Dialogic® NaturalAccess™ CX Device Interface API Developer's Manual

October 2009 64-0499-01

www.dialogic.com

Copyright and legal notices

Copyright © 2000-2009 Dialogic Corporation. All Rights Reserved. You may not reproduce this document in whole or in part without permission in writing from Dialogic Corporation at the address provided below.

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Corporation at the address indicated below or on the web at www.dialogic.com.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Any use case(s) shown and/or described herein represent one or more examples of the various ways, scenarios or environments in which Dialogic® products can be used. Such use case(s) are non-limiting and do not represent recommendations of Dialogic as to whether or how to use Dialogic products.

Dialogic, Dialogic Pro, Brooktrout, Diva, Cantata, SnowShore, Eicon, Eicon Networks, NMS Communications, NMS (stylized), Eiconcard, SIPcontrol, Diva ISDN, TruFax, Exnet, EXS, SwitchKit, N20, Making Innovation Thrive, Connecting to Growth, Video is the New Voice, Fusion, Vision, PacketMedia, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Corporation or its subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. The names of actual companies and product mentioned herein are the trademarks of their respective owners.

This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Revision history

Revision	Release date	Notes
9000-6916-10	February 2000	NBS, CX 2000C 9000-6916-10 Beta
9000-6916-11	May 2000	NBS, PSF 4.0
9000-6916-12	September 2000	NBS, CX 2000 and CX 2000C 1.0
9000-6916-13	March 2001	NBS, Natural Access 2000-2
9000-6916-14	August 2001	NBS, Natural Access 2001-1
9000-6916-15	November 2001	NBS, Natural Access 2002-1 Beta
9000-6916-16	May 2002	CYF, Natural Access 2002-1
9000-6916-17	April 2003	SRG, Natural Access 2003-1
9000-6916-18	April 2004	MCM, Natural Access 2004-1
64-0499-01	October 2009	LBG, NaturalAccess R9.0
Last modified: September 3, 2009		

Refer to the www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

Table Of Contents

Chapter 1: Introduction 7

Chapter 2: Terminology 9

Chapter 3: Overview of the CDI service 11

CDI service definition 11
CDI service call control 11
CDI service board management and monitoring 11
CDI service and Natural Access 11
Natural Access services 12
Contexts and event queues 12
Natural Access programming model 13
Managing parameters in Natural Access 13
Setting up the Natural Access environment 14
Initializing Natural Access 14
Creating event queues and contexts 14
Opening services 15
Receiving CDI service events 15
Linking with the CDI service 16

Chapter 4: Performing call control 17

CX 2000 modes 17
Enabling station power 17
Monitoring signaling bits 18
Starting signal detection 18
Stopping signal detection 18
Ringing 18
Starting ringing 19
Stopping ringing 19
Coordinating caller ID FSK with ring 20
Detecting DTMF tones 20
Starting DTMF detection 20
Stopping DTMF detection 20
Transmitting tones 20
Starting tone transmission 21
Stopping tone transmission 21
Setting on-board timers 21
Sending and receiving signaling patterns 22
Querying signaling bits 22
Asserting a signaling pattern at a station 22

Chapter 5: Managing boards 23

Querying the status of a board 23
Monitoring status changes 23
Monitoring board temperature 23
Controlling the application LED 24

Chapter 6: Function reference 25

Function summary 25
Using the function reference 26

cdiAssertSignal 27
cdiDisableBattery 28
cdiEnableBattery 29
cdiGetBoardStatus 30
cdiGetSignalState 32
cdiReadThermometer 33
cdiStartDTMFDetector 34
cdiStartRing 36
cdiStartSignalDetector 38
cdiStartStatusMonitoring 40
cdiStartTimer 41
cdiStartTones 43
cdiStopDTMFDetector 45
cdiStopRing 46
cdiStopSignalDetector 47
cdiStopStatusMonitoring 48
cdiStopTimer 49
cdiStopTones 50
cdiUserLed 51

Chapter 7: Demonstration programs 53
Summary of the demonstration programs 53
Preparing to run the demonstration programs 53
Building the demonstration programs 54
Setting up the required configuration files 54
Booting the system 58
CX 2000 call center demonstration program: cdicc 59
PBX demonstration program: cdipbx 63

Chapter 8: Errors and events 67
CDI service errors 67
Errors ordered alphabetically 67
Errors ordered numerically 68
CDI service events 69
Events ordered alphabetically 69
Events ordered numerically 70

Chapter 9: Parameters 71
CDI service parameters 71
CDI_DTMFDETECTOR_PARMS 72
CDI_GLOBAL_PARMS 72
CDI_RING_PARMS 73
CDI_SIGNALDETECTOR_PARMS 73
CDI_TONE_PARMS 74

1 Introduction

The *Dialogic® NaturalAccess™ CX Device Interface API Developer's Manual* explains how to develop an application using the NaturalAccess CX Device Interface API (CDI API) in the NaturalAccess environment.

This manual defines telephony terms where applicable, but assumes that you are familiar with telephony concepts. It also assumes that you are familiar with the C programming language.

Read the *Dialogic® NaturalAccess™ Software Developer’s Manual* before using this manual. The *Dialogic® NaturalAccess™ Software Developer’s Manual* contains detailed information about NaturalAccess concepts, architecture, and application development. This information must be fully understood before you use the CDI API with NaturalAccess.

2 Terminology

Note: The product to which this document pertains is part of the NMS Communications Platforms business that was sold by NMS Communications Corporation ("NMS") to Dialogic Corporation ("Dialogic") on December 8, 2008. Accordingly, certain terminology relating to the product has been changed. Below is a table indicating both terminology that was formerly associated with the product, as well as the new terminology by which the product is now known. This document is being published during a transition period; therefore, it may be that some of the former terminology will appear within the document, in which case the former terminology should be equated to the new terminology, and vice versa.

Former terminology	Dialogic terminology
CG 6060 Board	Dialogic® CG 6060 PCI Media Board
CG 6060C Board	Dialogic® CG 6060C CompactPCI Media Board
CG 6565 Board	Dialogic® CG 6565 PCI Media Board
CG 6565C Board	Dialogic® CG 6565C CompactPCI Media Board
CG 6565e Board	Dialogic® CG 6565E PCI Express Media Board
CX 2000 Board	Dialogic® CX 2000 PCI Station Interface Board
CX 2000C Board	Dialogic® CX 2000C CompactPCI Station Interface Board
AG 2000 Board	Dialogic® AG 2000 PCI Media Board
AG 2000C Board	Dialogic® AG 2000C CompactPCI Media Board
AG 2000-BRI Board	Dialogic® AG 2000-BRI Media Board
NMS OAM Service	Dialogic® NaturalAccess™ OAM API
NMS OAM System	Dialogic® NaturalAccess™ OAM System
NMS SNMP	Dialogic® NaturalAccess™ SNMP API
Natural Access	Dialogic® NaturalAccess™ Software
Natural Access Service	Dialogic® NaturalAccess™ Service
Fusion	Dialogic® NaturalAccess™ Fusion™ VoIP API
ADI Service	Dialogic® NaturalAccess™ Alliance Device Interface API
CDI Service	Dialogic® NaturalAccess™ CX Device Interface API
Digital Trunk Monitor Service	Dialogic® NaturalAccess™ Digital Trunk Monitoring API
MSPP Service	Dialogic® NaturalAccess™ Media Stream Protocol Processing API
Natural Call Control Service	Dialogic® NaturalAccess™ NaturalCallControl™ API
NMS GR303 and V5 Libraries	Dialogic® NaturalAccess™ GR303 and V5 Libraries

Former terminology	Dialogic terminology
Point-to-Point Switching Service	Dialogic® NaturalAccess™ Point-to-Point Switching API
Switching Service	Dialogic® NaturalAccess™ Switching Interface API
Voice Message Service	Dialogic® NaturalAccess™ Voice Control Element API
NMS CAS for Natural Call Control	Dialogic® NaturalAccess™ CAS API
NMS ISDN	Dialogic® NaturalAccess™ ISDN API
NMS ISDN for Natural Call Control	Dialogic® NaturalAccess™ ISDN API
NMS ISDN Messaging API	Dialogic® NaturalAccess™ ISDN Messaging API
NMS ISDN Supplementary Services	Dialogic® NaturalAccess™ ISDN API Supplementary Services
NMS ISDN Management API	Dialogic® NaturalAccess™ ISDN Management API
NaturalConference Service	Dialogic® NaturalAccess™ NaturalConference™ API
NaturalFax	Dialogic® NaturalAccess™ NaturalFax™ API
SAI Service	Dialogic® NaturalAccess™ Universal Speech Access API
NMS SIP for Natural Call Control	Dialogic® NaturalAccess™ SIP API
NMS RJ-45 interface	Dialogic® MD1 RJ-45 interface
NMS RJ-21 interface	Dialogic® MD1 RJ-21 interface
NMS Mini RJ-21 interface	Dialogic® MD1 Mini RJ-21 interface
NMS Mini RJ-21 to NMS RJ-21 cable	Dialogic® MD1 Mini RJ-21 to MD1 RJ-21 cable
NMS RJ-45 to two 75 ohm BNC splitter cable	Dialogic® MD1 RJ-45 to two 75 ohm BNC splitter cable
NMS signal entry panel	Dialogic® Signal Entry Panel

3 Overview of the CDI service

CDI service definition

The CDI service enables you to perform station call control and board management functions on the CX 2000 and CX 2000C station interface boards.

The CDI service supports only the CX 2000 board family. For information on the CX 2000 board, refer to the *CX 2000 Installation and Developer's Manual*. For information on the CX 2000C board, refer to the *CX 2000C Installation and Developer's Manual*.

CDI service call control

Using the CDI service, an application can perform basic call control operations on CX 2000 and CX 2000C boards. These operations include:

- Detecting DTMF tones
- Transmitting tones
- Setting on-board timers
- Sending and receiving signaling patterns

CDI service board management and monitoring

The CDI service provides the following board management and monitoring functions:

- Temperature monitoring
- Telco power detection
- Station module detection

CDI service and Natural Access

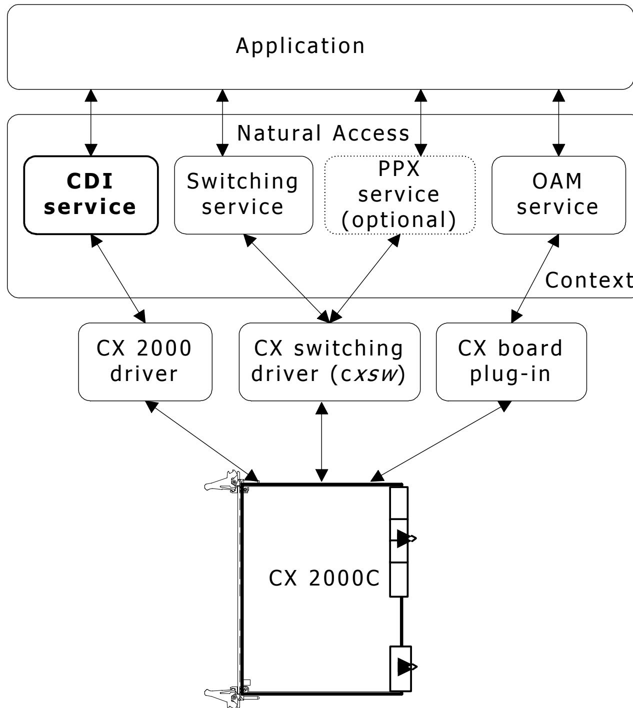
This topic describes:

- Natural Access services
- Contexts and event queues
- Natural Access programming model
- Managing parameters in Natural Access

Natural Access is a development environment that provides standard programming interfaces for hardware independent telephony functions. You must have Natural Access installed on your system to build applications using the CDI service. This topic provides background information about Natural Access and describes how the CDI service fits into the Natural Access environment. For detailed information about Natural Access, refer to the *Natural Access Developer's Reference Manual*.

Natural Access services

A Natural Access service is a group of logically related telephony functions. The CDI service is a Natural Access device service that provides station call control-related functions for CX products (refer to the following illustration). Other services address other aspects of a telephony application.



Natural Access and the CDI service

Contexts and event queues

Natural Access organizes services and accompanying resources around a single processing context. A context usually represents an application instance controlling a single telephone call.

Natural Access provides multi-processing support. Multiple Natural Access application processes can perform tasks on behalf of the same context. This is called context sharing. Natural Access applications can transfer control of contexts, such as contexts associated with individual telephone calls, to other Natural Access applications. This is called context hand-off.

An event queue is the communication path from a Natural Access service to an application. The Natural Access service generates events indicating certain conditions or state changes and sends them to applications through the event queue.

Natural Access programming model

Natural Access employs an asynchronous programming model to take advantage of concurrent processing. When called, most functions return immediately, indicating the operation was initiated. The application can then call other functions while Natural Access is processing the command.

Natural Access functions can be synchronous or asynchronous.

- Synchronous functions indicate completion by sending a return value. The return value is either SUCCESS or an error code.
- Asynchronous functions return SUCCESS if the function was successfully initiated. The execution result arrives later in an event. Asynchronous functions that return a non-zero value were never initiated; therefore, no subsequent events are generated. If an asynchronous function fails after being initiated, Natural Access delivers a DONE event to the application with an error code in the event value field.

Refer to *CDI service errors* on page 67 and *CDI service events* on page 69 for more information.

Managing parameters in Natural Access

You can alter the characteristics of Natural Access services by modifying associated parameters. Most CDI service parameter structures have default values that are sufficient for most configurations.

Natural Access manages parameters for services on a context basis. The context maintains a copy of the parameters for all services opened on the context.

Use the following Natural Access functions to obtain or change parameter information:

Function	Description
ctaGetParmByName	Retrieves a single field for a given parameter name.
ctaSetParmByName	Modifies a single field for a given parameter name.
ctaGetParmID	Retrieves the parameter ID for a given parameter descriptor name.
ctaGetParmInfo	Retrieves a parameter field definition.
ctaGetParms	Returns parameter values for a given parameter structure.
ctaRefreshParms	Resets the values of all context parameters on a context to the global defaults.

Refer to the *Natural Access Developer's Reference Manual* for information about the parameter functions.

Setting up the Natural Access environment

Before you can call functions from the CDI library, set up the Natural Access environment. Follow these steps:

- Initialize Natural Access
- Create event queues and contexts, and attach the contexts to an event queue
- Open services, including the CDI service, on each context.

Initializing Natural Access

Register services in the call to **ctaInitialize** by specifying the service and service manager names. Only the services initialized with **ctaInitialize** can be opened by the application. Service managers are dynamic link libraries (DLLs) in Windows.

Use CDIMGR, the CX board family service manager, in the call to **ctaInitialize**. Refer to the *Natural Access Developer's Reference Manual* for more information about using **ctaInitialize.**

Creating event queues and contexts

After initializing Natural Access, create the event queues and contexts. Create one or more event queues by calling **ctaCreateQueue** and specifying the service manager to attach to each queue. The CDI service manager is CDIMGR. When you attach or bind a service manager to an event queue, you make that service manager available to the event queue.

To create a context, call **ctaCreateContext** and provide the queue handle (***ctaqueuehd***) returned from **ctaCreateQueue**. All events for services on the context are received in the specified event queue.

ctaCreateContext returns a context handle (***ctahd***). The context handle is supplied by the application when invoking CDI service functions. Events communicated back to the application are also associated with the context.

Refer to the *Natural Access Developer's Reference Manual* for details on the programming models created by the use of contexts and event queues.

Opening services

To open services on a context, call **ctaOpenServices**. When opening a service on a context, you must specify information about the service and the service manager, and which resources to attach to the context.

The following table shows how to fill in this information to start CDI on a context:

Parameter	CDI service setting
CTA_SERVICE_NAME	svcname: set to CDI. svcmgrname: set to CDIMGR.
CTA_SERVICE_ADDR	Reserved for client/server usage. Initialize to 0.
CTA_SERVICE_ARGS	Not used by CDI.
CTA_MVIP_ADDR	board: The logical board number. bus: Set to 0. stream: Set to 0 for MVIP streams for switching and board management. Set to 4 for DSP streams for call control. timeslot: Set to 0 through 47 for station lines 1 through 48. The number of timeslots depends upon the specific board configuration. mode: Set to 0.

Receiving CDI service events

When the CDI service manager is attached to an event queue, it opens the board driver and associates the muxable wait object returned by the driver open command with the event queue. When this wait object is signaled on receipt of events from the board, **ctaWaitEvent** processes the events through the CDI service and passes any event generated back to the calling function.

Events arrive in the form of the standard CTA event data structure (defined in *ctadef.h*):

```
typedef struct CTA_EVENT
{
    DWORD    id;                /* Event code (and source service ID)    */
    CTAHD    ctahd;             /* Natural Access context handle         */
    DWORD    timestamp;         /* Timestamp                             */
    DWORD    userid;            /* User ID (defined by ctaCreateContext) */
    DWORD    size;              /* Size of buffer if buffer != NULL      */
    void     *buffer;           /* Buffer pointer                        */
    DWORD    value;             /* Event status or event-specific data   */
    DWORD    objHd;             /* Service client side object handle     */
} CTA_EVENT;
```

This structure informs the application which event occurred on which context and includes additional information specific to the event.

The CTA_EVENT structure contains the following fields:

Field	Description
id	Event code defined in the library header file. The event's prefix relates the event to a specific NMS Communications library of functions. For example, all CDI events are prefixed with CDIEVN_ (CDIEVN_SOMETHING_ HAPPENED). All Natural Access events are prefixed with CTAEVN_.
ctahd	Context handle (line handle) returned from **ctaCreateContext**.
timestamp	Time since the DSP was booted, in milliseconds (modulo 65536). The resolution for CX board events is 2 milliseconds. To provide a wider range, the timer alternates between even values (0, 2, 4, ...65534) and odd values (1, 3, 5, ...65535), approximately every 65 seconds.
userid	User-supplied ID. This field is unaltered by Natural Access and facilitates asynchronous programming. Its purpose is to correlate a context with an application object/context when events occur.
size	Size (bytes) of the area pointed to by buffer. If the buffer is NULL, this field may be used to hold an event-specific value.
buffer	Pointer to data returned with the event. The field contains an application process address. The event's size field contains the actual size of the buffer.
value	Event-specific value. This field can hold a reason code or an error code.
objHd	Service client-side object handle.

Linking with the CDI service

When building a Natural Access application that uses the CDI service, compile the program with *cdidef.h* and link to the *cdiapi* library (*cdiapi.lib* for Windows, *libcdiapi.so* for UNIX).

Refer to the *Natural Access Developer's Reference Manual* for more information about service implementation.

4 Performing call control

CX 2000 modes

The CX 2000 and CX 2000C digital signal processor (DSP) software operates in enhanced mode by default. In enhanced mode, the DSP performs operations that otherwise would have to be performed by the host application (for example, stopping tones or DTMF detection upon on-hook). In enhanced mode, the DSP also performs extra contextual error-checking (for example, denying DTMF detection if on-hook or denying ring if off-hook).

Enhanced mode provides the following benefits:

- Lowers the host-DSP message traffic.
- Improves system response time.
- Maximizes the use of shared resources, if any (for example, DTMF detector resources).
- Assists debugging of user applications by providing contextual errors.
- Provides flash-hook detection.

You may want to disable enhanced mode for legacy software needs, for prototyping, or for debugging. To disable enhanced mode, clear the EnhancedMode parameter in the CDI_GLOBAL_PARMS structure.

Enhanced mode affects the behavior of the following functions:

- **cdiStartDTMFDetector**
- **cdiStartRing**
- **cdiStartSignalDetector**
- **cdiStartTones**

Enabling station power

Before a station can be used, power (battery) must be supplied to it. To supply power to a station attached to a board, invoke **cdiEnableBattery** with the context handle to the board. This function connects the DC loop power to the station and enables the audio. **cdiEnableBattery** returns SUCCESS or an error code.

To disable power to a station, invoke **cdiDisableBattery** with the context handle to the board. This function removes the DC loop power from the station and disables the audio.

Monitoring signaling bits

Once battery is enabled, the DSP can monitor signaling bits. This topic describes how to:

- Start signal detection
- Stop signal detection

Starting signal detection

To start on-board monitoring of the signaling bits for a station, invoke **cdiStartSignalDetector**. In the invocation, pass a pointer to the CDI_SIGNALDETECTOR_PARMS structure.

If monitoring is started successfully, **cdiStartSignalDetector** returns SUCCESS, and signal monitoring events are enabled. Subsequent events indicate bit state changes.

Stopping signal detection

If enhanced mode is enabled, the DSP automatically stops detecting signaling from a station when the station goes on-hook (CDIEVN_ON_HOOK is returned). If enhanced mode is disabled, the application can invoke **cdiStopSignalDetector** to stop detection.

If the station is on-hook, and **cdiStopSignalDetector** is invoked, CTAERR_FUNCTION_NOT_ACTIVE is returned.

Ringing

As long as a station is on-hook, an application can cause it to ring. This topic describes:

- Starting ringing
- Stopping ringing
- Coordinating caller ID FSK with ring

Starting ringing

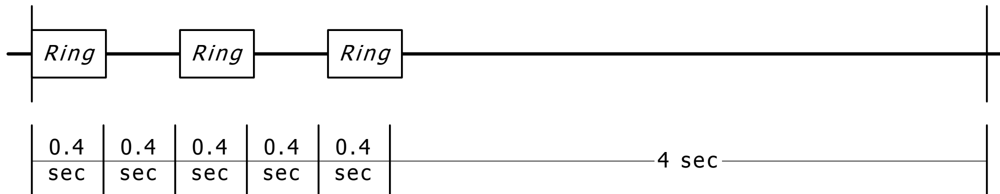
To cause a station to ring, invoke **cdiStartRing**. In the invocation, pass a pointer to the CDI_RING_PARMS structure.

CX 2000 products are preconfigured with three different 6-second cadence types:

cadence_type = 0 "ring...ring..."

cadence_type = 1 "ring ring...ring ring..."

cadence_type = 2 "ring ring ring...ring ring ring..."



Preconfigured ring cadences for CX 2000 products

You can change one or more of these cadences. For more information, refer to the *CX 2000 Installation and Developer's Manual* or the *CX 2000C Installation and Developer's Manual*.

cdiStartRing cannot be invoked for a station while that station is ringing. If this function is invoked during an ongoing ring, CTAERR_FUNCTION_ACTIVE is returned.

Stopping ringing

If enhanced mode is enabled, the DSP automatically stops station ringing when the station goes off-hook (CDIEVN_OFF_HOOK is returned). If enhanced mode is disabled, the application can invoke **cdiStopRing** to stop an ongoing ring.

If the station is off-hook and **cdiStopRing** is invoked, CTAERR_FUNCTION_NOT_ACTIVE is returned.

Coordinating caller ID FSK with ring

CX boards can be configured to generate events that track ring cycles. The primary application for this is FSK caller ID generation. An application can determine when the first ring cycle is complete and can switch through and enable FSK generation on another resource board in the system to send caller ID between the first two ring cycles. Refer to *cdiStartRing* on page 36 for more information.

Detecting DTMF tones

When a station is off-hook, the DSP can detect DTMF tones and return their values to the application. This topic describes:

- Starting DTMF detection
- Stopping DTMF detection

Starting DTMF detection

To start DTMF detection, invoke **cdiStartDTMFDetector**. In the invocation, pass a pointer to the CDI_DTMFDETECTOR_PARMS structure. If detection is started successfully, **cdiStartDTMFDetector** returns SUCCESS.

CDIEVN_DTMF_STARTED is generated for each detected tone. The event value field contains the ASCII representation of the key. When the tone ends, CDIEVN_DTMF_ENDED is generated.

If **cdiStartDTMFDetector** is invoked when the DSP does not have enough MIPS to satisfy the request, CTAERR_OUT_OF_RESOURCES is returned.

Stopping DTMF detection

If enhanced mode is enabled, the DSP automatically stops DTMF detection for a station when the station goes on-hook (CDIEVN_ON_HOOK is returned). If enhanced mode is disabled, the application can invoke **cdiStopDTMFDetector** to stop detection.

If the station is on-hook, and **cdiStopDTMFDetector** is invoked, CTAERR_FUNCTION_NOT_ACTIVE is returned.

Transmitting tones

When a station is off-hook, the CX board can transmit tones to the station. Each tone can consist of one or two frequency components, each at different amplitudes. You can also set up a pattern (cadence) of up to two iterations of the tone, each of different lengths.

This topic describes:

- Starting tone transmission
- Stopping tone transmission

Starting tone transmission

To start the transmission of a tone, invoke **cdiStartTones**. If the function invocation is successful, **cdiStartTones** returns SUCCESS. When tone transmission is complete, CDIEVN_TONE_DONE is generated.

cdiStartTones cannot be invoked for a station while the DSP is currently playing a tone at that station. If this function is invoked during an ongoing tone, CTAERR_FUNCTION_ACTIVE is returned.

Stopping tone transmission

If enhanced mode is enabled, the DSP automatically stops transmitting tones to a station when the station goes on-hook (CDIEVN_ON_HOOK is received), or a DTMF tone is detected (CDIEVN_DTMF_STARTED is received). If enhanced mode is disabled, the application can invoke **cdiStopTones** to stop transmission.

If the station is on-hook, and **cdiStopTones** is invoked, CTAERR_FUNCTION_NOT_ACTIVE is returned.

Setting on-board timers

The CDI service supports one on-board timer per context on a CX board. You can set up this timer to run for a specified amount of time and then expire, returning an event to the application. Alternatively, you can configure the timer to tick, generating a specified number of tick events at a periodic rate before expiring.

To set up the timer, invoke **cdiStartTimer**. The timer has a 2 ms resolution.

You can abort a timer by invoking **cdiStopTimer**. **cdiStopTimer** returns SUCCESS if successfully invoked. When the timer is stopped, CDIEVN_TIMER_DONE is generated. The value field of this event contains CTA_REASON_STOPPED. If you invoke **cdiStopTimer** when the timer has already stopped, CDIEVN_TIMER_STOPPED is generated.

Sending and receiving signaling patterns

This topic describes how to:

- Query the current state of the supervisory signaling bits.
- Assert a supervisory signaling pattern at a station.

These functions are not intended for use in typical applications. Ordinarily, the supervisory signaling is transparently controlled by the on-board DSP. Use these functions when special-purpose applications require explicit control of the supervisory signaling. In such cases, disable enhanced mode as described in *CX 2000 modes* on page 17.

Querying signaling bits

To retrieve the current state of the supervisory signaling bits for a station, invoke **cdiGetSignalState**, passing the context handle to the station. **cdiGetSignalState** returns a pointer to a value indicating the state of the bits:

Pattern Value	Description
00000***xxx*** (***xxx*** is any value)	The bits are low (the station is on-hook).
00001***xxx*** (***xxx*** is any value)	The bits are high (the station is off-hook).

Do not use **cdiGetSignalState** when enhanced mode is enabled.

Asserting a signaling pattern at a station

To assert a signaling pattern at a station, invoke **cdiAssertSignal**. With the invocation, pass the context handle to the station and one of the following values:

Pattern value	Description
0x00	Battery off, no ringing
0x08	Battery on, no ringing
0x0C	Battery on, continuous ringing

Do not use **cdiAssertSignal** when enhanced mode is enabled.

5 Managing boards

Querying the status of a board

An application can query the status of a CX board. The following information is available:

- Whether various voltage levels are available to power the stations, and ringing at the stations
- The number of subassemblies present
- Whether the board is operating at a normal temperature, or is overheating
- System errors

To query the status of a board, invoke **cdiGetBoardStatus**. **cdiGetBoardStatus** returns a CDI_BOARD_STATUS structure containing status information.

CDI_BOARD_STATUS is defined in *cdidef.h*. Refer to *cdiGetBoardStatus* on page 30 for a description of the fields in this structure.

Monitoring status changes

Applications can register to automatically receive events whenever the board status changes. To register an application to receive board status change events, start **cdiStartStatusMonitoring** on one context per board. While you can monitor more than one context for a board, monitoring one context is sufficient and conserves system resources. Open this context on a non-DSP port such as local:0:0.

If **cdiStartStatusMonitoring** is successfully invoked, it returns SUCCESS. Any board status change generates CDIEVN_BOARD_STATUS_CHANGE. The application can then invoke **cdiGetBoardStatus** to determine what has changed.

To unregister an application from receiving status change events, invoke **cdiStopStatusMonitoring**.

Monitoring board temperature

CX boards provide protection from overheating by monitoring board temperature and shutting down heat-producing operations if the temperature becomes too high. A board can have one of the following temperature states:

Temperature state	Description
Normal	Temperature is within normal operating range.
Warning	Temperature has reached warning range.
Protection	Temperature has reached protection range.

The board's temperature is typically within the normal range. The board stays in this range as long as adequate cooling is provided and appropriate guidelines are followed. To determine the exact temperature range represented by each state and for guidelines on providing optimum cooling for CX products, refer to either the CX 2000 or CX 2000C installation and developer's manuals.

If a board begins to overheat and the temperature reaches the warning range, CDIEVN_BOARD_STATUS_CHANGE is generated. An application receives this event only if it is registered to receive status change events as described in *Monitoring status changes* on page 23. The thermal_state field in the CDI_BOARD_STATUS structure returned by **cdiGetBoardStatus** is set to CDI_THERMAL_WARNING.

If the warning condition is not addressed, overheating may progress to the protection state. When the protection state is reached, thermal_state becomes CDI_THERMAL_PROTECTION. At this stage, the board automatically takes the following actions, preventing new incoming or outgoing calls from being processed:

- The DTMF detector becomes unavailable.
- Invocations of **cdiStartRing** return CTAERR_BOARD_ERROR. No ringing takes place.
- New calls receive a special tone in place of the dial tone. This tone has the following characteristics and cannot be changed:

Parameter	Value
freq1	1,000 Hz
freq2	2,000 Hz
ampl1	-15
ampl2	-15
ton1	100 ms
toff1	100 ms
ton2	0 ms
toff2	0 ms
iterations	-1 (forever)

In extreme situations, the hardware reduces the loop current to the stations, impacting the operation of telephones or other attached devices.

An application can query the exact board temperature at any time by invoking **cdiReadThermometer** with the number of a thermometer to check. (A board has five thermometers, 0 through 4.) **cdiReadThermometer** returns the exact current temperature measured by the thermometer in degrees Celsius. To determine a typical operating temperature, query the temperature after the board has been running under a typical load (with a number of stations off-hook) for one hour.

Controlling the application LED

CX 2000 boards have a green LED that the application can control. The application defines the purpose (generally maintenance, internal debugging, or other custom use). To control the LED, invoke **cdiUserLed**.

6 Function reference

Function summary

The following functions perform CDI service low-level call control:

Function	Synchronous/ Asynchronous	Description
cdiAssertSignal	Synchronous	Asserts a supervisory signaling pattern to the line.
cdiDisableBattery	Synchronous	Removes the power to the station attached to a port.
cdiEnableBattery	Synchronous	Supplies power to the station attached to a port.
cdiGetSignalState	Synchronous	Retrieves the current state of the supervisory signaling bits.
cdiStartDTMFDetector	Asynchronous	Enables the DTMF detector.
cdiStartRing	Synchronous	Starts station ringing.
cdiStartSignalDetector	Asynchronous	Starts monitoring the signaling bits.
cdiStartTimer	Asynchronous	Starts (or restarts) a timer on a port.
cdiStartTones	Asynchronous	Starts the transmission of a tone.
cdiStopDTMFDetector	Synchronous	Disables the DTMF detector.
cdiStopRing	Synchronous	Stops station ringing.
cdiStopSignalDetector	Synchronous	Stops monitoring the signaling bits.
cdiStopTimer	Asynchronous	Aborts the timer operation.
cdiStopTones	Synchronous	Stops the tone transmission.

The following functions manage CDI service boards:

Function	Synchronous/ Asynchronous	Description
cdiGetBoardStatus	Synchronous	Registers the application to receive board status change events.
cdiReadThermometer	Synchronous	Returns the current temperature of a board.
cdiStartStatusMonitoring	Asynchronous	Returns the current board status.
cdiStopStatusMonitoring	Synchronous	Unregisters the application from receiving board status change events.
cdiUserLed	Synchronous	Sets the user LED.

Using the function reference

A typical function description includes:

<table>
<tr><td>Prototype</td><td>The prototype is shown followed by a listing of the function's arguments. Data types include:
WORD (16-bit unsigned)
DWORD (32-bit unsigned)
INT16 (16-bit signed)
INT32 (32-bit signed)
BYTE (8-bit unsigned)
If a function argument is a data structure, the complete data structure is defined. Refer to CDI service parameters on page 71 for a description of all data structures and parameters.</td></tr>
<tr><td>Return values</td><td>The return value for a function is either SUCCESS or an error code. For asynchronous functions, a return value of SUCCESS indicates the function was initiated. Subsequent events indicate the status of the operation.

Refer to the CDI service errors on page 67 for a list of errors returned by CDI service functions.</td></tr>
<tr><td>Events</td><td>Asynchronous functions list DONE events when the function is complete. Additional information such as reason codes and return values may be provided in the value field of the event. Synchronous functions do not generate events.

Refer to the CDI service events on page 69 for information about CDI service events and reason codes.</td></tr>
<tr><td>Example</td><td>Example functions that start with Demo are excerpts taken from demonstration function libraries shipped with the product.

Example functions that start with my are excerpts taken from sample application programs shipped with the product.

The notation <code>/* ... */</code> indicates additional code that is not shown.</td></tr>
</table>

cdiAssertSignal

Asserts a supervisory signaling pattern at the station.

Prototype

DWORD **cdiAssertSignal** (CTAHD ***ctahd***, DWORD ***pattern***)

Argument	Description
ctahd	Context handle.
pattern	Bit mask to assert. Refer to the Details section for more information.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Invalid handle.
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.

Details

cdiAssertSignal asserts a signaling pattern at a station. ***pattern*** can be any of the following values:

Pattern value	Description
0x00	Battery off, no ringing.
0x08	Battery on, no ringing.
0x0C	Battery on, continuous ringing.

cdiAssertSignal is not intended for use in typical applications. The supervisory signaling is transparently controlled by the on-board DSP. Use this function when special purpose applications require explicit control of the supervisory signaling. In such cases, disable enhanced mode as described in *CX 2000 modes* on page 17.

Refer to *Sending and receiving signaling patterns* on page 22 for more information.

See also

cdiGetSignalState

cdiDisableBattery

Removes the DC loop power from a station and disables the audio. Invoke **cdiEnableBattery** to connect the DC loop power to the station and to use voice and signaling features. Refer to *Enabling station power* on page 17 for more information.

Prototype

DWORD **cdiDisableBattery** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

cdiEnableBattery

Connects the DC loop power to the station and enables the audio. Invoke **cdiEnableBattery** to use voice and signaling features. Refer to *Enabling station power* on page 17 for more information.

Prototype

DWORD **cdiEnableBattery** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

See also

cdiDisableBattery

cdiGetBoardStatus

Queries the current board status.

Prototype

DWORD **cdiGetBoardStatus** (CTAHD ***ctahd***, CDI_BOARD_STATUS ****board_status***)

Argument	Description
ctahd	Context handle.

board_status	Pointer to the CDI_BOARD_STATUS structure containing status information:
	`typedef struct`
	`{`
	`     DWORD size;`
	`     DWORD voltage_presence;`
	`     DWORD board_presence;`
	`     DWORD thermometer_presence;`
	`     DWORD thermal_state;`
	`     DWORD hmic_error;`
	`     DWORD slic_error[3];`
	`} CDI_BOARD_STATUS;`
	Refer to the Details section for field descriptions.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

CDI_BOARD_STATUS is defined in *cdidef.h*. The following table describes the values for the fields in this structure:

Field	Description
voltage_presence	Valid values: CDI_VOLTAGE_LOW_BATTERY (0x0001) = Low battery voltage detected. CDI_VOLTAGE_HIGH_BATTERY (0x0002) = High battery voltage detected. CDI_VOLTAGE_RING_VOLTAGE (0x0004) = Ring voltage detected. CDI_VOLTAGE_EXT_RINGER (0x0008) = External ringer enabled.
board_presence	Valid values: CDI_BOARD_FIRST_STATION_GROUP (0x000) = Station group on base board present. CDI_BOARD_SECOND_STATION_GROUP (0x0002) = Station group on first station interface module present. CDI_BOARD_THIRD_STATION_GROUP (0x0004) = Station group on second station interface module present. CDI_BOARD_REAR_BOARD (0x0008) = CX 2000C rear transition board present.
thermometer_presence	Corresponds to each sensor. Valid values: CDI_THERMOMETER_0_POPULATED (0x0001) = Thermometer 0 enabled. CDI_THERMOMETER_1_POPULATED (0x0002) = Thermometer 1 enabled. CDI_THERMOMETER_2_POPULATED (0x0004) = Thermometer 2 enabled. CDI_THERMOMETER_3_POPULATED (0x0008) = Thermometer 3 enabled. CDI_THERMOMETER_4_POPULATED (0x0010) = Thermometer 4 enabled.
thermal_state	Valid values: CDI_THERMAL_NORMAL (0x0001) = Temperature within normal operating range. CDI_THERMAL_WARNING (0x0002) = Temperature reached warning range. CDI_THERMAL_PROTECTION (0x0003) = Temperature reached protection range. **Note:** The thermal_state field values cannot be combined.
hmic_error	Valid values: No error CDI_HMIC_ERROR_SYSTEM (0x0001) = Reserved. CDI_HMIC_ERROR_CLOCK (0x0002) = Reserved. CDI_BSP_SYNC_ERROR (0x0004) = Clock error detected and resynchronization done.
slic_error	Array representing the error status for each port group (1..3). The lowest 16 bits in each slic_error element represent the error status of the corresponding 16 individual ports.

Refer to *Monitoring status changes* on page 23 for more information.

See also

cdiStartStatusMonitoring, **cdiStopStatusMonitoring**

cdiGetSignalState

Queries the current state of the supervisory signaling bits.

Prototype

DWORD **cdiGetSignalState** (CTAHD ***ctahd***, DWORD ***pattern***)

Argument	Description
ctahd	Context handle.
pattern	Pointer to the current pattern sent to the line. Refer to the Details section for more information.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

pattern returned by **cdiGetSignalState** points to a value indicating the state of the supervisory signaling bits for a station. Possible values are:

Pattern value	Description
00000***xxx*** (***xxx*** is any value)	The bits are low (the station is on-hook).
00001***xxx*** (***xxx*** is any value)	The bits are high (the station is off-hook).

cdiGetSignalState is not intended for use in typical applications. The supervisory signaling is transparently controlled by the on-board DSP. Use this function when special purpose applications require explicit control of the supervisory signaling. In such cases, disable enhanced mode as described in *CX 2000 modes* on page 17.

Refer to *Sending and receiving signaling patterns* on page 22 for more information.

See also

cdiAssertSignal

cdiReadThermometer

Queries the current temperature of a board.

Prototype

DWORD **cdiReadThermometer** (CTAHD ***ctahd***, DWORD ***tid***, DWORD ****temp***)

Argument	Description
ctahd	Context handle.
tid	Thermometer ID (0,1,2,3,4).
temp	Pointer to the current temperature in degrees Celsius.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_BOARD_ERROR	Sensor is not installed or is not working.
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

An application can query the exact board temperature at any time by invoking **cdiReadThermometer** with the number of a thermometer to check. (A board has five thermometers, 0 through 4.) The function returns the exact current temperature measured by the thermometer in degrees Celsius. Once the system is powered on, the system temperature takes approximately one hour to stabilize.

Refer to *Monitoring board temperature* on page 23 for more information.

cdiStartDTMFDetector

Enables the DTMF detector.

Prototype

DWORD **cdiStartDTMFDetector** (CTAHD ***ctahd***, CDI_DTMFDETECTOR_PARMS ****parms***)

Argument	Description
ctahd	Context handle.

parms	Pointer to a structure containing DTMF detection parameters. Specify NULL to use the default values.
	`typedef struct`
	`{`
	`   DWORD     size;`
	`   DWORD     onqualtime;`
	`   DWORD     offqualtime;`
	`   DWORD     DTMF_mode;`
	`} CDI_DTMFDETECTOR_PARMS`
	Refer to *CDI_DTMFDETECTOR_PARMS* on page 72 for field descriptions.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_BOARD_ERROR	Board is overheating, and has reached the protection stage. At this stage, the board does not accept or place calls on any ports. Refer to *Monitoring board temperature* on page 23 for details.
CTAERR_INVALID_STATE	Port must be off-hook to invoke this function.
CTAERR_FUNCTION_ACTIVE	Function has already been invoked, and the DTMF detector is active.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_OUT_OF_RESOURCES	DSP does not have enough MIPS to satisfy the request. Retry later.

Events

Event	Description
CDIEVN_DTMF_STARTED	DTMF tone is detected. The event value field contains the ASCII representation of the key. The DTMF_mode parameter in CDI_DTMFDETECTOR_PARMS determines when this event is reported.
CDIEVN_DTMF_ENDED	DTMF tone currently detected just ended.

Details

When a station is off-hook, the DSP can detect DTMF tones and return their values to the application. To start DTMF detection, invoke **cdiStartDTMFDetector**. In the invocation, pass a pointer to the CDI_DTMFDETECTOR_PARMS structure.

If the board is running in enhanced mode, the DSP automatically stops DTMF detection for a station when the station is placed on-hook (CDIEVN_ON_HOOK is returned). Otherwise, invoke **cdiStopDTMFDetector** to stop DTMF detection.

Refer to *Detecting DTMF tones* on page 20 for more information.

cdiStartRing

Starts station ringing.

Prototype

DWORD **cdiStartRing** (CTAHD ***ctahd***, CDI_RING_PARMS ****parms***)

Argument	Description
ctahd	Context handle.

parms	Pointer to a structure specifying ring cadence:
	`typedef struct`
	`{`
	`   DWORD     size;`
	`   DWORD     cadence_type;`
	`} CDI_RING_PARMS`
	Refer to the Details section for more information.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_BOARD_ERROR	Board is overheating, and has reached the protection state. At this stage, the board does not accept or place calls on any ports. Refer to *Monitoring board temperature* on page 23 for more information.
CTAERR_FUNCTION_ACTIVE	Function was already invoked, and the DSP is ringing the station.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_STATE	Port must be on-hook to invoke this function.

Events

Event	Description
CDIEVN_RING_IS_OFF	Ring cycle finished. FSK transmission is allowed.
CDIEVN_RING_IS_ON	Ring cycle started. FSK transmission is not allowed until CDIEVN_RING_IS_OFF is received.

Details

cdiStartRing causes an on-hook station to ring. The cadence_type parameter in CDI_RING_PARMS determines the pattern (cadence) of the ring. CX 2000 products are preconfigured with three different 6-second cadences to choose from:

cadence_type setting	Cadence
0	"ring...ring..."
1	"ring ring...ring ring..."
2	"ring ring ring...ring ring ring..."

You can change one or more of these cadences. For more information, refer to the *CX 2000 Installation and Developer's Manual* or the *CX 2000C Installation and Developer's Manual*.

If the board is running in enhanced mode, the DSP automatically stops ringing when the station is taken off-hook. Otherwise, invoke **cdiStopRing** to stop ringing a station.

Coordinating caller ID FSK with ring

CX boards can be configured to generate events that track ring cycles. The primary application for this is FSK caller ID generation. An application can determine when the first ring cycle is complete and can switch through and enable FSK generation on another resource board in the system to send caller ID between the first two ring cycles.

To enable the ring cycle events after **cdiStartRing** is invoked, use the OR operator to include RING_CYCLE_EVENTS_MASK in the EnhancedMode variable of the CDI_GLOBAL_PARMS structure. These events track ring cycles, not the individual ring bursts comprised in the cadence of the ring cycle.

Note: Depending upon the exact timing of when **cdiStopRing** is called, a trailing ring event may occur after calling **cdiStopRing**.

Refer to *Ringing* on page 18 for more information.

See also

cdiStopRing

cdiStartSignalDetector

Starts on-board monitoring of the signaling bits. You must issue **cdiEnableBattery** before you start **cdiStartSignalDetector**.

Prototype

DWORD **cdiStartSignalDetector** (CTAHD ***ctahd***, CDI_SIGNALDETECTOR_PARMS ***parms***)

<table>
<tr><th>Argument</th><th>Description</th></tr>
<tr><td>ctahd</td><td>Context handle.</td></tr>
<tr><td>parms</td><td>Pointer to a structure specifying signal detection parameters:<pre>
typedef struct
{
   DWORD   size;
   DWORD   timeon;
   DWORD   timeoff;
   DWORD   min_flash_hook;
   DWORD   max_flash_hook;
} CDI_SIGNALDETECTOR_PARMS
</pre>Refer to CDI_SIGNALDETECTOR_PARMS on page 73 for field descriptions.</td></tr>
</table>

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_INVALID_CTAHD	Handle is invalid.

Events

Event	Description
CDIEVN_OFF_HOOK	Station picked up.
CDIEVN_ON_HOOK	Station hung up.
CDIEVN_FLASH_HOOK	Flash-hook generated.
CDIEVN_A_CHANGE_ON	A bit was just set.
CDIEVN_A_CHANGE_OFF	A bit was just reset.

Details

Once station power is enabled, the DSP can monitor signaling bits. Use **cdiStartSignalDetector** to start on-board monitoring of the signaling bits.

When operating in enhanced mode, the DSP qualifies bit changes according to the parameters in the CDI_SIGNALDETECTOR_PARMS structure, as described in the following table:

Bit state change	Event presented to application
Qualified on-hook to off-hook (more than the timeon field value).	CDIEVN_OFF_HOOK
All off-hook to on-hook transitions less than min_flash_hook are ignored.	No events
Qualified off-hook to on-hook (more than min_flash_hook field value and remains on for less than the max_flash_hook field value).	CDIEVN_FLASH_HOOK
Qualified off-hook to on-hook (more than the max_flash_hook field value).	CDIEVN_ON_HOOK

When operating with enhanced mode disabled, only low-level supervisory signaling events CDIEVN_A_CHANGE_ON and CDIEVN_A_CHANGE_OFF are presented. Flash-hook detection is disabled.

Refer to *Monitoring signaling bits* on page 18 for more information about signal detection.

See also

cdiStopSignalDetector

cdiStartStatusMonitoring

Registers the caller to receive board status change events.

Prototype

DWORD **cdiStartStatusMonitoring** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

Events

Event	Description
CDIEVN_BOARD_STATUS_CHANGE	The board status changed. Invoke **cdiGetBoardStatus** for details.

Details

Applications can register to receive events whenever the board status changes. To register an application, invoke **cdiStartStatusMonitoring** on a non-DSP port such as local:0:0. While you can monitor more than one context for a board, monitoring one context is sufficient and conserves system resources.

Any board status change generates CDIEVN_BOARD_STATUS_CHANGE. The application can then invoke **cdiGetBoardStatus** to determine what changed.

To unregister an application, invoke **cdiStopStatusMonitoring**.

Refer to *Monitoring status changes* on page 23 for more information.

cdiStartTimer

Starts (or restarts) a timer on this port.

Prototype

DWORD **cdiStartTimer** (CTAHD ***ctahd***, DWORD ***timeout***, DWORD ***count***)

Argument	Description
ctahd	Context handle.
timeout	Time out value in milliseconds.
count	Number of events.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_INVALID_CTAHD	Handle is invalid.

Events

Event	Description
CDIEVN_TIMER_DONE	The timer stopped. The event value field contains one of the following reason codes: CTA_REASON_TIMEOUT The timer ran for the full specified time, and then expired. CTA_REASON_STOPPED The timer was halted with **cdiStopTimer**.
CDIEVN_TIMER_START	The timer started.
CDIEVN_TIMER_TICK	The timer ticked. Refer to the Details section for more information.

Details

The CDI service supports one on-board timer per context on a CX board. You can set up this timer to run for a specified amount of time and then expire, returning an event to the application. Alternatively, you can configure the timer to tick, generating a specified number of tick events at a periodic rate before expiring:

Timer task	Steps
Run once and then expire	• Set ***timeout*** to timeout length (in ms). • Set ***count*** to 1. When the timer is started, CDIEVN_TIMER_START is generated. When the timer expires, CDIEVN_TIMER_DONE is returned. The value field of this event contains CTA_REASON_FINISHED.
Tick	• Set ***timeout*** to tick rate (in ms). • Set ***count*** to number of ticks required. When the timer is started, CDIEVN_TIMER_START is generated. As the timer ticks, CDIEVN_TIMER_TICK events are generated. On the final tick, CDIEVN_TIMER_DONE is returned. The value field of this event contains CTA_REASON_FINISHED.

The timer has a 2 ms resolution.

You can abort a timer by invoking **cdiStopTimer**.

A running timer can be restarted at any time before normal expiration by a subsequent call to **cdiStartTimer**.

cdiStartTones

Starts the transmission of a single or double-cadenced dual tone.

Prototype

DWORD **cdiStartTones** (CTAHD ***ctahd***, CDI_TONE_PARMS ***parms***)

Argument	Description
ctahd	Context handle.

parms	Pointer to a structure specifying the frequency, cadence, and iterations of the tone:
	`typedef struct`
	`{`
	`   DWORD     size;`
	`   DWORD     freq1;`
	`   DWORD     freq2;`
	`   INT32     ampl1;`
	`   INT32     ampl2;`
	`   DWORD     ontime;`
	`   DWORD     offtime;`
	`   DWORD     ontime2;`
	`   DWORD     offtime2;`
	`   INT32     iterations;`
	`} CDI_TONE_PARMS`
	Refer to CDI_TONE_PARMS for a description of these fields.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_FUNCTION_ACTIVE	Function was already invoked.
CTAERR_INVALID_CTAHD	Handle is invalid.
CTAERR_INVALID_STATE	Port must be off-hook to invoke this function.

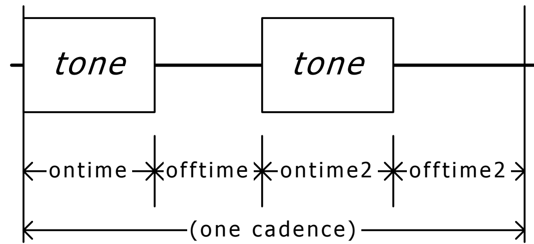
Events

Event	Description
CDIEVN_TONE_DONE	Tone transmission is complete.

Details

When a station is off-hook, you can transmit tones to the station. Each tone can consist of one or two frequency components, each at different amplitudes. You can also set up a pattern (cadence) of up to two iterations of the tone, each of different lengths.

To start the transmission of a tone, invoke **cdiStartTones**. In the invocation, pass a pointer to the CDI_TONE_PARMS structure. The following illustration shows how the parameters in CDI_TONE_PARMS determine the cadence:



Cadence parameters

When enhanced mode is enabled, the DSP automatically stops tones upon detection of on-hook or DTMF. Otherwise, invoke **cdiStopTones** to stop an ongoing tone.

Note: If the board enters the protection state (due to overheating), all stations receive an alarm tone indicating maintenance attention is needed. This tone overrides any other tones.

Refer to *Transmitting tones* on page 20 for more information.

cdiStopDTMFDetector

Disables the DTMF detector.

Prototype

DWORD **cdiStopDTMFDetector** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	DTMF detector was already stopped, either by a previous invocation of this function or automatically by the DSP.
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

If a board is not running in enhanced mode, use **cdiStopDTMFDetector** to stop DTMF detection (started using **cdiStartDTMFDetector**)**.**

If the board is running in enhanced mode, the DSP automatically stops DTMF detection for a station when the station is placed on-hook (CDIEVN_ON_HOOK is returned). In this case, **cdiStopDTMFDetector** returns CTAERR_FUNCTION_NOT_ACTIVE.

Refer to *Detecting DTMF tones* on page 20 for more information.

cdiStopRing

Stops station ringing.

Prototype

DWORD **cdiStopRing** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Ringing was already stopped, either by a previous invocation of this function or automatically by the DSP.
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

If a board is not running in enhanced mode, use **cdiStopRing** to stop station ringing (started using **cdiStartRing**).

If the board is running in enhanced mode, the DSP automatically stops station ringing when the station is taken off-hook (CDIEVN_OFF_HOOK is returned). In this case, **cdiStopRing** returns CTAERR_FUNCTION_NOT_ACTIVE.

Refer to *Ringing* on page 18 for more information.

cdiStopSignalDetector

Stops monitoring the signaling bits. Refer to *Monitoring signaling bits* on page 18 for more information about signal detection.

Prototype

DWORD **cdiStopSignalDetector** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

See also

cdiStartSignalDetector

cdiStopStatusMonitoring

Unregisters the application from receiving board status change events. The application is no longer notified if the board status changes. Refer to *Monitoring status changes* on page 23 for more information.

Prototype

DWORD **cdiStopStatusMonitoring** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

See also

cdiStartStatusMonitoring

cdiStopTimer

Aborts an ongoing timer (started using **cdiStartTimer**). Refer to *Setting on-board timers* on page 21 for more information.

Prototype

DWORD **cdiStopTimer** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Handle is invalid.

Events

Event	Description
CDIEVN_TIMER_DONE	Timer operation stopped. The value field contains CTA_REASON_STOPPED.

cdiStopTones

Stops tone transmission.

Prototype

DWORD **cdiStopTones** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_FUNCTION_NOT_ACTIVE	Tone transmission was already stopped, either by a previous invocation of this function or automatically by the DSP.
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

If a board is not running in enhanced mode, use **cdiStopTones** to stop transmission of a tone to a station (started using **cdiStartTones**).

If the board is running in enhanced mode, the DSP automatically stops tone transmission at a station when the station is placed on-hook or a DTMF is detected. In this case, **cdiStartTones** returns CTAERR_FUNCTION_NOT_ACTIVE.

Refer to *Transmitting tones* on page 20 for more information.

cdiUserLed

Turns the application LED on or off.

Prototype

DWORD **cdiUserLed** (CTAHD ***ctahd***, DWORD ***value***)

Argument	Description
ctahd	Context handle.
value	Value for the application LED. Refer to the Details section for more information.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Argument is not in the valid range.
CTAERR_INVALID_CTAHD	Handle is invalid.

Details

CX 2000 boards have a green LED that the application can control. The application defines the purpose (generally maintenance, internal debugging, or other custom use). To control the LED, invoke **cdiUserLed** with one of the following values:

Value	Description
RESET_USER_LED	Turn off the application LED.
SET_USER_LED	Turn on the application LED
TOGGLE_USER_LED	Change the LED to the other state.

7 Demonstration programs

Summary of the demonstration programs

The following demonstration programs are supplied with the CDI service:

Program	Description
cdicc	Demonstrates a call center application using the CDI service with mixed board support in a single application.
cdipbx	Demonstrates a PBX application using the CDI service.

Natural Access is shipped with source code for all demonstration programs. Each demonstration program is shipped as source and makefiles.

Before you start the demonstration programs, ensure that:

- Natural Access (including the CX software) is properly installed.
- Hot Swap Manager service is started (for CX 2000C).
- Point-to-Point Switching (PPX) service is installed and started.

Preparing to run the demonstration programs

cdicc and *cdipbx* require the following setup:

- The only hardware installed in the system is a single CX 2000 or CX 2000C board and a single AG 4000, AG 4000C, AG 4040, or AG 4040C board that provides T1 trunks.
- The specific configuration files are placed in the correct directories.
- The correct system services are installed and running.

To set up and run the demonstration programs, follow these steps:

1. Build the demonstration programs.
2. Set up the board keyword and system configuration files.
3. Boot the system.
4. Run the demonstration programs.

Building the demonstration programs

As a prerequisite for using the demonstration programs, build the *ccclib* component, then build the executables for the demonstration programs. To do this, go to a command prompt and issue the appropriate *make* command for your operating system (for example, nmake for Windows).

The CDI service demonstration programs are located in the following directories:

Program	Location
ccclib	Windows: *\nms\ctaccess\demos\ccclib* UNIX: */opt/nms/ctaccess/demos/ccclib*
cdipbx	Windows: *\nms\ctaccess\demos\cdipbx* UNIX: */opt/nms/ctaccess/demos/cdipbx*
cdicc	Windows: *\nms\ctaccess\demos\cdicc* UNIX: */opt/nms/ctaccess/demos/cdicc*

Note: It is assumed that the NMS software was loaded in directory *\nms* (on UNIX, the path is */opt/nms).*

Setting up the required configuration files

Make sure that you have the correct board keyword and system configuration files in the proper directories. Follow these steps:

1. Create a board keyword file for the AG 4000, AG 4000C, AG 4040, or AG4040C board.
2. Create a board keyword file for the CX 2000 or CX 2000C board.
3. Create an *oamsys.cfg* system configuration file specifying these board keyword files.
4. Create a configuration file for the Point-to-Point Switching (PPX) service.

AG 4000 and AG 4040 board keyword file

The AG 4000 keyword file resides in *\nms\ag\cfg* on Windows (*/opt/nms/ag/cfg* on UNIX) and is called *agpi4000.cfg*. The AG 4040 board uses the same configuration file as the AG 4000 board. For more information about configuring your boards, refer to the appropriate board installation and developer's manual.

The AG 4000 keyword file for this example follows:

```
#
#
#       AG Plugin-in Config File for AG 4000
#

 Clocking.HBus.ClockSource = OSC
 Clocking.HBus.ClockMode = MASTER_A

# TCP files are shipped with the AG-CAS sub-package of Natural Access.
# Be sure that you installed the protocols that are specified below before
# trying to start a board with this configuration file.
 TCPFiles[0] = nocc.tcp           # "no trunk control" protocol
 TCPFiles[1] = wnk0.tcp           # 2-way wink protocol

# DSP (.m54) files to link in

 DSP.C5x.DSPFiles = callp.m54 dtmf.m54 mf.m54 ptf.m54 tone.m54 voice.m54 signal.m54

 DLMFiles[0] = gtp.leo
 DLMFiles[1] = voice.leo
 DLMFiles[2] = svc.leo

#-----------------------------------------------------------------------------
# Other default settings may be found in your AG 4000 Installation and
# Developer's Manual.
#-----------------------------------------------------------------------------

 XLaw = MU-LAW
```

CX 2000 board keyword file

The CX 2000 board keyword file resides *\nms\cx\cfg* on Windows (*/opt/nms/cx/cfg* on UNIX), and is called *cx2000.cfg*. For more information about configuring your boards, refer to the appropriate board installation and developer's manual.

The CX 2000 keyword file for this example follows:

```
  #
  #  Standalone operation
  #
#Clocking.HBus.ClockMode   = STANDALONE
#Clocking.HBus.ClockSource = OSC

  #
  #  Master the CT Bus (drive clock A)
  #
#Clocking.HBus.ClockMode   = MASTER_A
#Clocking.HBus.ClockSource = OSC

  #
  #  Slave to the CT Bus (slave from clock A)
  #
Clocking.HBus.ClockMode   = SLAVE
Clocking.HBus.ClockSource = A_CLOCK
SwitchConnection          = YES
```

System configuration file

When your board keyword files are complete, create a system configuration file describing the overall configuration of your system and assigning a board keyword file to each board. *oamsys* creates records in the NMS OAM database for your boards based on this file. The system configuration file is typically named *oamsys.cfg*. By default, *oamsys* looks for a file with this name when it starts up.

The Bus and Slot values in the system configuration file are required to identify the PCI bus and slot for each board. Obtain these values using the *pciscan* utility, by entering the following at a command prompt:

```
pciscan
```

pciscan displays information on the boards that are configured in the system, including the Bus and Slot values.

Once you have used *pciscan* to obtain these values, you can verify them by blinking an LED on each board. For example, assume the utility indicates that there is a board at bus 2, slot 10 and another board at bus 2, slot 14. To determine which bus/slot combination contains which board, use a bus/slot combination as arguments (separated by a space) to the *pciscan* utility. For example:

```
pciscan 2 10
```

The utility flashes the blue Hot Swap LED on the CX 2000C board (or the board locate LED on a CX 2000 board) in the given bus/slot combination. Use this bus/slot information in your configuration file for Bus and Slot.

For more information about creating a system configuration file, refer to the *NMS OAM System User's Manual*. The following sample system configuration file (*oamsys.cfg*) is for two boards (CX 2000C and AG 4000C) configured in a system:

```
#--------------------------------------------------------
# oamsys.cfg
# This is the default OAM system configuration file,
# used by the oamsys tool. It describes all the NMS
# boards in a chassis.
#
# Each board is has its own section in this file.
# Each is delimited by a user-defined board name in
# square brackets. All board names and numbers must
# be unique.
#
# You will need to modify the bus and slot numbers
# for each board to match your chassis configuration.
# You may also need to add more board sections if you
# have more boards in your chassis.
#-----------------------------------------------

    #--------------------------------------------------------------
    # Logical board number, physical bus, physical slot
    #
    #-------------------------------------------------------------

[Board0]
  Product = AG_4000C_T1
  Number = 0
# This assumes that pciscan located this board on pci bus 2, slot 10
# Make the appropriate changes for your configuration
  Bus = 2
  Slot = 10
  File = agpi4000.cfg

[Board1]
  Product = CX 2000C-48
  Number = 1
```

```
# This assumes that pciscan located this board on pci bus 2, slot 14
# Make the appropriate changes for your configuration
  Bus    = 2
  Slot   = 14
  File   = cx2000.cfg
```

PPX configuration file

The Point-to-Point Switching (PPX) service requires its own configuration file, which it uses when it is started. This service must be started by the user. The PPX configuration file (*ppx.cfg*) must reside in *\nms\ctaccess\cfg* (*/opt/nms/ctaccess/cfg* under UNIX). The following example shows the *ppx.cfg* file needed to support the demonstration program configuration:

```
#                       ppx.cfg
#
#          --- Point-to-Point Switching Service ---
#          ---        Configuration             ---
#
#   This file defines to the point-to-point switching service
#   the available bus, timeslots, and switches it may use for
#   establishing connection paths.
#
#   This configuration file contains example switch definitions
#   for the various NMS boards. Use this as a guide for
#   configuring the PPX service to your particular hardware
#   configuration.
#
#   Some Reminders:
#       - All switch numbers must be unique.
#       - The "DeviceNumber" is the board number established
#         during H/W configuration.
#       - The PPX service speaks MVIP-95, not MVIP-90.
#
#
[PPX]
Fabric
        IdleCode = MU_LAW     # Default idle pattern [ MU_LAW | A_LAW ]

        Bus H100                       # H100 bus
            CTBUS:0..15:0..31
            CTBUS:16..23:0..63
            CTBUS:24..31:0..127
        End Bus

        Bus MVIP_90                    # MVIP-90 bus
            Bus H100 (CTBUS:0..15:0..31)
        End Bus

        Switch 0                       # AG4000
             SwitchType = HMIC
             DeviceName = "agsw"
             DeviceNumber = 0
             Bus H100 ()
             Inputs
                 LOCAL:0..14(2):0..23     # Trunk (or ..29 for Es)
                 LOCAL:16..18(2):0..127   # DSP
                 LOCAL:20..26(2):0        # HDLC
             End Inputs
             Outputs
                 LOCAL:1..15(2):0..23     # Trunk (or ..29 for Es)
                 LOCAL:17..19(2):0..127   # DSP
                LOCAL:21..27(2):0         # HDLC
             End Outputs
         End Switch

         Switch 1                       # CX2000
            SwitchType = HMIC
            DeviceName = "cxsw"
            DeviceNumber = 1
```

```
            Bus H100 ()
            Inputs
                LOCAL:0..2(2):0..47       # 48 station interfaces
                LOCAL:4..6(2):0..47       # Limited DSP
            End Inputs
            Outputs
                LOCAL:1..3(2):0..47       # 48 station interfaces
                LOCAL:5..7(2):0..47       # Limited DSP
            End Outputs
        End Switch
End Fabric
```

Booting the system

Once all the board keyword files are edited and placed in the proper directories, boot the AG and CX boards using the following procedure:

1. Start the Hot Swap Manager service (CompactPCI only). For more information, refer to the *NMS OAM System User's Manual*.

   If this service is already started, proceed to step 2.

2. Start the PPX Service (Point-to-Point Switching). For more information, refer to the *Point-to-Point Switching Service Developer's Reference Manual*.

   If this service is not listed, install the PPX service by entering the following command at the prompt:

   ```
   ppxservicecfg -install
   ```

3. Configure and boot the AG and CX boards in the NMS OAM database by entering the following command at the prompt:

   ```
   oamsys
   ```

At this point, both boards are booted and ready to execute the demonstration programs.

CX 2000 call center demonstration program: cdicc

Name

cdicc

Purpose

Demonstrates a call center application using the CDI service with mixed board support in a single application.

Usage

`cdicc` [***options***]

where ***options*** include:

Option	Description
`-B` [***sw***:]***dev***	T1 board [PPX switch number:] and board device number (*oamsys.cfg* board number). Default = 0:0.
`-b` [***sw***:]***dev***	CX 2000 or CX 2000C board [PPX switch number:] and board device number (*oamsys.cfg* board number). Default = 1:0.
`-T` ***n***	T1 trunk or span number. Default = 1. The values are one-based.
`-C` ***n***	T1 channel or slot number on trunk or span. Default = 0. Value is zero based.
`-c` ***n***	CX 2000 or CX 2000C board channel or slot number. Default = 0. Value is zero based.
`-o` ***n***	0 = Inbound call center demonstration. 1 = Outbound call center demonstration (default).
`-d` ***n***	Number to dial for outbound demonstration. Default = 5551212.

For example, for the outbound demonstration, enter:

```
cdicc -B0:0 -b1:1 -C0 -c0 -T1 -o1 -d2711100
```

For the inbound demonstration, enter:

```
cdicc -B0:0 -b1:1 -C0 -c0 -T1 -o0
```

Note: The demonstration uses the wnk0 protocol.

Featured functions

cdiDisableBattery, **cdiEnableBattery**, **cdiStartSignalDetector**, **cdiStartTimer**, **cdiStartTones**, **cdiStopSignalDetector**, **cdiStopTimer**, **cdiStopTones**

Description

cdicc performs the functions of a simple call center with two call center application modes: inbound and outbound. Depending on the -o command line flag, the demonstration program does the following:

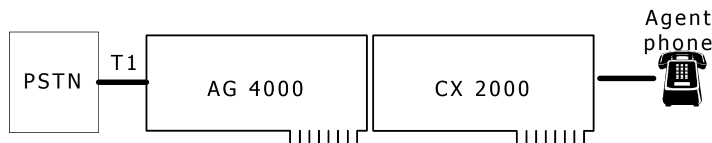
- If the demonstration is in inbound mode, the program waits for an incoming call on the specified T1 port. When an incoming call is offered, the call is answered. Call center queue prompts play requesting an account number and asking the caller to hold for the next available agent. The prompts continue to play until the agent handset connected to the CX board is taken off-hook. When the agent is off-hook and a prompt sequence concludes, the caller is provided ring-back tone and the agent is provided headset ring tone prior to call connection.
- If the demonstration is in outbound mode, the demonstration program waits for the agent handset connected to the CX board to be taken off-hook. When the agent is off-hook, the demonstration program dials the telephone number specified in the command line. The agent hears ringing and then holds for the subscriber at the end of the T1 line to answer the call.

The demonstration program performs all initial switching to voice and signaling DSPs, switching of caller and agent, and subsequent reconnection of DSPs using the Point-to-Point Switching Service (PPX).

Procedure

The following procedure assumes that you are using an AG 4000 board and a CX 2000 or CX 2000C board with a 2500-type telephone connected to one of the lines. You can run *cdicc* with the *cdipbx* demonstration program to provide the function of the PSTN. Both demonstration programs can share one pair of AG 4000/C or AG 4040/C and CX 2000/C boards.

The demonstration configuration is as follows:



cdicc demonstration program configuration

To run *cdicc*:

1. Configure the boards as described in *Preparing to run the demonstration programs* on page 53.
2. Boot the boards as described in *Booting the system* on page 58.
3. Connect a 24-port cable to the top 24-port connector on rear I/O connector of the CX 2000 or CX 2000C board and to a 24-port telephone jack breakout box.
4. Plug a telephone into port 1 of the breakout box and another telephone into port 11 of the breakout box.
5. Place a T1 loopback cable into the T1 and T2 lines of the AG board.

Inbound procedure

1. Navigate to the following directory:

   \nms\ctaccess\demos\cdicc (Windows)

   opt/nms/ctaccess/demos/cdicc (UNIX)

2. Start *cdicc* by entering the following command:

   ```
   cdicc -B0:0 -b1:1 -T2 -C0 -c10 -o0
   ```

   The application automatically detects the T1 board type and sets stream values accordingly. The call center agent telephone connected to port 11 (-c10) does not have a dial tone.

3. In another command prompt window, navigate to the following directory:

   \nms\ctaccess\demos\cdipbx (Windows)

   opt/nms/ctaccess/demos/cdipbx (UNIX)

4. Start *cdipbx* by entering the following command:

   ```
   cdipbx -B0:0 -b1:1 -T1 -C0 -c0 -N1 -n1 -x101
   ```

   The customer telephone connected to port 1 (-c0) has a dial tone.

5. Take the customer handset attached to the *cdipbx* demonstration (station in port 1) off-hook and dial a 7-digit number to access the T1 line of the *cdicc* demonstration.

   You hear a prompt welcoming you to the CX 2000 Call Center, and you are prompted for a 4-digit account number.

6. Key in any four digits on the telephone's DTMF keypad.

   You then hear three additional prompts asking you to hold, to continue to hold, and providing a short description of the CX 2000 or CX 2000C board.

   If the agent handset remains on-hook, these three prompts continue playing over and over as a set.

7. Take the agent handset off-hook.

   When the customer hears a complete prompt set, the agent handset hears a call indication tone and the customer hears a ring-back tone.

   Immediately after the tones are played, the customer and the agent are connected and can begin to talk.

8. Hang up each handset.

Outbound procedure

1. Navigate to the following directory:

   \nms\ctaccess\demos\cdicc (Windows)

   opt/nms/ctaccess/demos/cdicc (UNIX)

2. Start *cdicc* by entering the following command:

   ```
   cdicc -B0:0 -b1:1 -T2 -C0 -c10 -o1 -d2711101
   ```

   The application automatically detects the T1 board type and sets stream values accordingly. The call center agent telephone connected to port 11 (-c10) does not have a dial tone.

3. In another command prompt window, navigate to the following directory:

   \nms\ctaccess\demos\cdipbx (Windows)

   opt/nms/ctaccess/demos/cdipbx (UNIX)

4. Start *cdipbx* by entering the following command:

   ```
   cdipbx -B0:0 -b1:1 -T1 -C0 -c0 -N1 -n1 -x101
   ```

   The customer telephone connected to port 1 (-c0) has a dial tone.

5. After starting both instances of the demonstration programs, pick up the agent telephone attached to the CX 2000 or CX 2000C board.

   The predictive dialer function in the demonstration program automatically dials the phone number indicated in the -d option on the specified T1 channel.

   The agent handset hears a call indication tone and the customer telephone rings.

   When the customer answers, the agent and the customer are connected and can begin to talk.

6. Hang up each handset.

PBX demonstration program: cdipbx

Name

cdipbx

Purpose

Demonstrates a PBX (private branch exchange) application using the CDI service.

Usage

`cdipbx` [***options***]

where ***options*** are:

Option	Description
-B [***sw***:]***dev***	T1 board [PPX switch number:] and board device number (*oamsys.cfg* board number). Default = 0:0.
-b [***sw***:]***dev***	CX 2000 or CX 2000C board [PPX switch number:] and board device number (*oamsys.cfg* board number). Default = 1:0.
-T ***n***	T1 trunk or span number. Default = 1. The values are one based.
-C ***n***	T1 channel or slot number on trunk or span. Default = 0. Value is zero based.
-c ***n***	CX 2000 or CX 2000C board channel or slot number. Default = 0. Value is zero based.
-N ***n***	Number of T1 channels. Default = 1.
-n ***n***	Number of phone channels (CX 2000 or CX 2000C ports). Default = 1.
-x ***n***	Base extension number (3 or 4 digits). Default = 100.

For example:

```
cdipbx -B0:0 -b1:1 -T1 -C0 -c0 -N1 -n5 -x101
```

Featured functions

cdiDisableBattery, **cdiEnableBattery**, **cdiStartSignalDetector**, **cdiStartTimer**, **cdiStartTones**, **cdiStopSignalDetector**, **cdiStopTimer**, **cdiStopTones**

Description

cdipbx simulates the functions of a PBX. It can be used in standalone mode with a single CX 2000 or CX 2000C board or with the call center demonstration program. In the CX 2000-only mode, calls can be made to local telephone extensions only. The local telephone extensions are governed by the -x option.

When used with the call center program, the PBX program provides local and off-premises calls.

Procedure

Standalone mode

Standalone mode uses only the CX 2000 or CX 2000C board.

To run *cdipbx* in standalone mode:

1. Configure the boards as described in *Preparing to run the demonstration programs* on page 53.
2. Boot the boards as described in *Booting the system* on page 58.
3. Connect a 24-port cable to the top 24-port connector on the rear I/O connector of the CX 2000 or CX 2000C board and to a 24-port telephone jack breakout box.
4. Plug a telephone into port 1 of the breakout box and another telephone into port 5 of the breakout box.

   Note: The configuration specified for the *cdipbx* is different from that used in the *cdicc* demonstration.
5. Navigate to the following directory:

   \nms\ctaccess\demos\cdipbx (Windows)

   opt/nms/ctaccess/demos/cdipbx (UNIX)
6. At a prompt, enter the following command:

   ```
   cdipbx -b1:1 -c0 -n5 -x101
   ```

   This command sets up the PBX demonstration program to use the first 5 ports of the CX 2000 or CX 2000C board. The -c0 option starts the demonstration program with CX2000 port 0. The -n5 option causes the demonstration program to use 5 consecutive ports. The first port's extension number is 101 (-x101). This instance of the PBX demonstration supports extensions 101 through 105. If you plug phones into ports 1 through 5 of the breakout box, you can dial any of the extensions in the range 101 through 105.
7. Pick up the handset attached to port 1 of the breakout box and dial the three digit extension 105.

   The telephone plugged into port 5 of the breakout box begins to ring.
8. Pick up the handset and begin to converse.
9. Hang up each handset.

Running cdipbx with cdicc

The PBX demonstration can be run with the call center demonstration. Expanding on the standalone mode example, you can run the demonstration to place calls both locally and off-premises.

Note: The command line setup for the PBX demonstration is similar to the outbound mode of the call center demonstration.

To run *cdipbx*:

1. Configure the boards as described in *Preparing to run the demonstration programs* on page 53.
2. Boot the boards as described in *Booting the system* on page 58.
3. Connect a 24-port cable to the top 24-port connector on the rear I/O connector of the CX 2000 or CX 2000C board and to a 24-port telephone jack breakout box.
4. Plug a telephone into port 1 of the breakout box and another telephone into port 11 of the breakout box.
5. Place a T1 loopback cable into the T1 and T2 ports of the AG board.
6. Navigate to the following directory:

   \nms\ctaccess\demos\cdipbx (Windows)

   opt/nms/ctaccess/demos/cdipbx (UNIX)
7. At a prompt, enter the following command:

   ```
   cdipbx -B0:0 -b1:1 -T1 -C0 -c0 -N1 -n5 -x101
   ```

   This command sets up the PBX demonstration to use the first 5 ports of the CX 2000 or CX 2000C board as well as enabling trunk 1 (-B0:0 -T1). The -c0 option starts the demonstration with CX 2000 port 0. The -n5 option causes the demonstration to use 5 consecutive ports. The first port's extension number is 101 (-x101). This instance of the PBX demonstration supports extensions 101 through 105. If you plug telephones into ports 1 through 5 of the breakout box, you can dial any of the extensions in the range 101 through 105.

   The -N1 option indicates you are using only one of the T1 channels.
8. Start the call center demonstration in another command prompt window in either inbound or outbound mode and follow the steps in CX 2000 call center demonstration program: cdicc.

8 Errors and events

CDI service errors

The CDI service error codes are presented in two tables:

- Alphabetically by error code name.
- Numerically by hexadecimal number.

All CDI service functions return either SUCCESS, or an error code indicating that the function failed and the reason for the failure.

All CDI service error codes are Natural Access codes, defined in the *ctaerr.h* include file. For a complete list of Natural Access error codes, refer to the *Natural Access Developer's Reference Manual*.

Errors ordered alphabetically

The following table alphabetically lists the CDI service errors:

Error name	Hex	Decimal	Description
CTAERR_BAD_ARGUMENT	0x07	7	*Problem*: A function argument had an invalid value, or a required pointer argument was NULL. *Solution:* Check all arguments for valid types and ranges.
CTAERR_BOARD_ERROR	0x03	3	An unexpected error occurred on the board. This function-dependent error is due either to a board overheating or to a temperature sensor malfunction.
CTAERR_FUNCTION_ACTIVE	0x0F	15	An attempt was made to start a function that is already started.
CTAERR_FUNCTION_NOT_ACTIVE	0x0E	14	An attempt was made to stop a function that was not running.
CTAERR_INVALID_CTAHD	0x05	5	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_STATE	0x0C	12	This function is not valid in the current state.
CTAERR_OUT_OF_RESOURCES	0x08	8	*Problem*: The CDI service was unable to find a free DTMF detector resource. *Solution:* Stop an unused DTMF detector (if any) or retry later.

Errors ordered numerically

The following table numerically lists the CDI service errors:

Hex	Decimal	Error name
0x03	3	CTAERR_BOARD_ERROR
0x05	5	CTAERR_INVALID_CTAHD
0x07	7	CTAERR_BAD_ARGUMENT
0x08	8	CTAERR_OUT_OF_RESOURCES
0x0C	12	CTAERR_INVALID_STATE
0x0E	14	CTAERR_FUNCTION_NOT_ACTIVE
0x0F	15	CTAERR_FUNCTION_ACTIVE

CDI service events

The CDI service events are presented in two tables:

- Alphabetically by event name.
- Numerically by hexadecimal number.

Events ordered alphabetically

The following table alphabetically lists the CDI events:

Event name	Description
CDIEVN_A_CHANGE_OFF	A bit just reset.
CDIEVN_A_CHANGE_ON	A bit just set.
CDIEVN_BOARD_STATUS_CHANGE	Board's status changed. Only applications registered to received status change events receive this event. Refer to *Monitoring status changes* on page 23.
CDIEVN_CX_HEART_BEAT_BACK	Communication with the DSP is re-established.
CDIEVN_CX_HEART_BEAT_LOST	Communication with the DSP is lost.
CDIEVN_DTMF_ENDED	Current DTMF ended.
CDIEVN_DTMF_STARTED	DTMF started. The ASCII representation of the key is in the value field.
CDIEVN_FLASH_HOOK	Station issued a flash-hook.
CDIEVN_OFF_HOOK	Station picked up.
CDIEVN_ON_HOOK	Station hung up.
CDIEVN_RING_IS_OFF	Ring cycle finished. FSK transmission is allowed.
CDIEVN_RING_IS_ON	Ring cycle started. FSK transmission is not allowed until CDIEVN_RING_IS_OFF is received.
CDIEVN_TIMER_DONE	On-board timer stopped. The event value field contains the reason.
CDIEVN_TIMER_START	On-board timer started.
CDIEVN_TIMER_STOP	On-board timer stopped.
CDIEVN_TIMER_TICK	On-board timer reporting a timer tick.
CDIEVN_TONE_DONE	Tone transmission is complete.

Events ordered numerically

The following table numerically lists the CDI events:

Hex	Decimal	Event name
0x1F2001	2039809	CDIEVN_TONE_DONE
0x1F2002	2039810	CDIEVN_DTMF_STARTED
0x1F2003	2039811	CDIEVN_DTMF_ENDED
0x1F2004	2039812	CDIEVN_ON_HOOK
0x1F2005	2039813	CDIEVN_OFF_HOOK
0x1F2006	2039814	CDIEVN_FLASH_HOOK
0x1F2007	2039815	CDIEVN_A_CHANGE_OFF
0x1F2008	2039816	CDIEVN_A_CHANGE_ON
0x1F2009	2039817	CDIEVN_BOARD_STATUS_CHANGE
0x1F200A	2039818	CDIEVN_TIMER_TICK
0x1F200B	2039819	CDIEVN_TIMER_DONE
0x1F200C	2039820	CDIEVN_CX_HEART_BEAT_LOST
0x1F200D	2039821	CDIEVN_CX_HEART_BEAT_BACK
0x1F200E	2039822	CDIEVN_TIMER_START
0x1F200F	2039823	CDIEVN_RING_IS_ON
0x1F2010	2039824	CDIEVN_RING_IS_OFF

9 Parameters

CDI service parameters

The CDI service stores its parameters in the following data structures:

- CDI_DTMFDETECTOR_PARMS
- CDI_GLOBAL_PARMS
- CDI_RING_PARMS
- CDI_SIGNALDETECTOR_PARMS
- CDI_TONE_PARMS

Use the following Natural Access functions to obtain or modify parameter information:

Function	Description
ctaGetParmByName	Retrieves a single field for a given parameter name.
ctaSetParmByName	Modifies a single field for a given parameter name.
ctaGetParmInfo	Retrieves a parameter field definition.
ctaGetParms	Returns parameter values for a given parameter structure.
ctaRefreshParms	Resets the values of all context parameters on a context to the global defaults.

For information about parameter management in Natural Access, refer to the *Natural Access Developer's Reference Manual*.

CDI_DTMFDETECTOR_PARMS

CDI_DTMFDETECTOR_PARMS controls the DTMF detector.

Dependent function

cdiStartDTMFDetector

Field name	Type	Units	Default	Description
onqualtime	DWORD	ms	30	Indicates the minimum duration for a DTMF (in ms). Tones shorter than this duration are ignored. Valid values: 30 - 32766
offqualtime	DWORD	ms	30	Indicates the minimum duration of silence after a DTMF (in ms). If two identical DTMFs follow one another, and the interval of silence between the tones is shorter than offqualtime, the detector considers them a single DTMF instance. Valid values: 30 - 32766
DTMF_mode	DWORD	Flag	0	Determines when CDIEVN_DTMF_STARTED is sent to the application, indicating that a DTMF tone started. Valid values: 0 = Event is sent after the qualification time (onqualtime). 1 = Event is sent when the key is released.

CDI_GLOBAL_PARMS

CDI_GLOBAL_PARMS controls the DSP functions.

Dependent functions

cdiStartDTMFDetector, **cdiStartRing**, **cdiStartTones**, **cdiStartSignalDetector**

Field name	Type	Units	Default	Description
EnhancedMode	DWORD	N/A	1	Enables the DSP to perform call control operations that otherwise would have to be performed by the host application (for example, stopping tones or DTMF detection upon on-hook). Refer to *CX 2000 modes* on page 17 for information about enhanced mode. Valid values: 0 = Enhanced mode disabled. 1 = Enhanced mode enabled.

CDI_RING_PARMS

CDI_RING_PARMS controls the type of ring.

Dependent function

cdiStartRing

Field name	Type	Units	Default	Description
cadence_type	DWORD	N/A	0	Specifies the type of ring the station uses. By convention, 0 is programmed to the national standard by the application. Valid values: 0 = Single ring 1 = Double ring 2 = Triple ring Refer to *Ringing* on page 18 for more information.

CDI_SIGNALDETECTOR_PARMS

CDI_SIGNALDETECTOR_PARMS controls the operation of the signal detector (A and B bits).

Dependent function

cdiStartSignalDetector

Field name	Type	Units	Default	Description
timeon	DWORD	ms	30	Sets the duration to qualify an off-hook condition. Valid values: 2 - 32766
timeoff	DWORD	ms	30	Sets the duration to qualify an on-hook condition. Valid values: 2 - 32766
min_flash_hook	DWORD	ms	350	Sets the duration to qualify a flash-hook condition. Valid values: 2 - 32766
max_flash_hook	DWORD	ms	1000	Sets the duration to qualify a disconnect condition. Valid values: 2 - 32766

CDI_TONE_PARMS

CDI_TONE_PARMS controls the operation of the tones.

Dependent function

cdiStartTones

Field name	Type	Units	Default	Description
freq1	DWORD	Hz	0	Frequency component. Valid values: 0 - 4000
freq2	DWORD	Hz	0	Second frequency component if the tone consists of two frequency components. Set to 0 if the tone consists of only one frequency. Valid values: 0 - 4000
ampl1	INT32	dB	0	Amplitude of freq1. Valid values: 0 to -100
ampl2	INT32	dB	0	Amplitude of freq2. Set to -100 if freq2 is 0. Valid values: 0 to -100
ontime	DWORD	ms	0	Duration of the tone. Valid values: 0 - 8000
offtime	DWORD	ms	0	Duration of the silence after the tone, before the tone is repeated (because iterations, ontime2, or both are greater than 0). Valid values: 0 - 8000
ontime2	DWORD	ms	0	Duration of the second iteration of the tone if the cadence consists of two iterations. Set to 0 if the cadence consists of only one iteration. Valid values: 0 - 8000
offtime2	DWORD	ms	0	Duration of the silence after the second iteration of the tone, before the cadence begins again. Set to 0 if the cadence consists of only one iteration. Valid values: 0 - 8000
iterations	INT32	N/A	-1	Number of times to repeat the cadence. Valid values: 1 - 65535, or -1 for infinite repeats

Index

B

battery 17, 28, 29

board management 23, 23, 23, 24

board status 30, 40, 48

C

cadence 18, 20

call center demonstration program 59

call control 17, 18, 20, 20, 21, 22, 59

caller ID 20

ccclib 54

CDI_BOARD_STATUS 23, 30

CDI_DTMFDETECTOR_PARMS 20, 34, 72

CDI_GLOBAL_PARMS 17, 72

CDI_RING_PARMS 18, 36, 73

CDI_SIGNALDETECTOR_PARMS 18, 38, 73

CDI_TONE_PARMS 20, 43, 74

cdiapi.lib 16

cdiAssertSignal 27

cdicc demonstration program 59

cdidef.h 16

cdiDisableBattery 28

cdiEnableBattery 29

CDIEVN_XXX_XXX 69

cdiGetBoardStatus 30

cdiGetSignalState 32

CDIMGR 14

cdipbx demonstration program 63

cdiReadThermometer 33

cdiStartDTMFDetector 34

cdiStartRing 36

cdiStartSignalDetector 38

cdiStartStatusMonitoring 40

cdiStartTimer 41

cdiStartTones 43

cdiStopDTMFDetector 45

cdiStopRing 46

cdiStopSignalDetector 47

cdiStopStatusMonitoring 48

cdiStopTimer 49

cdiStopTones 50

cdiUserLed 51

configuration files 53

contexts 14

CTA_EVENT 15, 69

ctaCreateContext 14

ctaCreateQueue 14

CTAERR_XXX_XXX 67

ctaGetParmByName 13

ctaGetParmID 13

ctaGetParmInfo 13

ctaGetParms 13

ctaInitialize 14

ctaOpenServices 15

ctaRefreshParms 13

ctaSetParmByName 13

ctaWaitEvent 15

D

DC loop power 17

demonstration programs 53

  cdicc 59

  cdipbx 63

  configuration files 53

  prerequisites 53, 58

DTMF 20, 34

E

enhanced mode 17, 72

errors 67

event queues 14

events 15, 69

F

FSK 20

functions 26

battery 28, 29

board management 30, 33, 40, 48, 51

board status 30, 40, 48

call control 27, 28, 29, 32, 34, 36, 38, 41, 43, 45, 46, 47, 49, 50

DTMF 34, 45

LED 51

ringing 36, 46

signaling bits 27, 32, 38, 47

station power 28, 29

temperature 33

timers 41, 49

tone transmission 43, 50

H

Hot Swap Manager service 53, 58

L

LED 24, 51

libcdiapi.so 16

N

Natural Access environment 11, 14

O

overheating 23

P

parameters 71

CDI_DTMFDETECTOR_PARMS 72

CDI_GLOBAL_PARMS 72

CDI_RING_PARMS 73

CDI_SIGNALDETECTOR_PARMS 73

CDI_TONE_PARMS 74

PBX demonstration program 63

pciscan 56

Point-to-Point Switching (PPX) service 53, 53

ppx.cfg 57

R

ringing 18, 36, 46

S

signaling bits 18, 22, 27, 32, 38, 47

station 17, 18, 18, 20, 20, 22

status of CX board 23, 23, 23, 30, 40, 48

T

temperature 23, 30, 33

timers 21, 41, 49

tone transmission 20, 43, 50