cc_Open( &dti_bd_hdl, "dtiB1", 0);
      .
      .
```

2. Retrieving the Network's Call Reference Value:

```
CRN  crn1=0;
 unsigned short  crn1_crv=0;
      .
      .
 rc = cc_GetNetCRV ( crn1, &crn1_crv );
      .
      .
```

3. Building and sending the Facility message to initiate the TBCT for ISDN NI2 protocols on a DMS switch:

```
typedef union {
    struct {
      unsigned char ie_id;                    // Byte 1
      unsigned char length;                   // Byte 2

      unsigned char prot_profile      :5;     // Byte 3, Intel Layout
      unsigned char spare             :2;
      unsigned char extension_1       :1;

      unsigned char comp_type;                // Byte 4
      unsigned char comp_length;              // Byte 5
      unsigned char comp_data[249];           // Bytes 6 to 254
      .
      .
// Preparing the Facility IE Element
tbct_ie.bits.ie_id          = 0x1C;
tbct_ie.bits.length         = 21;

tbct_ie.bits.extension_1    = 1;
tbct_ie.bits.spare          = 0x00;
tbct_ie.bits.prot_profile   = 0x11; // Supplementary Service (ROSE)

tbct_ie.bits.comp_type      = 0xA1; // Invoke
tbct_ie.bits.comp_length    = 18;   // Component Length (Data Only)

tbct_ie.bits.comp_data[0]   = 0x02; // Invoke Identifier, tag
tbct_ie.bits.comp_data[1]   = 0x01; // Invoke Identifier, length
tbct_ie.bits.comp_data[2]   = 0x2E; // Invoke Identifier, invoke ie (varies)
```

```
tbct_ie.bits.comp_data[3]   = 0x06; // Operation Object, tag
tbct_ie.bits.comp_data[4]   = 0x07; // Operation Object, length
tbct_ie.bits.comp_data[5]   = 0x2A; // Operation Object, Operation Value
tbct_ie.bits.comp_data[6]   = 0x86; // Operation Object, Operation Value
tbct_ie.bits.comp_data[7]   = 0x48; // Operation Object, Operation Value
tbct_ie.bits.comp_data[8]   = 0xCE; // Operation Object, Operation Value
tbct_ie.bits.comp_data[9]   = 0x15; // Operation Object, Operation Value
tbct_ie.bits.comp_data[10]  = 0x00; // Operation Object, Operation Value
tbct_ie.bits.comp_data[11]  = 0x08; // Operation Object, Operation Value

tbct_ie.bits.comp_data[12]  = 0x30; // Sequence, tag
tbct_ie.bits.comp_data[13]  = 0x04; // Sequence, length (varies, combined length of Link
& D Channel ID )

tbct_ie.bits.comp_data[14]  = 0x02; // Link ID, tag
tbct_ie.bits.comp_data[15]  = 0x02; // Link ID, length (varies)
tbct_ie.bits.comp_data[16]  = (unsigned char) ((crn2_crv>>8)&0xFF);
// Link ID, linkid value (varies)
tbct_ie.bits.comp_data[17]  = (unsigned char) (crn2_crv&0xFF);
// Link ID, inkid value (varies)

// The D Channel Identifier is Optional
//  tbct_ie.bits.comp_data[18]  = 0x04; // D Channel ID, tag
//  tbct_ie.bits.comp_data[19]  = 0x04; // D Channel ID, length
//  tbct_ie.bits.comp_data[20]  = 0x00; // D Channel ID, dchid (varies)
//  tbct_ie.bits.comp_data[21]  = 0x00; // D Channel ID, dchid (varies)
//  tbct_ie.bits.comp_data[22]  = 0x00; // D Channel ID, dchid (varies)
//  tbct_ie.bits.comp_data[23]  = 0x00; // D Channel ID, dchid (varies)

/*
** Load all the IEs into a single IE block
**   !!NOTE!! - IE must be added in IE ID order!
*/
ie_blk.length = (5 + 18);
for ( ctr = 0; ctr < ie_blk.length; ctr++ ) {
      ie_blk.data[ctr] = tbct_ie.bytes[ctr];
} /* end if */
/*
** Send out a facility message that will execute the transfer
*/
rc = cc_SndMsg( crn2, SndMsg_Facility, &ie_blk );
```

4. Processing the Network response to TBCT request:

```
typedef union {
    struct {
      unsigned char ie_id;                    // Byte 1
      unsigned char length;                   // Byte 2

      unsigned char prot_profile      :5;     // Byte 3, Intel Layout
      unsigned char spare             :2;
      unsigned char extension_1       :1;

      unsigned char comp_type;                // Byte 4
      unsigned char comp_length;              // Byte 5
      unsigned char comp_data[249];           // Bytes 6 to 254

    } bits;
unsigned char bytes[254];
} FACILITY_IE_LAYOUT;
      .
```

```
FACILITY_IE_LAYOUT  *tbct_ie;
      .
IE_BLK ie_list;

ext_id = (EXTENSIONEVTBLK*) (metaevt.extevtdatap);
/*assumes 'metaevt' is filled by gc_GetMetaEvent */
switch ( event )
      {
               .
               .
            case GCEV_EXTENSION:
                  switch (ext_id)
                  {
                        .
                        .
                        .

                  // retrieve facility IE
            for (ie_len = 2; ie_len < ie_list.length;)
            {
                  if (ie_list[ie_len] == FACILITY_IE)
      // found the facility IE
      {

            tbct_ie = &ie_list[ie_len];   // process the Facility IE
tbct_ie_len = tbct_id->length;
#define FACILITY_IE     0x1C
#define RETURN_RESULT   0xA2
#define RETURN_ERROR    0xA3
#define REJECT          0xA4
#define INVOKE_IDEN_TAG 0x02
                              if (tbct_ie->bits.comp_type == RETURN_RESULT)
                              // network accepted TBCT request{
                              .
                              .
// if subscribed to Notification to Controller, check for Invoke component //
if (tbct_ie->bits.comp_data[0] == INVOKE_IDEN_TAG)
{
      invoke_iden = tbct_ie->bits.comp_data[2];
                           // get invoke identifier
}}
                              else if (tbct_ie->bits.comp_type == RETURN_RESULT)
                                    // network accepted TBCT request




}
                  else
                  {
                  /* if it is not facility IE, go to the next IE */
                  /* if this is single byte IE */
                  if (ie_list[ie_len] & 0x80)
                        /* increment by one byte */
                        ie_len = ie_len + 1;
                  else/* otherwise increment by length of the IE */
                        ie_len = ie_len + ie_list[ie_len + 1];
            }
            }
            break;
                  .
                  .
```

5. Processing the Network notification for disconnecting transferred calls:

```
ext_id = (EXTENSIONEVTBLK*) (metaevt.extevtdatap);
/*assumes 'metaevt' is filled by gc_GetMetaEvent */


switch ( event )
       {
               .
               .
               .
            case GCEV_EXTENSION:
                   switch (ext_id);
                             .
                             .
                             .
                     case GCIS_EXEV_NOTIFY:
                          gc_GetInfoElem( boarddev, &ie_list );
                              .
                              .
                              .

                    // retrieve Notification IE
            for (ie_len = 2; ie_len < ie_list.length;)
            {
                  if (ie_list[ie_len] == NOTIFICATION_IE)
            // found the Notification IE
      {


                  }
                  else
                  {
                  /* if it is not facility IE, go to the next IE */
                  /* if this is single byte IE */
                  if (ie_list[ie_len] & 0x80)
                        /* increment by one byte */
                        ie_len = ie_len + 1;
                  else
                        /* otherwise increment by length of the IE */
                        ie_len = ie_len + ie_list[ie_len + 1];
            }
            }
break;
                  .
                  .
                  .
  }
```

Non-Call Associated Signaling (NCAS)

NCAS allows users to communicate by means of user-to-user signaling without setting up a circuit-switched connection (it does not occupy B channel bandwidth). A temporary signaling connection is established and cleared in a manner similar to the control of a circuit-switch connection.

NOTES: **1.** This feature is supported for the 5ESS protocol only

2. For more on NCAS, refer to Technical Reference 41459, AT&T Network ISDN Primary Rate and Special Application Specification.

Since NCAS calls are not associated with any B channel, applications should receive and transmit NCAS calls on the D channel line device. Once the NCAS connection is established successfully, the application can transmit user-to-user messages using the CRN associated with the NCAS call. The Dialogic software and firmware support 16 simultaneous NCAS calls per D channel.

The following figures provide line diagrams that illustrate the operation of the NCAS feature.

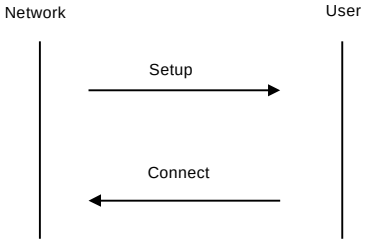

Figure 9. User-Accepted Network-Initiated NCAS Request

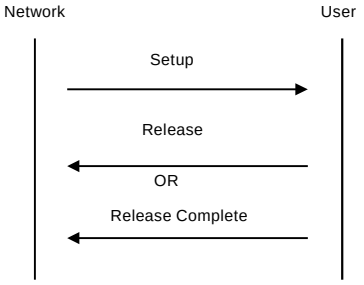

Figure 10. User-Rejected Network-Initiated NCAS Request

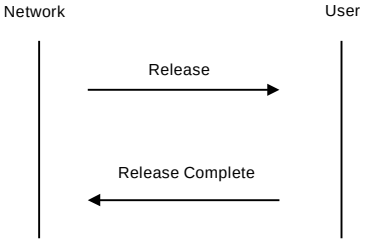

Figure 11. User-Disconnected NCAS Call

The following scenarios demonstrate the procedures for a user-initiated and a network-initiated NCAS call.

User-initiated call

In the following scenario, the user initiates and disconnects the NCAS call for dtiB1.

Synchronous Programming

Application	Device Driver	State	Firmware	Network
cc_Open( ) -->		NULL		
	Return with line device <--			
Set up NCAS call parameter in MAKECALL_BLK				

Application	Device Driver	State	Firmware	Network
	Return <--			
cc_MakeCall( ) D-channel line devices (dtiB1T24) -->				
	CALL_OUTGOING -->		Setup -->	
	CCEV_CONNECTED <--	CONNECTED		Connect <--
NCAS call connected				
cc_SetInfoElem( ) setup user-to-user information, D-channel line devices (dtiB1T24) -->				
cc_SndMsg( ) send User-to-user signaling -->	CALL_UUI -->		UUI -->	
	CCEV_USRINFO <--		CALL_UUI <--	UUI <--
cc_GetCallInfo( ) retrieve user-to-user information -->				
cc_DropCall( ) -->	CALL_DISCONNECTED -->	IDLE	Release -->	
	CCEV_DROPCALL <--		CALL_CLEARED <--	
cc_ReleaseCall( ) -->	CALL_DEALLOC -->			Rel_Comp <--
	Driver releases CRN return	NULL	CALL_DEALLOC_ACK <--	

The following code samples demonstrate the use of the Dialogic API at various stages of the NCAS call scenario.

1. Opening a D channel line level device:

```
LINEDEV     D_chan_dev1_hdl;
      .
      .
 rc = cc_Open( &D_chan_dev1_hdl, "dtiB24", 0);
      .
```

2. Setting up the MAKECALL_BLK for an NCAS call:

```
MAKECALL_BLK *makecallp;
      .
      .
// initialize makecall block
      makecallp->isdn.BC_xfer_cap              =  BEAR_CAP_UNREST_DIG;
      makecallp->isdn.BC_xfer_mode             =  ISDN_ITM_PACKET;
      makecallp->isdn.BC_xfer_rate             =  PACKET_TRANSPORT_MODE;
      makecallp->isdn.usrinfo_layer1_protocol  =  NOT_USED;
      makecallp->isdn.usr_rate                 =  NOT_USED;
      makecallp->isdn.destination_number_type  =  NAT_NUMBER;
      makecallp->isdn.destination_number_plan  =  ISDN_NUMB_PLAN;
      makecallp->isdn.destination_sub_number_type     =  OSI_SUB_ADDR;
      makecallp->isdn.destination_sub_phone_number[0] = '1234'
      makecallp->isdn.origination_number_type  = NAT_NUMBER;
      makecallp->isdn.origination_number_plan  = ISDN_NUMB_PLAN;
      makecallp->isdn.origination_phone_number[0]   =  '19739903000'
      makecallp->isdn.origination_sub_number_type   = OSI_SUB_ADDR;
      makecallp->isdn.origination_sub_phone_number[0] =  '5678'
      makecallp->isdn.facility_feature_service        =  ISDN_SERVICE;
      makecallp->isdn.facility_coding_value    =  ISDN_SDN;
// or ISDN_ACCUNET, please check with your service provider
      makecallp->isdn.usrinfo_bufp             =  NULL;
      makecallp->isdn.nsfc_bufp                =  NULL;
            .
            .
```

Network-initiated call

In the following scenario, the network initiates and disconnects the NCAS call for dtiB1.

Synchronous Programming

Application	Device Driver	State	Firmware	Network
cc_Open( ) D channel line devices (dtiB1T24) -->		NULL		
	Return with line device <--			
cc_WaitCall( ) D-channel line devices (dtiB1T24) -->				
	CCEV_OFFERED D-channel line devices (dtiB1T24) <--	OFFERED	CALL_INCOMING <--	Setup <--
cc_AnswerCall( ) -->	Call_Connect -->	CONNECTED	Connect -->	
NCAS call connected				
cc_SetInfoElem( ) setup user-to-user information, D channel line devices (dtiB1T24) -->				
cc_SndMsg( ) send user-to-user signaling -->	CALL_UUI -->		UUI -->	
	CCEV_USRINFO <--		CALL_UUI <--	UUI <--
cc_GetCallInfo( ) retrieve user-to-user information				
		IDLE		Release <--
			Release Comp -->	
	CCEV_DISCONNECT <--		CALL_DISC <--	
cc_DropCall( ) -->	CALL_DISCONNECTED -->			

	CCEV_DROPCALL <--		CALL_CLEARED <--	
cc_ReleaseCall( ) -->	CALL_DEALLOC<br-->			
	Driver releases CRN return	NULL	CALL_DEALLOC_ ACK <--	

Appendix B - DPNSS Call Scenarios

This appendix describes call scenarios that are specific to the DPNSS protocol. Each scenario provides a table that illustrates the Dialogic Application Programming Interfaces (APIs) issued by the application to either initiate a transaction or to respond to an external action, and the resulting Dialogic event that is returned to the application. A step-by-step description of the scenario follows the table for further clarification.

The following call scenarios are provided in this chapter in the order listed below:

- Executive Intrusion - Normal
- Executive Intrusion - With Prior Validation
- Local Diversion - Outbound
- Local Diversion - Inbound
- Remote Diversion - Outbound
- Remote Diversion - Inbound
- Virtual Call - Outbound
- Virtual Call - Inbound

Executive Intrusion - Normal

Step	Dialogic API	Action/Result	Dialogic Event
1	**cc_MakeCall( )** (with Intrusion IE)	-->	
2		<--	CCEV_PROCEEDING
3	---	INTRUSION SUCCEEDED <--	--- CCEV_CONNECTED
4	---	INTRUSION FAILED <--	--- CCEV_DISCONNECT

1. Places an outgoing call (**cc_MakeCall( )**) to a busy extension with intrusion information set to "Normal." See *Appendix C* for the format of Intrusion IE.
2. Receives call proceeding (CCEV_PROCEEDING).
3. Receives call connected (CCEV_CONNECTED) event. Call successfully intruded.
4. Receives call disconnect (CCEV_DISCONNECT) event. Call was not intruded.

Executive intrusion - with prior validation

Step	Dialogic API	Action/Result	Dialogic Event
1	**cc_MakeCall( )** (with Intrusion IE)	-->	
2		<--	CCEV_PROCEEDING (with Busy IE)
3	**cc_SendMsg( )** (SndMsg_Intrude)	-->	
4	---	INTRUSION SUCCEEDED <--	--- CCEV_CONNECTED
5	---	INTRUSION FAILED <--	--- CCEV_DISCONNECT

1. Places an outgoing call (**cc_MakeCall( )**) to a busy extension with intrusion information set to "Prior Validation." See *Appendix C* for the format of Intrusion IE.
2. Receives call proceeding (CCEV_PROCEEDING) event with indication that remote party was busy. Use **cc_GetSigInfo( )** to retrieve Busy IE. See *Appendix C* for Busy IE's format.
3. Sends intrude request using (**cc_SndMsg( )**). See the **cc_SndMsg( )** function description for details.
4. Receives call connected (CCEV_CONNECTED) event. Call successfully intruded.
5. Receives call disconnect (CCEV_DISCONNECT) event. Call was not intruded.

Local diversion - outbound

Step	Dialogic API	Action/Result	Dialogic Event
1	**cc_MakeCall( )**	-->	
2		<--	CCEV_PROCEEDING (with Diversion IE, diversion location: DIVERT_LOCAL)
3		<--	CCEV_CONNECTED

1. Places an outgoing call (**cc_MakeCall( )**).
2. Receives call proceeding (CCEV_PROCEEDING) event with indication that call was diverted to another location. Use **cc_GetSigInfo( )** to retrieve Diversion IE. See *Appendix C* for Diversion IE's format.
3. Receives call connected (CCEV_CONNECTED) event. Call established.

Local diversion - inbound

Step	Dialogic API	Action/Result	Dialogic Event
1		<--	CCEV_OFFERED

2	**cc_SndMsg( )** (SndMsg_Divert, diversion location: DIVERT_LOCAL)	-->	
3	**cc_AnswerCall( )**	-->	
4		<--	CCEV_ANSWERED

1. Receives incoming call (CCEV_OFFERED) event.
2. Diverts incoming call (**cc_SndMsg( )**) to different extension. Use **cc_SndMsg( )** to divert the incoming call. See the **cc_SndMsg( )** function description for details.
3. Answer incoming call (**cc_AnswerCall( )**).
4. Receives call connected (CCEV_ANSWERED) event.

Remote diversion - outbound

Step	Dialogic API	Action/Result	Dialogic Event
1	**cc_MakeCall( )**	-->	
2	**cc_SndMsg( )** (SndMsg_Divert, diversion location: DIVERT_REMOTE)	<--	CCEV_PROCEEDING (with Diversion IE, diversion location: DIVERT_REMOTE)
3	**cc_DropCall( )**	-->	
4		<--	CCEV_DROPCALL
5	**cc_ReleaseCall( )**	-->	
6	**cc_MakeCall( )** (with Diversion IE)	-->	
	---	DIVERT SUCCEEDED	---
7		<--	CCEV_PROCEEDING
8		<--	CCEV_DIVERTED
9		<--	CCEV_CONNECTED
	---	DIVERT FAILED	---
10		<--	CCEV_DISCONNECT

1. Party 1 calls Party 2 by issuing **cc_MakeCall( )**.
2. Party 1 receives CCEV_PROCEEDING event from Party 2 with indication that call needs to be diverted to Party 3. Diversion IE will contain the telephone number of Party 3. See *Appendix C* for Diversion IE's format.
3. Party 1 disconnects original call to Party 2.
4. Party 1 receives call disconnect (CCEV_DROPCALL) event from Party 2.
5. Releases first call.
6. Party 1 diverts call to Party 3. Calling party number IE should contain Party 3's telephone number. Diversion IE should contain Party 2's telephone number. See the **cc_SetInfoElem( )** function description for details on sending Diversion IE.
7. Party 1 receives proceeding (CCEV_PROCEEDING) event from Party 3.
8. Party 1 receives divert successful (CCEV_DIVERTED) event from Party 3.
9. Party 1 receives call connected (CCEV_CONNECTED) event from Party 3. Call successfully diverted.
10. Party 1 receives divert failed (CCEV_DISCONNECT) event from Party 3. Call was not diverted.

Remote diversion - inbound

Step	Dialogic API	Action/Result	Dialogic Event
1		<--	CCEV_OFFERED
2	**cc_SndMsg( )** (SndMsg_Divert, diversion location: DIVERT_REMOTE)	-->	
3		<--	CCEV_DISCONNECT
4	**cc_DropCall( )**	-->	
5		<--	CCEV_DROPCALL
6	**cc_ReleaseCall( )**	-->	

1. Party 2 receives incoming call (CCEV_OFFERED) from Party 1.

2. Party 2 diverts incoming call to Party 3. Send Party 3's telephone number as Diversion number. See *Appendix C* for the format of SndMsg_Divert message.
3. Party 1 disconnects call to Party 2.
4. Party 2 drops call (**cc_DropCall( )**).
5. Party 2 receives drop call event (CCEV_DROPCALL) event from Party 1.
6. Releases call.

Transfer

Step	Dialogic API	Action/Result	Dialogic Event
1		<--	CCEV_OFFERED (CRN 1)
2	**cc_AnswerCall( )** (CRN 1)	-->	
3	**cc_HoldCall( )** (CRN 1)	-->	
4		<--	CCEV_HOLDACK (CRN 1)
5	**cc_MakeCall( )** (CRN 2, with Inquiry IE)	-->	
6		<--	CCEV_PROCEEDING (CRN 2 with Inquiry IE)
7		<--	CCEV_CONNECTED (CRN 2 with Inquiry IE)
8	**cc_SndMsg( )** (SndMsg_Transfer, CRN 1)	-->	
9	**cc_SndMsg( )** (SngMsg_Transfer, CRN 2)	-->	
10		<--	CCEV_TRANSFERACK (CRN 1)
11		<--	CCEV_TRANSFERACK (CRN 2)
12	**Cross connect CRN 1's and CRN 2's SCbus time slot**		
13		<--	CCEV_TRANSIT (CRN 1)
14	**cc_SndMsg( )** (SndMsg_Transit, CRN 2)	-->	
15		<--	CCEV_TRANSIT (CRN 2)
16	**cc_SndMsg( )** (SndMsg_Transit, CRN 1)	-->	
17		<--	CCEV_DISCONNECT (CRN 1)

18	**cc_DropCall( )** (CRN 1)	-->	
19		<--	CCEV_DROPCALL (CRN 1)
20	**cc_ReleaseCall( )** (CRN 1)	-->	
21		<--	CCEV_DISCONNECT (CRN 2)
22	**cc_DropCall( )** (CRN 2)	-->	
23		<--	CCEV_DROPCALL (CRN 2)
24	**cc_ReleaseCall( )** (CRN 2)	-->	

1. Party 2 receives incoming call (CCEV_OFFERED) from Party 1.
2. Party 2 answers call (**cc_AnswerCall( )**) from Party 1.
3. Party 2 places call on hold (**cc_HoldCall( )**).
   NOTE: Some switches may not support Hold.
4. Party 2 receives call on hold acknowledge (CCEV_HOLDACK) event.
5. Party 2 places an inquiry call (**cc_MakeCall( )**) to Party 3. Application should use Party 1's telephone number as the calling party number and Party 3's telephone number as called party number. See *Appendix C* for the Inquiry IE format.
6. Party 2 receives call proceeding (CCEV_PROCEEDING) event with Inquiry IE from Party 3. See *Appendix C* for the Inquiry IE format.
7. Party 2 receives call connected (CCEV_CONNECTED) event with Inquiry IE from Party 3. See *Appendix C* for the Inquiry IE format.
8. Party 2 sends transfer request (**cc_SndMsg( )**) to Party 1 with TRANSFER_ORIG as transfer direction. See *Appendix C* for message format.
9. Party 2 sends transfer request (**cc_SndMsg( )**) to Party 3 with TRANSFER_TERM as transfer direction. See *Appendix C* for message format.
10. Party 2 receives transfer acknowledge (CCEV_TRANSFERACK) from Party 1.
11. Party 2 receives transfer acknowledge (CCEV_TRANSFERACK) from Party 3. Transfer completed. At this time, Party 2 loses control of the call.
12. Application should cause Party 1 to listen to Party 2's transmit time slot and Party 2 to listen to Party 1's transmit time slot.

13. Party 2 receives transit (CCEV_TRANSIT) event from Party 1. Party 2 should retrieve the content of the Transmit Message using **cc_GetSigInfo( )**.
14. Party 2 sends content of Transmit Message (unchanged) from Party 1 to Party 3 (**cc_SndMsg( )**). See *Appendix C* for message format.
15. Party 2 receives transit (CCEV_TRANSIT) event from Party 3. Party 2 should retrieve the content of the Transit Message using **cc_GetSigInfo( )**.
16. Party 2 sends content of Transit Message (unchanged) from Party 3 to Party 1 (**cc_SndMsg( )**). See *Appendix C* for message format.
17. Party 2 receives disconnect all (CCEV_DISCONNECT) event from Party 1.
18. Party 2 drops call (**cc_DropCall( )**) to Party 1.
19. Party 2 receives drop call event (CCEV_DROPCALL) event from Party 1.
20. Releases call to Party 1.
21. Party 2 receives disconnect all (CCEV_DISCONNECT) event from Party 3.
22. Party 2 drops call (**cc_DropCall( )**) to Party 3.
23. Party 2 receives drop call event (CCEV_DROPCALL) event from Party 3.
24. Releases call to Party 3.

NOTES: **1.** Steps 3 and 4 are optional and need not be carried out on most switches.

2. Steps 12 through 16 may be repeated multiple times depending on when or whether the distant PBX supports Route Optimization. When Route Optimization occurs, or if either end of the transferred call is terminated, the call flow proceeds to step 17.

Virtual call -outbound

Step	Dialogic API	Action/Result	Dialogic Event
1	**cc_MakeCall( )** (with Virtual Call IE)	-->	
2		<--	CCEV_DISCONNECT
3	**cc_DropCall( )**	-->	

4		<--	CCEV_DROPCALL
5	**cc_ReleaseCall( )**	-->	

1. Places an outgoing call (**cc_MakeCall( )**) with Virtual Call IE and any other information set, such as NSI strings or Extension Status. See *Appendix C* for the format of Virtual Call IE.
2. Receives call disconnected (CCEV_DISCONNECT) event. Use **cc_ResultValue( )** to retrieve the clearing cause. RESP_TO_STAT_ENQ means the call was Acknowledged and FACILITY_REJECT means the call was Rejected.
3. Issues **cc_DropCall( )**.
4. Receives drop call (CCEV_DROPCALL) event.
5. Issues **cc_ReleaseCall( )**.

Virtual call - inbound

Step	Dialogic API	Action/Result	Dialogic Event
1		<--	CCEV_OFFERED (with Virtual Call IE)
2	**cc_DropCall( )**	-->	
3		<--	CCEV_DROPCALL
4	**cc_ReleaseCall( )**	-->	

1. Receives call offer (CCEV_OFFERED) event with indication that this is a virtual call. Use **cc_GetSigInfo( )** to retrieve Virtual Call IE and any other information, such as NSI strings.
2. Issues **cc_DropCall( )** with clearing cause set to RESP_TO_STAT_ENQ to acknowledge the call or FACILITY_REJECT to reject the call.
3. Receives drop call (CCEV_DROPCALL) event.
4. Issues **cc_ReleaseCall( )**.

Appendix C - IEs and ISDN Message Types for DPNSS

This appendix lists the information elements (IEs) and ISDN message types in the ISDN software library that support the DPNSS protocol.

Information Elements for cc_GetCallInfo( ) and cc_GetSigInfo( )

The following tables describe the different types of IEs that can be retrieved for DPNSS using the **cc_GetCallInfo( )** and **cc_GetSigInfo( )** functions.

Intrusion IE:

Field	Description	Field Selection	Definition
1. IE ID	Busy IE ID	BUSY_IE	Busy IE value for the CCEV_PROCEEDING event indicates that the called party is busy

Diversion IE:

Field	Description	Field Selection	Definition
1. IE ID	Diversion IE ID	DIVERSION_IE	1. A DIVERSION_IE value in a CCEV_OFFERED event provides information about "diverted from" party. 2. A DIVERSION_ IE value in a CCEV_PROCEEDING event provides information about "divert to" party.
2. Data	Diversion IE Length	2 + length of Diversion Number	Number of data bytes in this IE

Field	Description	Field Selection	Definition
3. Data	Diversion Type	DIVERT_IMMEDIATE	Diverted immediately
		DIVERT_ON_BUSY	Diverted when called party was busy
		DIVERT_NO_REPLY	Diverted when called party did not answer
4. Data	Diversion Location	DIVERT_LOCAL	Local diversion
		DIVERT_REMOTE	Remote diversion
5. Data	Diversion Number	ASCII string	Diverted number

Diversion Validation IE:

Field	Description	Field Selection	Definition
1. IE ID	Diversion Validation IE ID	DIVERSION_VALIDATION_IE	When this IE is part of a CCEV_OFFERED event, it indicates that the diversion number needs to be validated.

Transit IE:

Field	Description	Field Selection	Definition
1. IE ID	Transit IE ID	TRANSIT_IE	This IE is received with a CCEV_TRANSIT event.
2. Data	Transit IE Length	Length of Transit data	Number of data bytes in this IE

Field	Description	Field Selection	Definition
3. Data	Transit Data	data	Transit data that needs to be sent to the other transfer party.

Text Display IE:

Field	Description	Field Selection	Definition
1. IE ID	Text Display IE ID	TEXT_DISPLAY_IE	This IE can be part of a CCEV_OFFERED event.
2. Data	Text Display IE Length	1 + length of Text Display string	Number of data bytes for this IE.
3. Data	Text Display Message Type	TEXT_TYPE_NOT_PRESENT TEXT_TYPE_NAME TEXT_TYPE_MESSAGE TEXT_TYPE_REASON	Associated text is of no particular type Associated text is a name Associated text is a message Associated text is a reason
4. Data	Text Display String	ASCII string	Text Display string. The '*' and '#' symbols cannot be used directly; 0x01 and 0x02 values should be substituted respectively

Network Specific Indications (NSI) IE:

Field	Description	Field Selection	Definition
1. IE ID	NSI IE ID	NSI_IE	This IE can be part of any event including the CCEV_NSI event.
2. Data	NSI IE Length	2 + Length of Network Specific Indications (NSI) string	Number of data bytes for this IE
3. Data	NSI Message Type	NSI_EEM NSI_LLM	End-to-end message Link-to-link message
4. Data	NSI String Length	Length of Network Specific Indications (NSI) string	Length of next NSI string
5. Data	NSI String	ASCII string	Network Specific Indications string

NOTE: NSI IE fields 4 and 5 can be repeated multiple times, as needed.

Extension Status IE:

Field	Description	Field Selection	Definition
1. IE ID	Extension Status IE ID	EXTENSION_STATUS_IE	This IE is used in conjunction with the Virtual Call IE to inquire about the current status of an extension.

Virtual Call IE:

Field	Description	Field Selection	Definition
1. IE ID	Virtual Call IE ID	VIRTUALCALL_IE	This IE, when part of a CCEV_OFFERED event, indicates a virtual call.

Information Elements for cc_SetInfoElem( )

The following tables describe the information elements that can be set for DPNSS using the **cc_SetInfoElem( )** function.

Intrusion IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	4	Required value
2. IE ID	Intrusion IE ID	INTRUSION_IE	Use with the **cc_MakeCall( )** function to indicate intrusion privilege.
3. Data	Intrusion IE Length	2	Number of data bytes for this IE
4. Data	Intrusion Type	INTRUDE_PRIOR_VALIDATION INTRUDE_NORMAL	Validate intrusion level prior to intrude Intrude (without validation)
5. Data	Intrusion Level	INTRUSION_LEVEL_1 INTRUSION_LEVEL_2 INTRUSION_LEVEL_3	Intrusion protection level 1 Intrusion protection level 2 Intrusion protection level 3

Diversion IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	4 + length of Diversion Number	
2. Data	Diversion IE ID	DIVERSION_IE	Use with the **cc_MakeCall( )** function to indicate why the call was diverted and from where the call was diverted.
3. Data	Diversion IE Length	2 + length of Diversion Number	Number of data bytes for this element
4. Data	Diversion Type	DIVERT_IMMEDIATE DIVERT_ON_BUSY DIVERT_NO_REPLY	Diverted immediately Diverted when called party was busy Diverted when called party did not answer
5. Data	Diversion Location	DIVERT_LOCAL DIVERT_REMOTE	Local diversion Remote diversion
6. Data	Diversion Number	ASCII string	Diverted number

Diversion Bypass IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	1	Required value

Field	Description	Field Selection	Definition
2. Data	Diversion Bypass IE ID	DIVERSION_BYPASS_IE	Use with the **cc_MakeCall( )** function to indicate that diversion is not allowed.

Inquiry IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	1	
2. Data	Inquiry IE ID	INQUIRY_IE	Use with the **cc_MakeCall( )** function to indicate three-party call.

Extension Status IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	1	Required value
2. Data	Extension Status IE ID	EXTENSION_STATUS_IE	Use in conjunction with the Virtual Call IE to inquire about the current status of an extension.

Virtual Call IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	1	Required value
2. Data	Virtual Call IE ID	VIRTUALCALL_IE	Use with the **cc_MakeCall( )** function to indicate virtual call.

Text Display IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	3 + length of Text Display String	Required value
2. Data	Text Display IE ID	TEXT_DISPLAY_IE	This IE can be part of a CCEV_OFFERED event.
3. Data	Text Display IE Length	1 + length of Text Display string	Number of data bytes for this information element
4. Data	Text Display Message Type	TEXT_TYPE_NOT_PRESENT	Associated text is of no particular type
		TEXT_TYPE_NAME	Associated text is a name
		TEXT_TYPE_MESSAGE	Associated text is a message
		TEXT_TYPE_REASON	Associated text is a reason

Field	Description	Field Selection	Definition
5. Data	Text DISPLAY String	ASCII string	Text Display string. The '*' and '#' symbols cannot be used directly; 0x01 and 0x02 values are substituted respectively

Network Specific Indications (NSI) IE:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	4 + length of NSI String	Required value
2. Data	NSI IE ID	NSI_IE	Identifies the Network Specific Indications IE.
3. Data	NSI IE Length	2 + length of NSI string	Number of data bytes for this IE
4. Data	NSI Message Type	NSI_EEM NSI_LLM	End-to-end message Link-to-link message
5. Data	NSI String Length	Length of Network Specific Indications string	Length of next NSI string
6. Data	NSI String	ASCII string	Network Specific Indications string

NOTE: NSI IE fields 5 and 6 can be repeated multiple times, as needed.

DPNSS Message Types for cc_SndMsg( )

The following tables describe the ISDN message types that support the DPNSS protocol.

SndMsg_Divert:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	4 + length of Diverted Number	Required value
2. Data	Diversion IE ID	DIVERSION_IE	Identifies the Diversion IE
3. Data	Diversion IE Length	2 + length of Diverted Number	Number of data bytes for this IE
4. Data	Diversion Type	DIVERT_IMMEDIATE DIVERT_ON_BUSY DIVERT_NO_REPLY	Diverted immediately Diverted when called party was busy Diverted when called party did not answer
5. Data	Diversion Location	DIVERT_LOCAL DIVERT_REMOTE	Local diversion Remote diversion
6. Data	Diversion Number	ASCII string	Diverted number

SndMsg_Intrude:

Field	Description	Field Selection	Definition
1. Length	Total number of bytes of the following data field	3	Required value
2. Data	Intrude IE ID	INTRUDE_IE	Identifies the Intrude IE
3. Data	Intrude IE Length	1	Number of data bytes for this IE

Field	Description	Field Selection	Definition
4. Data	Intrude Type	INTRUDE INTRUDE_WITHDRAW	Intrude Withdraw intrusion

SndMsg_NSI:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	4 + length of NSI String	Required value
2. Data	NSI IE ID	NSI_IE	Identifies the NSI IE
3. Data	NSI IE Length	2 + length of Network Specific Indications (NSI) string	Number of data bytes for this IE
4. Data	NSI Message Type	NSI_EEM NSI_LLM	End-to-end message Link-to-link message
5. Data	NSI String Length	Length of Network Specific Indications (NSI) string	Length of next NSI string
6. Data	NSI String	ASCII string	Network Specific Indications string

NOTE: NSI IE fields 5 and 6 can be repeated multiple times as needed.

SndMsg_Transfer:

Field	Description	Field Selection	Definition
1. Length	Total bytes of the following data field	3	Required value

Field	Description	Field Selection	Definition
2. Data	Transfer IE ID	TRANSFER_IE	Identifies the Transfer IE
3. Data	Transfer IE Length	1	Number of data bytes for this IE
4. Data	Transfer Direction	TRANSFER_ORIG TRANSFER_TERM	Originating end Terminating end

SndMsg_Transit:

1. Length	Total bytes of the following data field	2 + Length of Transit Data	Required value
2. Data	Transit IE ID	TRANSIT_IE	Identifies the Transit IE
3. Data	Transit IE Length	Length of Transit Data	Number of data bytes for this information element
4. Data	Transit Data	data	Transit data received from a CCEV_TRANSIT event

Appendix D – BRI Supplemental Services

The ISDN API functions allow BRI boards to perform the following Supplemental Services:

- Called/Calling Party Identification
- Subaddressing
- Hold/Retrieve
- Call Transfer
- Message Waiting

Call Hold and Retrieve are invoked using the following API functions (see the appropriate function descriptions in *Chapter* 5. ISDN Function Reference for more information):

- **cc_HoldAck( )**
- **cc_HoldCall( )**
- **cc_HoldRej( )**
- **cc_RetrieveAck( )**
- **cc_RetrieveCall( )**
- **cc_RetrieveRej( )**

The other Supplemental Services are invoked by sending information from the board to the PBX using an appropriate API function. This information is sent as the part of the Layer 3 frame called the **Information Element** (see *Section* 3.2.2. Network Layer (Layer 3) Frames for more information). In order for the PBX to interpret the Information Elements as Supplemental Service requests, the Information Elements must be sent as Facility Messages.

The following functions can be used to send Facility Messages:

- **cc_SndMsg( )** - Sends a call state associated message to the PBX.

- **cc_SndNonCallMsg( )** - Sends a non-Call State related message to the PBX. This function does not require a call reference value.
- **cc_SetInfoElem( )** - Sets an information element (IE) allowing the application to include application-specific ISDN information elements in the next outgoing message.

The following functions are used to retrieve Facility Messages:

- **cc_GetCallInfo( )** - Retrieves the information elements associated with the CRN.
- **cc_GetNonCallMsg( )** - Retrieves a non-Call State related ISDN messages to the PBX.

The **cc_SndMsg( )** and **cc_SndNonCallMsg( )** functions are used to send Facility Messages or Notify Messages to the PBX. The Facility Message (as defined in ETS 300-196-1) is composed of the following elements:

- Protocol discriminator
- Call reference
- Message type
- Facility Information Element

The Supplemental Service to be invoked and its associated parameters are specified in the Information Element. This information is PBX-specific and should be provided by the PBX manufacturer. Facility Messages are sent using the **cc_SndMsg( )** or **cc_SndNonCallMsg( )** function with msg_type = SndMsg_Facility. These functions

1. format the Facility Message, inserting the protocol discriminator, call reference number (only for **cc_SndMsg( )**) and message type elements
2. add the Information Element data (stored in an application buffer)
3. send all the information to the PBX

The PBX, in turn, interprets and acts on the information, and sends a reply to the BRI board.

As an example, to invoke Supplemental Service 'X', the **cc_SndMsg( )** function with msg_type = SndMsg_Facility could be used. The Information Element would be defined in a data structure as follows:

```
ieblk.length = 11;
ieblk.data[0]  = 0x1c;  /* IE Identifier */
ieblk.data[1]  = 0x09;  /* Length of information */
ieblk.data[2]  = 0x91;  /* Protocol Profile */

/* information */
ieblk.data[3]  = 0xa1;  /* Component Type */
ieblk.data[4]  = 0x06;  /* Component Length */
ieblk.data[5]  = 0x02;  /* invoke tag id */
ieblk.data[6]  = 0x01;  /* invoke tag length */
ieblk.data[7]  = 0x00;  /* invoke id */
ieblk.data[8]  = 0x02;  /* operation tag */
ieblk.data[9]  = 0x01;  /* operation length */
ieblk.data[10] = 0x06;  /* operation */
```

NOTE: The information included in the Information Element is dependent on the Supplemental Service being invoked.

The data sent to the switch would be formatted as follows:

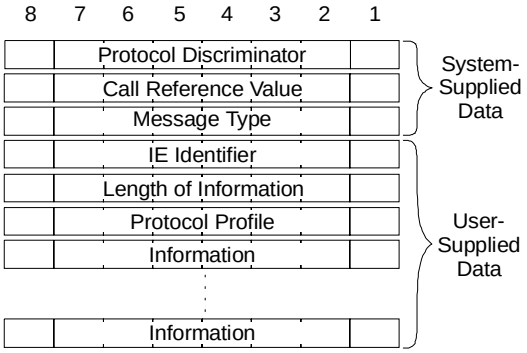

Figure 12. Information Element Format

Information elements can also be sent using the **cc_SetInfoElem( )** function, which allows the BRI board to send application-specific information elements in the next outgoing message. (For more information, see the **cc_SetInfoElem( )** function description.)

When a Supplemental Service is invoked, the network may return a NOTIFY Message to the user. This message can be retrieved using the **cc_GetCallInfo( )** function.

The Notify Message (as defined in ETS 300-196-1) is composed of the following elements:

- Protocol discriminator
- Call reference
- Message type
- Notification Indicator

The Notify message is coded as follows:

8	**7**	**6**	**5**	**4**	**3**	**2**	**1**
x	x	x	x	x	x	x	x
Protocol discriminator							
x	x	x	x	x	x	x	x
Call reference							
x	x	x	x	x	x	x	x
Message Type							
0	0	1	0	0	1	1	1
	Notification Indicator Information element identifier						
0	0	0	0	1	0	0	1
Length of Notification Indicator contents							
1/1	x	x	x	x	x	x	x
ext.	Notification Description						
0	x	x	x	x	x	x	x
ext.	Notification Description						
1	0	1	0	0	0	0	1
Notification Data Structure							

Coding requirements for other supported Supplemental Services are listed in Table 60.

Table 60. ETSI Specification Cross-Reference for Supplemental Services

Supplementary Service/Description	**ETS 300 Specification**
Explicit Call Transfer - enables a user (user A) to transform two of that user's calls (an active call and a held call), each of which can be an incoming call or an outgoing call, into a new call between user B and user C. "Call Transferred Alerting" and "Call Transferred Active" messages are returned by the network to the user.	367/369/369
Call Hold/ Retrieve - allows a user to interrupt communications on an existing call and then subsequently, if desired, re-establish communications. When on Hold, the user may retrieve that call from hold, originate a new call, retrieve another call, or establish connection to an incoming call, for example, a waiting call.	139/140/141
Subaddressing (allows direct connection to individual extensions or devices sharing the same phone number, or, as a proprietary messaging mechanism). Provides additional addressing above the ISDN number of the called user.	059/060/061
Called/Calling Party Identification (CLIP) - Provides the calling user's ISDN number and subaddress information to the called user. This information is sent in the "Setup message" (see ETS300 102-1) by the calling user to the switch, and from the switch to the called user.	089/091/092
Called/Calling Party Identification (CLIR) - Restricts presentation of the calling user's ISDN number to the called user.	090/091/093
Called/Calling Party Identification (COLP) - Provides the calling user's ISDN number to the called user.	094/096/097
Called/Calling Party Identification (COLR) - restricts the ISDN and the subaddress of the called user.	095/096/098
Advice of Charge - S	178/181/182
Advice of Charge - D	179/181/182

Supplementary Service/Description	ETS 300 Specification
Message Waiting Indication	650/745-1/356-20

Appendix E - Establishing ISDN Cable Connections

This appendix explains the basic principles of ordering ISDN service and establishing a connection between the Dialogic Digital Network Interface boards and the Network Termination Unit (NTU).

Ordering Service

When ordering your ISDN service from a carrier, keep the following points in mind when talking to a service representative:

- Be specific when describing the kinds of service options you want. Your carrier may offer options that the carrier representative did not mention.
- Find out as much as you can about details of the turn-up process.
- Be sure to find out which aspects of service your carrier is responsible for and which aspects are your responsibility. Carriers may offer end-to-end coverage, or responsibility for the lines may lie with several different companies. Not knowing who to contact in the case of difficulties can delay repairs and impact productivity.
- Have the manufacturer's name, equipment numbers, and equipment registration numbers for the customer-site equipment you are installing or have installed.

You may want to consider hiring a third-party telecommunications or telephone consultant to coordinate service with a carrier. Or, you might delegate parts of the service acquisition process to others. Though these options may involve additional costs, the installation process is streamlined as a result of enlisting the help of someone knowledgeable about the service-ordering procedure.

Establishing Connections to an NTU

The Network Termination Unit (NTU) is usually the first piece of equipment on the customer premise that connects to the ISDN line. Customer equipment must be cabled to the NTU. Dialogic does not supply a board to NTU cable. You must

either purchase one from your supplier or build one yourself. If you are building your own cable, it must fit the following specifications:

Characteristic	Recommendations/Requirements
Cable Type:	The recommended cable type is twisted-pair cable in which each of the two pairs is shielded and the two pairs have a common shield as well. Shielding helps prevent noise and the twisting helps prevent cross talk.
Connectors:	The cable connects to the board via an ISO8877 Modular connector on the front or rear bracket of the board. See your NTU documentation for more information.

When building your NTU-to-board cable, be sure you understand how the NTU documentation has labeled NTU pinouts for transmit and receive to local equipment.

Be sure to test your cable after you have built and installed it. The green LEDs on the rear of the Digital Network Interface board turn on when the board firmware has been downloaded and the board is receiving clocking and synchronization information from the network.

NOTE: If the pinout appears correct but you receive a red and green light, the transmit and receive may have to be switched on one end.

Appendix F - Related Publications

This appendix lists publications that provide additional information on Dialogic products and ISDN technology.

Dialogic References

- *Voice Software Reference – Features Guide for LINUX*
- *Voice Software Reference – Features Guide for Windows*
- *Voice Software Reference – Programmer's Guide for LINUX*
- *Voice Software Reference – Programmer's Guide for Windows*
- *Voice Software Reference – Standard Runtime Library for LINUX*
- *Voice Software Reference – Standard Runtime Library for Windows*
- *Digital Network Interface Reference*

In addition, refer to the appropriate *Quick Install* card for the system's hardware.

Glossary

ASCII American Standard Code for Information Interchange

ANI Automatic Number Identification. A service that identifies the phone number of the calling party.

ANI-on-Demand A feature of AT&T ISDN service whereby the user can automatically request caller ID from the network even when caller ID does not exist.

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.

Basic Rate Interface A standard digital telecommunication service, available in many countries and in most of the United States, with an ability to digitally transmit both voice and data over standard 64 kbps lines. A BRI line consists of two 64 kbps channels for a total of 128 kbps.

BC Bearer Capability

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

Bearer Capability A field in an ISDN call setup message that specifies the speed at which data can be transmitted over an ISDN line.

BRI Basic Rate Interface

Call Reference Number (CRN) A number assigned by the application to identify a call on a specific line device.

Call Waiting feature Call Waiting allows a network to make an outgoing call while no channel is available for this call, and allows a terminal to

be notified of an incoming call (as per the basic call establishment process) with an indication that no information channel is available.

CEPT Conference des Administrations Europeenes des Postes et Telecommunications. A collection of groups that set European telecommunication standards.

CES Connection Endpoint Suffix

CRN Call Reference Number

CRV Call Reference Value

D channel Signaling channel used for transmitting signaling information across ISDN networks. This information is used to control transmission of data on associated B channels.

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.

DIALOG/HD Voice and/or telephone network interface resource boards that communicate via the SCbus. These boards include D/160SC-LS, D/240SC, D/240SC-T1, and D/300SC-E1. (Also referred to as **SpanCards**.)

DNIS Dialed Number Identification Service. A feature of 800 lines that allows a system with multiple 800 lines in its queue to access the 800 number the caller dialed. Also provides caller party number information.

DPNSS Digital Private Network Signaling System. An E-1 primary rate protocol used in Europe to pass calls transparently between PBXs.

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

drop-and-insert configuration A configuration in which two network interface resources are connected via an internal bus, such as the SCbus, to connect calls from one network interface to another. A call from one network interface can be "dropped" to a resource, such as a voice resource, for processing. In return, the resource can "insert" signaling and audio and retransmit this new bit stream via the internal bus to

connect the call to a different channel. Drop-and-insert configurations provide the ability to access an operator or another call.

DSL Digital Subscriber Loop

E-1 Another name given to the CEPT digital telephony format devised by the CCITT that carries data at the rate of 2.048 Mbps (DS-1 level).

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.

glare When an inbound call arrives while an outbound call is in the process of being setup, a "glare" condition occurs. Unless the protocol specifies otherwise, the incoming call takes precedence over the outbound call.

IA5 International Alphabet No. 5 (defined by CCITT)

IE Information Element

Information Element (IE) Used by the ISDN (Integrated Services Digital Network) protocol to transfer information. Each IE transfers information in a standard format defined by CCITT standard Q.931.

Integrated Services Digital Network (ISDN) A collection of standards for defining interfaces and operation of digital switching equipment for voice and data transmissions.

ISDN Integrated Services Digital Network

LAPB Link Access Protocol-Balanced

LAPD Link Access Protocol on the D channel

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

Non-Call Associated Signaling (NCAS) allows users to communicate by user-to-user signaling without setting up a circuit-switched connection (this signal does not occupy B channel bandwidth). A temporary signaling connection is established and cleared in a manner similar to the control of a circuit-switch connection. Since NCAS calls are not associated with any B channel, applications receive and transmit NCAS calls on the D channel line device. Once the NCAS connection is

established, the application can transmit user-to-user messages using the CRN associated with the NCAS call. An ISDN feature that supports the 5ESS protocol.

Network Facility Associated Signal (NFAS) Allows multiple spans to be controlled by a single D channel subaddressing.

NULL A call state in which no call is assigned to the device (line or time slot).

PRI Primary Rate Interface

Primary Rate Interface A standard digital telecommunication service, available in many countries and most of the United States, that allows the transfer of voice and data over T-1 or E-1 lines. The T-1 ISDN Primary Rate protocol consists of 23 voice/data channels (B-channels) and one signaling channel (D-channel). The E-1 ISDN Primary Rate protocol consists of 30 voice/data channels, one signaling channel (D-channel), and one framing channel to handle synchronization.

PSTN Public Switched Telephone Network

Public Switched Telephone Network An abbreviation used by the CCITT. Refers to the worldwide telephone network accessible to all those with either a telephone or access privileges.

SAPI Service Access Point Identifier

SCbus The TDM (Time Division Multiplexed) bus connecting SCSA (Signal Computing System Architecture) voice, telephone network interface, and other technology resource boards together.

SIT Special Information Tone

SpanCard See DIALOG/HD.

Special Information Tone Detection of an SIT sequence indicates an operator intercept or other problem in completing the call.

SPID Service Profile Interface ID

SRL Standard Runtime Library

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

TEI Terminal Endpoint Identifier (see Recommendations Q.920 and Q.921)

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

T-1 A digital line transmitting at 1.544 Mbps over 2 pairs of twisted wires. Designed to handle a minimum of 24 voice conversations or channels, each conversation digitized at 64 Kbps. T-1 is a digital transmission standard in North America.

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. A multithreaded process has more than one thread that are executed seemingly simultaneously. When the last thread finishes its task, then the process terminates. The main thread is also referred to as a primary thread; both main and primary thread refer to the first thread started in a process. A thread of execution is just a synonym for thread.

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.

Two B Channel Transfer (TBCT) Connects two independent B Channel calls at an ISDN PRI user's interface to each other at the PBX or CO. The ISDN PRI user sends a Facility message to the PBX or CO requesting that the two B Channel calls be connected. If accepted, the user is released from the calls.

USID User Service Identifier

UUI User-to-User Information. Proprietary messages sent to remote system during call establishment.

Vari-A-Bill Service bureaus can vary the billing rate of a 900 call at any time during the call. Callers select services from a voice-automated menu and each service can be individually priced.

voice channel Designates a bi-directional transfer of data for a single call between a voice device processing that call and the SCbus. Digitized voice from the analog or T-1/E-1 interface device is transmitted over the SCbus to the voice receive (listen) channel for processing by the voice device. The voice device sends the response to the call over the voice transmit channel to an SCbus time slot that transmits this response to the analog or T-1/E-1 interface device.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W