Dialogic
Making Innovation Thrive™

Dialogic® NaturalAccess™ Software Developer's Manual

October 2009 64-0495-01

www.dialogic.com

Copyright and legal notices

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
9000-6423-17	March, 2001	SRR, Natural Access 4.02
9000-6423-18	April, 2001	SRR, Natural Access 4.03
9000-6423-19	June, 2001	SRR
9000-6423-21	November, 2001	SRR, NACD 2002-1 Beta
9000-6423-22	May, 2002	LBG, NACD 2002-1
9000-6423-23	April, 2003	SRR, NACD 2003-1
9000-6423-24	April, 2004	MCM, Natural Access 2004-1
9000-6423-25	March, 2005	CYF, Natural Access 2005-1
9000-6423-26	October, 2007	DEH, Natural Access 2005-5
64-0495-01	October, 2009	LBG, NaturalAccess R9.0
Last modified: September 2, 2009		

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

Table Of Contents

1 Introduction

The *Dialogic® NaturalAccess™ Software Developer's Manual:*

- Contains detailed information about NaturalAccess concepts, architecture, and application development.
- Provides a reference of functions and errors.

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

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.

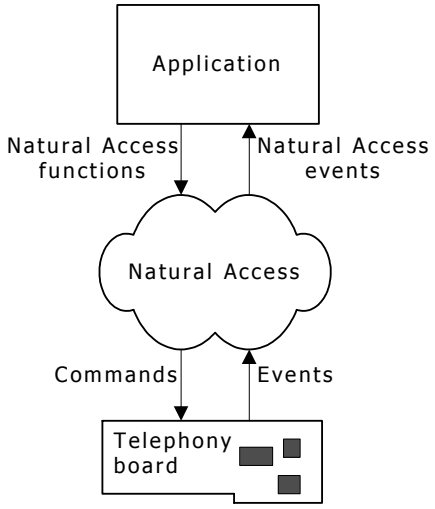
Former terminology	Dialogic terminology
CG 6060 Board	Dialogic® CG 6060 PCI Media Board
CG 6060C Board	Dialogic® CG 6060C CompactPCI Media Board
CG 6565 Board	Dialogic® CG 6565 PCI Media Board
CG 6565C Board	Dialogic® CG 6565C CompactPCI Media Board
CG 6565e Board	Dialogic® CG 6565E PCI Express Media 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 Natural Access

Natural Access definition

Natural Access is a development environment that provides standard programming interfaces for hardware-independent telephony functions. The following illustration shows the relationship of Natural Access to an application and to a telephony board:



Natural Access provides

- A standard set of telephony functions grouped into logical services.

  Natural Access services provide a variety of functions such as establishing and maintaining network connections, determining call status, playing and recording voice messages, and generating and detecting DTMF tones.

- A standard application programming interface (API) for services that are independent of the hardware.

  The Natural Access environment is designed to support different hardware and services from different vendors. Applications can access services providing the same programming interface without needing to know the type of hardware implementing the services. When using the Natural Access development environment, you do not program for a specific piece of hardware, but more abstractly for a service with a consistent API.

- A standard interface for accessing and changing service parameters.

  All services providing the same functionality have a standard set of parameters. The service's parameters have default values that are sufficient for most configurations. To adapt the service for special configurations, the parameters can be modified to enable or disable features. Natural Access provides a standard set of functions to retrieve and modify service parameters.

- Error trapping features.

  Natural Access provides a mechanism to register an application-defined error handler.
- An architecture that supports as-needed resource allocation.

  During Natural Access initialization, you specify the services required for the application. Resources for only those services will be allocated.
- Support of multiple programming models for most application needs.

  The Natural Access architecture enables single-threaded, multi-threaded, and multi-process programming models to meet any application program requirements.

Verifying the installation

Before running the verification procedures described in this section, make sure that the boards are installed, configured, and verified. For more information, see the board-specific installation and developer's manual.

To verify the Natural Access installation, follow these steps:

Step	To...	Run this program...
1	Display the version information for all services and service managers that are registered during Natural Access initialization.	Service version confirmation (*ctavers*)
2	Start the Natural Access Server (*ctdaemon*), or verify that it is running.	Natural Access Server (*ctdaemon*)
3	Verify Natural Access, NCC service, and ADI service functionality.	Interactive test (*ctatest*)

Confirming installed service versions (ctavers)

ctavers retrieves and displays the version information for all services and service managers so that you can confirm that the correct service versions are available for *ctdaemon* operation.

To run *ctavers*, enter the following command on the command line:

```
ctavers
```

ctavers lists the versions and build dates of all Natural Access core components.

For more information on *ctavers*, see *Service version confirmation: ctavers* on page 237.

Starting the Natural Access Server (ctdaemon)

Start the Natural Access Server (*ctdaemon*) as a Windows service, a UNIX daemon, or a console application. The Natural Access Server (*ctdaemon*) must always be running to provide Natural Access functionality.

Windows service

Following the Natural Access installation, reboot your system. When you reboot the system, the Natural Access Server (*ctdaemon*) starts automatically as a Windows service. If the service is not running, start it in one of the following ways:

- Access a command prompt and enter:

```
net start ctdaemon
```

- In the Windows Control Panel, double-click on **Services** and start the NMS CT daemon service.

UNIX daemon

Access a command prompt and enter:

```
ctdaemon
```

Console application for Windows or UNIX:

This method allows full console interaction with *ctdaemon.*

Access a command prompt and enter:

```
ctdaemon -i
```

For more information on *ctdaemon* options, refer to *Natural Access Server: ctdaemon* on page 238.

Verifying functionality (ctatest)

ctatest executes many of the NCC and ADI service functions from a simple command menu. *ctatest* also enables you to create and manipulate contexts. Connect a telephone to the NMS board being tested to play, record, and hear tones.

ctatest displays information similar to the following:

```
NA Demonstration and Test Program  V.2.1 (Jun  3 2001)   NA Version is 4.03
H  help              Q  quit              !  repeat command   W  trace
PI get parm id       RP refresh prm       PD set parms        VD view parms

CP close svcs(old)   OP open svcs(old     X  show state

AX attach context    CX create context    DX destroy context  EC enumerate cnxts
PX create pers. cnt  VX create void cntx  CV close svcs(new)  OV open svcs(new)
CN cntxt info        CH change context

GM get event source  SM set event source  CE cont event fetch GE get 1 event
SE stop event fetch  LE loopback event    FE format event     EH set event hdlr

SP start protocol    UP stop protocol     AC answer call      PC place call
RC release call      BC block calls       UC unblock calls    C? call status
JC reject call       CC accept call       CI accept address   AS assert signal

PF play file         PM play memory       P? play/rec status  PS play/rec stop
RF record file       RM record memory     R? play/rec status  RS play/rec stop
MG modify play gain  MS modify playspeed  EI encoding info
DG digit get         DF digit flush       CD collect digits   SC collect stop
DP digit peek        FS FSK Send          FR FSK Receive      FA FSK Abort rcv
GN gen net tone      GT gen user tone     GD gen DTMFs        GS generation stop

TP toggle pattern    GW generate wink     ST start timer      AT abort timer
EB enable bit det    DB disabl bit det    SQ signal bit query
EE enable energydet  DE disabl energydet  ED enable DTMF det  DD disabl DTMF det
ET enable tone det   DT disabl tone det   EM enable MF det    DM disabl MF det
SD start dial        AD abort dial        CB callprog begin   CS callprog stop

SV get svc version   RV get svr version   FF find file        TR set server trace
TS set time slot     SS set default srv   ES query services   SH server shutdown
CL get context list

OC hold call         RR retrieve call     DC disconnect call  ND send digits
IH set call handle   L? get call status   CQ query capability VT supervised xfer

AO attach object     DO detach object     DH get object descr

Initializing and creating the queue...
ok.
```

Enter one- and two-letter commands to execute Natural Access, ADI service, Voice Message service, and NCC service functions. Some commands prompt for additional information, such as frequencies and amplitudes for tone generators.

For more information on running *ctatest*, see *Interactive test: ctatest* on page 198.

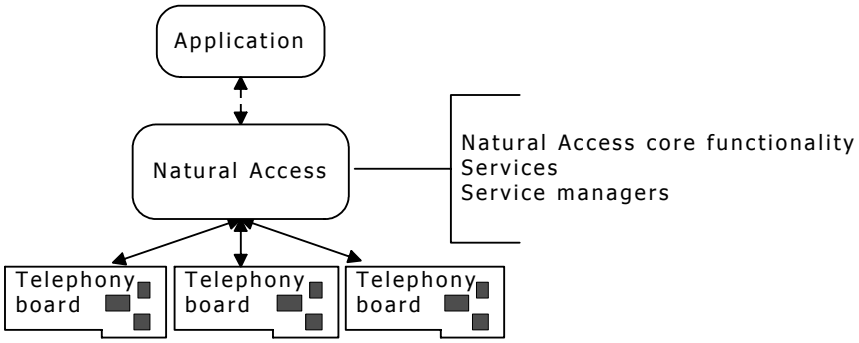
4 Natural Access architecture

Natural Access components

Natural Access includes the following components:

Component	Description
Core	Provides lower-level command processing functions that are available in linkable libraries or as part of an executable Natural Access Server (*ctdaemon*).
Services	Groups of related telephony functions.

The following illustration shows the Natural Access components:



Natural Access core functionality

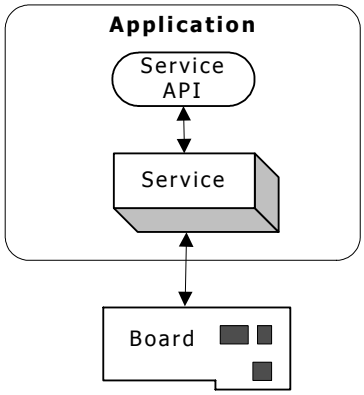
When creating applications with Natural Access, you use Natural Access core functions to set up the operating environment and service functions to access telephony functionality. Natural Access core functions provide a standardized interface for performing tasks such as

- Initializing the Natural Access environment
- Creating, sharing, and destroying event queues
- Creating, sharing, and destroying contexts
- Opening and closing services
- Controlling parameters, events, and tracing

Services

A service is a group of logically related telephony functions. A service can be implemented on multiple boards. All services with the same functionality have a standard API regardless of the hardware providing the functionality, allowing device-independent programming.

The following illustration shows a standard application programming interface (API):



Each service has a defined set of standard parameters managed by Natural Access. Equivalent services implemented on different hardware platforms can have additional parameters called extension parameters to handle unique functionality for that specific board. Natural Access provides the capability to access and modify both types of parameters.

Natural Access provides two types of services:

- Base services that provide generic functionality across a variety of applications.
- Domain services that provide special purpose functionality for a specific group of tasks.

Base services

Natural Access provides the following base services:

- ADI service
- Digital Trunk Monitor (DTM) service
- Natural Call Control (NCC) service
- Operations, Administration, and Maintenance (OAM) service
- Parameter Management (PRM) service
- Point-to-Point Switching (PPX) service
- Switching (SWI) service
- Voice Message (VCE) service

For more information about a service, refer to the service-specific reference manual.

Domain services

Natural Access provides the following domain services, available as separate installable packages:

- NaturalFax
- NaturalConference
- Media Stream Processing Protocol (MSPP)

Service managers

Service managers are software wrappers surrounding one or more services. They implement the required Natural Access binding functions that enable a service to integrate into Natural Access. Service managers are provided as dynamic link libraries (DLL) in Windows or shared libraries in UNIX.

Note: Applications specify service managers during Natural Access setup.

Principles of operation

The basic principles of Natural Access operation involve an asynchronous programming model that utilizes concurrent processing and provides a flexible method of organizing applications.

The application uses the services' functions. When called, most functions return immediately, where SUCCESS indicates the function was initiated. The application can call other functions while Natural Access is processing the command.

Natural Access sends the command to the service, which in turn sends a command to a telephony board. The board performs the requested functions and sends events to the service indicating its state, for example, function was started. The service sends events to Natural Access, which makes them available to the application.

The client-server architecture of Natural Access enables Natural Access to process application commands locally and remotely. Applications and Natural Access can run on the same host or on separate hosts. A host runs multiple instances of Natural Access Server (*ctdaemon*) and is identified by a host name or IP address.

This topic describes:

- Contexts
- Service instances
- Events
- Command processing
- Event processing

Contexts

Natural Access organizes services and accompanying resources around a single processing context. To access service functionality, an application creates a context and attaches the services it requires. A context usually represents an application instance controlling a single call. The context maintains defined parameters for each type of service, allowing each call to have its own characteristics.

Not all contexts are associated with a call. For example, an application performing voice conversions does not require a telephone line.

Context handles

When an application creates or attaches to a context, a context handle is returned. A context handle provides a programming reference to the resources allocated for a context by the Natural Access implementation.

Void context handles provide an application with access to the Natural Access Server (*ctdaemon*) when an associated context object is not applicable. The void context handle is only a reference to addressing information that uniquely identifies the server on which Natural Access commands are executed and not an actual context object. For example, **ctaGetVersionEx** returns the build date and compatibility level of the server itself. This information is global to the server and is associated with all contexts created by the server, not with just one context.

For more information, refer to *Creating contexts* on page 36.

Service instances

When an application opens a service on a context, it creates a service instance. A service instance defines a service (a logical group of functions) and its associated resource data (for example, MVIP board, stream, and timeslot). A context is a collection of service instances that groups functions and resources, creates service dependencies, and handles parameter management. A service must always be associated with a context, but not all contexts require a service.

An application can create only one service instance of a particular type (for example, ADI or Switching) on each context. For example, to use the ADI service functions on all 120 channels of a CG resource board, you must create 120 contexts and open the ADI service on each context.

Service objects

A service object is a collection of data representing a particular instance of the resource. For example, the Natural Call Control service is a logical group of functions that supports answering, placing, transferring, and disconnecting a call. The managed resource is the telephone line. The Natural Call Control service uses service objects to manage information about each call. The service object maintains such information as the telephone number being called, the state of the call, and the direction of the call.

Service object handles

Natural Access Server returns a service object handle that uniquely identifies an instance of the service object. A service object handle provides a programming reference to an instance of a service object. A service object handle is similar to a context handle.

All service object handles indirectly refer to a context handle. Natural Access can derive a context handle from a service object handle.

Multiple service object handles can be associated with one instance of a service object, but each service object handle is unique.

Events

All Natural Access functions return a status or return code. For asynchronous functions, the return code indicates whether the function was initiated. During function execution, events are generated indicating the occurrence of certain conditions or state changes.

Events are represented as CTA_EVENT C data structures, which contain the following fields:

Field	Description
id	Natural Access event code for the event. All Natural Access event codes are prefixed with XXXEVN_ where XXX is the three-letter service abbreviation, for example, VCEEVN_PLAY_DONE.
ctahd	Context handle on which the event occurred as returned from **ctaCreateContext**, **ctaCreateContextEx**, or **ctaAttachContext**.
timestamp	Time, in milliseconds, when the event was generated.
userid	User context value supplied to **ctaCreateContext** or **ctaCreateContextEx**.
value	Event-specific value used to communicate a single 32-bit datum.
size	Size (in bytes) of the area pointed to by buffer. This field also indicates whether a data buffer associated with the event must be freed using **ctaFreeBuffer**. If the buffer is NULL, this field can be used to hold an event-specific value.
buffer	If a buffer is returned with the event, this field points to data that is event-specific. The field contains an application buffer address while the event's size field contains the actual size of the buffer.
objHd	Service object handle (for example, swihd, vcehd, callhd).

Event queues

An event queue is the communication path from a service to an application. A service generates events indicating certain conditions or state changes. An application retrieves the events from the event queue.

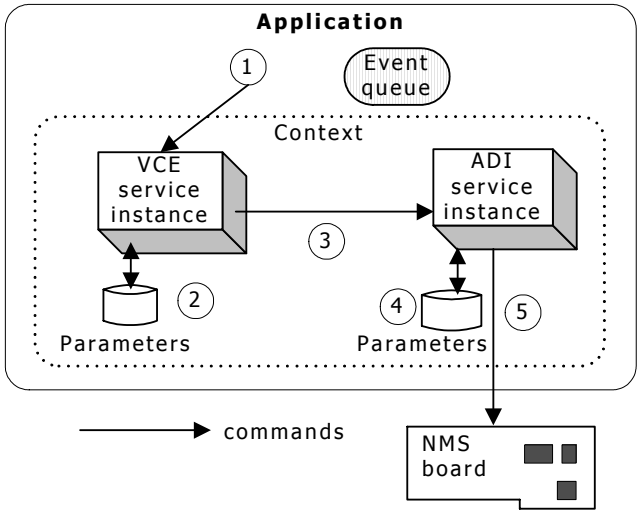
When the event queue is created, you can specify the service managers to be attached to it. Only services supported by the specified service managers are allowed to be opened on the contexts attached to the queue.

When creating a context, you specify an event queue. All events from the services that you open on the context are sent to the associated event queue.

One event queue can be created for the entire application process or multiple event queues can be used. The number of event queues you create depends on the programming model used. Some models use one event queue to handle all contexts; multiple telephone calls are handled with the same event queue. Other models use one event queue for each call. Refer to *Creating event queues* on page 35 for more information.

Command processing

The following illustration shows the interaction between the application and Natural Access components when the application calls a Voice Message (VCE) service function. Commands are processed from the application program down to the hardware.

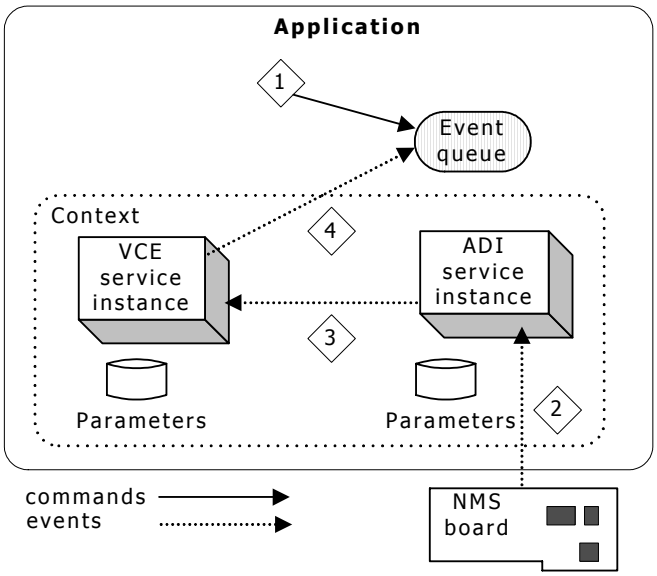


Commands are processed in the following sequence:

Stage	Description
1	The application invokes a Voice Message service (VCE) function with a context handle or service object handle.
2	The Voice Message service instance accesses its context parameters when processing the command.
3	Because the Voice Message service requires hardware resources and the ADI service on the same context provides the player/recorder functionality, the Voice Message service sends a command to the ADI service instance.
4	The ADI service accesses its context parameters when processing the command.
5	The ADI service sends a command to the board driver.

Event processing

The following illustration shows the interaction between the application and Natural Access components when the application retrieves an event for the Voice Message service. Events are processed from the hardware up to the application.



Events are processed in the following sequence:

Stage	Description
1	The application calls **ctaWaitEvent** to wait for asynchronous events requiring processing by Natural Access services.
2	The ADI service receives a board event from the board driver and processes the event.
3	The ADI service sends an ADI event to the Voice Message service instance.
4	The Voice Message service processes the ADI event and sends a VCE event to the event queue associated with the context.

For more information, refer to *Receiving events on event queues* on page 46.

Communicating with Natural Access

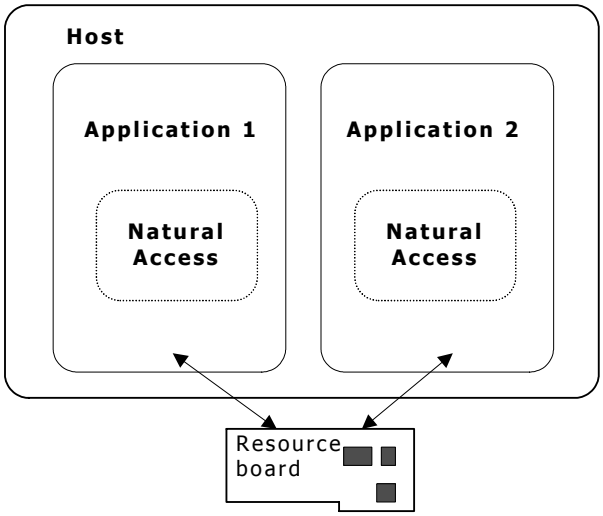
By providing core functions and service functions, Natural Access creates an operating environment that enables applications to interact with system resources (for example, the boards). A wide range of programming models is available to suit most application requirements.

An application can access Natural Access functionality in one of the following ways:

- In-process
- Client-server

In-process approach

By using the in-process approach shown in the following illustration, each application maintains an independent Natural Access programming environment. An application uses the Natural Access functionality to create, manage, and destroy resources. This approach provides the most efficient processing.



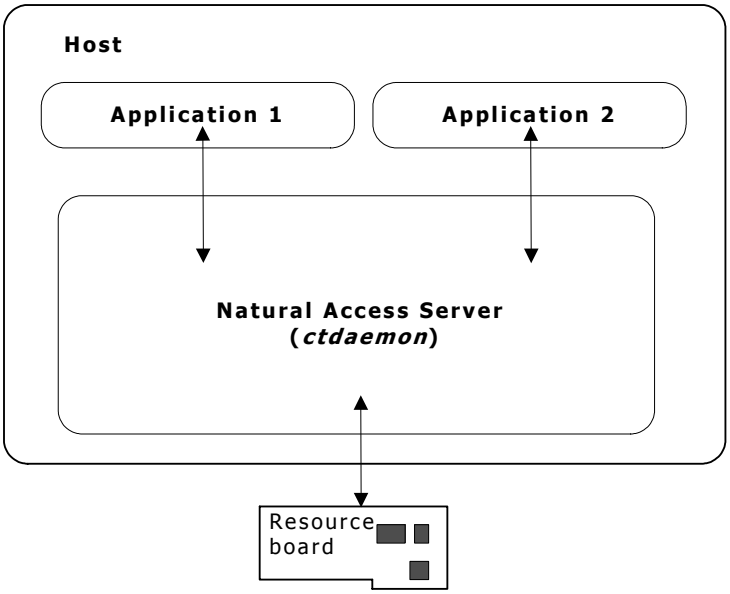
Client-server approach

The client-server architecture of Natural Access enables applications to share resources and distribute processing tasks. Natural Access Server (*ctdaemon*) creates and manages processing contexts on behalf of the applications. This section provides three examples of the client-server approach, based on the location of the application in relation to Natural Access Server (*ctdaemon*).

Example 1: local host

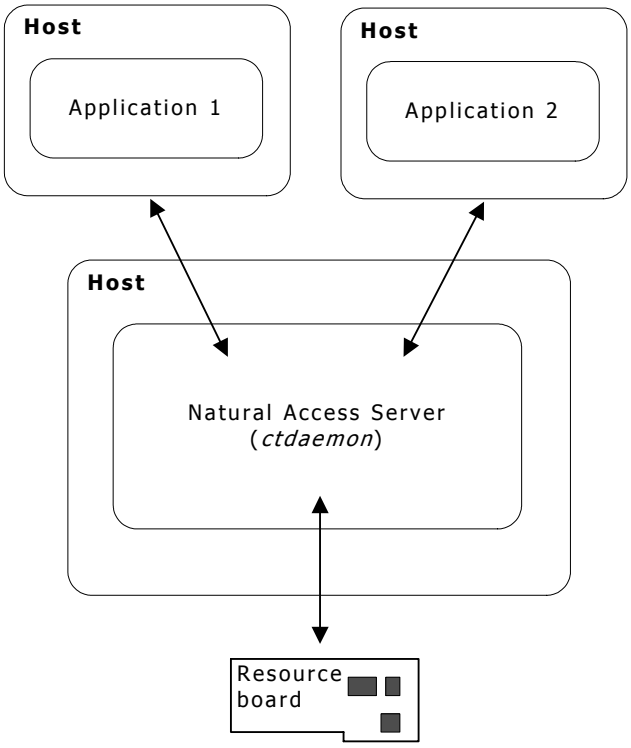
In the configuration shown in the following illustration, multiple applications share a common resource locally. The applications and Natural Access Server (*ctdaemon*) are running in the same host. For example, in an IVR system, one application can establish connections to and from the PSTN, while another plays voice files over this connection.

In addition, applications can perform a task and then relinquish control of the shared resource to another application. For example, one application can answer an incoming call. Then, based on an established criteria, the application can route call processing to another application. The first application can then return to waiting for incoming calls.



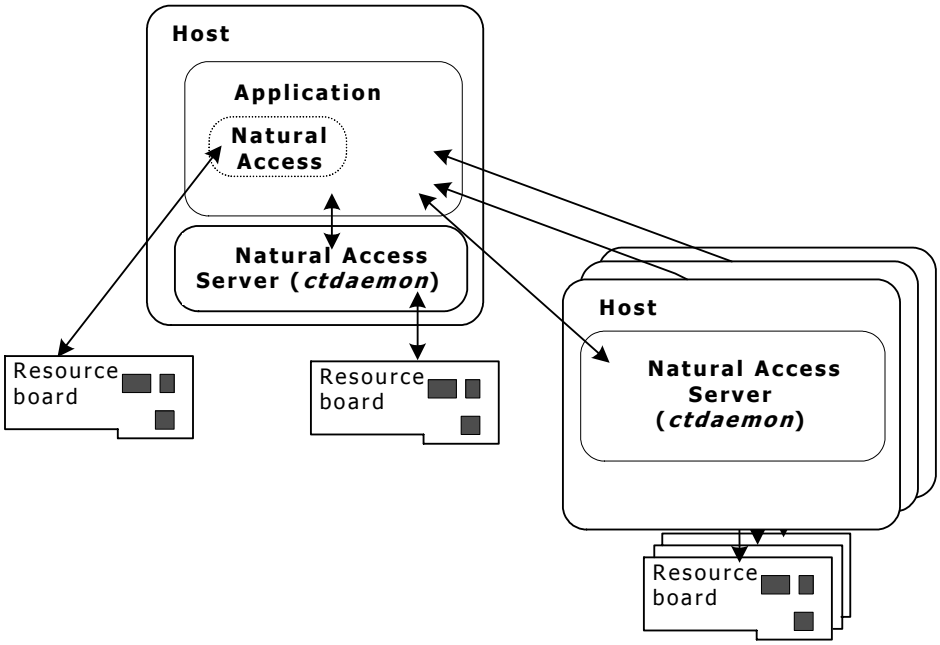
Example 2: remote host

In the configuration shown in the following illustration, applications share resources through a remote Natural Access Server (*ctdaemon*). The applications and Natural Access Server (*ctdaemon*) are running in separate hosts. This approach is an efficient method of performing multiple tasks utilizing a remote Natural Access Server (*ctdaemon*) over a network connection.



Example 3: multiple hosts

In the configuration shown in the following illustration, an application simultaneously uses an independent Natural Access programming environment to manage local resources, and a local instance and multiple remote instances of Natural Access Server (*ctdaemon*) to manage shared resources. This application utilizes all the methods of communicating with Natural Access.



5 Developing applications

Programming models

Natural Access is a collection of re-entrant libraries capable of supporting programming models to suit most application requirements. Natural Access supports single-threaded (single-thread-per-process) as well as multi-threaded (multiple-threads-per-process) applications. It also supports multiple contexts accessing multiple instances of Natural Access Server (*ctdaemon*) on various hosts.

You can create different programming models using contexts and event queues. This topic contains the following sample programming models:

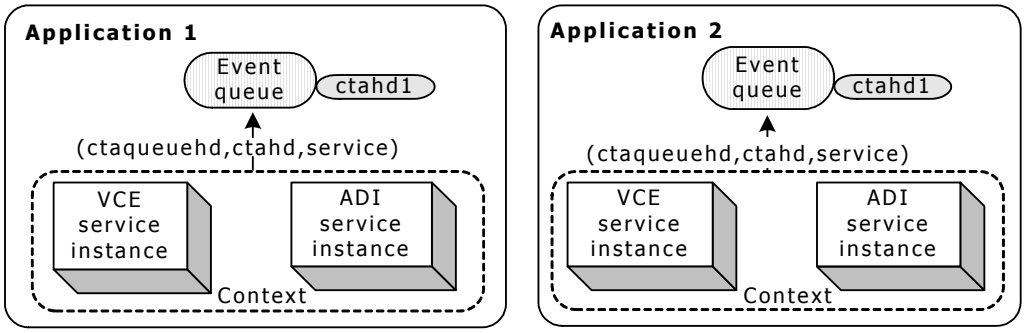
- One context, one event queue per application
- Multiple contexts, one event queue per context
- Multiple contexts, one event queue per application
- Multiple contexts, one event queue for all hosts

Variations on these models are also supported.

One context, one event queue

The one context, one event queue configuration shown in the following illustration is the simplest approach for application development. This model uses multiple application processes, each with its own event queue.

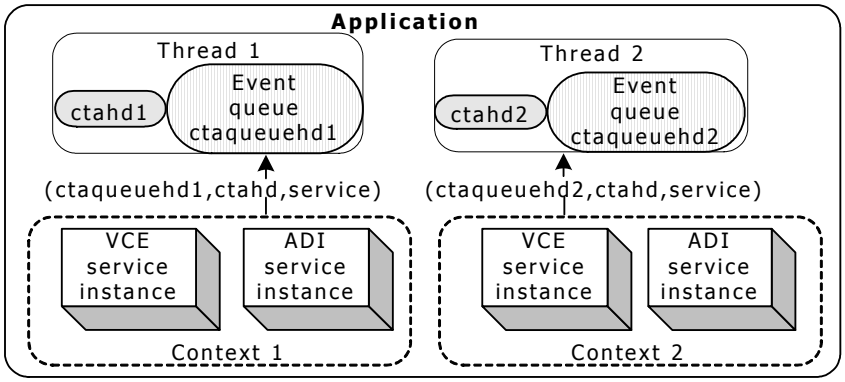
This configuration is best suited to systems with a low number of contexts. Using this configuration, applications can create synchronous wrappers allowing a sequential stateless thread of execution.



Multiple contexts, one queue per context

Multi-threaded applications use the multiple contexts, multiple event queues configuration shown in the following illustration. Each thread has its own event queue with one context. The thread can be developed with synchronous functions as in the one event queue, one context configuration.

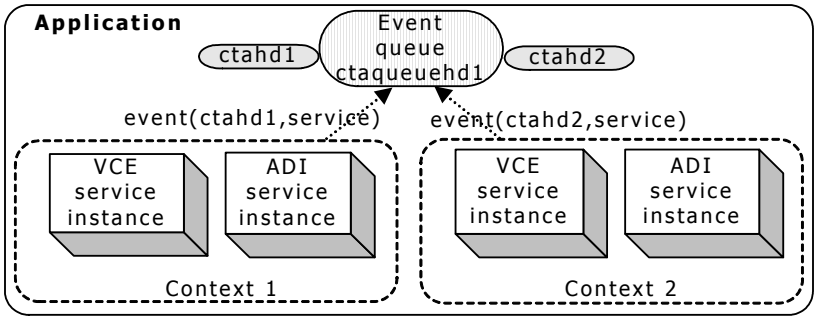
Each thread controls an event queue. A thread can sleep waiting for an event from its services without interfering with the reception of events in other threads.



Multiple contexts, one queue per application

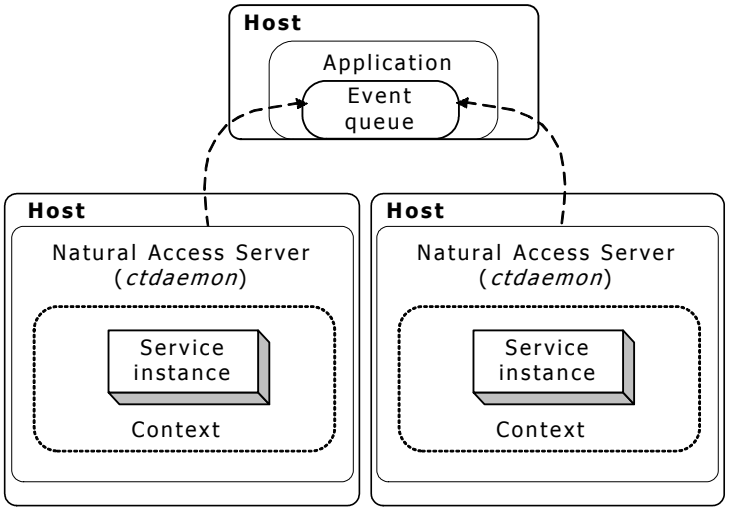
The multiple contexts, one event queue configuration shown in the following illustration is the most efficient for system resource utilization. The operating system resource demand is minimized at the expense of development complexity.

In this configuration, the queue receives events for all contexts. This approach requires the application to implement a state-machine model.



Multiple contexts, one queue for all hosts

An application can use services on multiple instances of Natural Access Server (*ctdaemon*) as shown in the following illustration. Each context is created on a specific server. A single event queue can receive events from contexts created on all specified servers.



Natural Access configuration file

The Natural Access configuration file (*cta.cfg*) specifies the Natural Access configuration. It includes the following specifications:

- Generic system settings
- Application-specific settings
- Natural Access Server (*ctdaemon*) settings
- Service-specific parameter defaults

Natural Access provides a default *cta.cfg* that you can modify for your specifications.

This topic presents:

- Default *cta.cfg*
- *cta.cfg* sections
- Setting the default server

Default cta.cfg

The following example shows the default *cta.cfg*:

```
#=========================================================================
# cta.cfg
#
# This is an example of a file that specifies Natural Access configuration.
# It allows you to specify:
#    Generic operational settings that apply to Natural Access
#    Server (ctdaemon) and all Natural Access applications.
#    Note: these settings can be overwritten by a Natural Access
#           application via ctaInitialize.
#    Application specific settings.
#    Natural Access Server (ctdaemon) specific settings.
#    Service-specific parameter defaults.
#
#=========================================================================



#=========================================================================
#  Natural Access System Configuration (ctasys)
#
# Valid options are:
#   Service = name, dll   - tells the application about available "services"
#                         - tells the Natural Access Server what
#                              "services" to export
#
# Note: NCC should always precede ADI when both services are listed.
#=========================================================================
[ctasys]
Service = ppx, ppxmgr
Service = swi, swimgr
Service = vce, vcemgr

#=========================================================================
# Natural Access application configuration (application)
#
#=========================================================================
[application]
# DefaultServer = {inproc | localhost | server_name}
DefaultServer = inproc

# ContextNameUniqueness = {0|1}  - Specification of responsibility for
#                                  the context name uniqueness
#                                  (application or Natural Access Server)
# 0 - Natural Access Server is responsible for the context name uniqueness.
#     Allow context name modification by Natural Access Server to ensure
#     name uniqueness (default)
# 1 - Application is responsible for the context name uniqueness.
#     Disallow context name modification by Natural Access Server,
#     uniqueness error expected from Natural Access Server
ContextNameUniqueness = 0

SessionTimeout = 3

#=========================================================================
# Natural Access Server configuration (server)
#
#=========================================================================
[server]
# ParmFlags = {0|1}   -  Natural Access Server parameter flags
# 0 - Natural Access Server provides the same default parameters for in-
#     process contexts and contexts created on the server.
# 1 - Natural Access Server provides different parameters for in-process
#     contexts and for contexts created on the server.
ParmFlags = 0

# TraceMode = {0|1|2}   -  Natural Access Server trace mode
```

```
# 0 - Tracing is disabled is default mode
# 1 - Enable tracing, immediate write to trace memory  (default mode)
#     (requires ctdaemon to be running)
# 2 - Write trace memory on error only, keep local trace memory for each
#     context
TraceMode = 1

 Remote tracing parameters
#   TraceMask = mask           - defines the default tracing mask
#   TraceFile = filename       - defines trace file name
#   StartCtaServer = {0|1}     - 0 disables starting of built-in
#                                Natural Access Server
TraceMask = 0
# TraceFile = cta.log
StartCtaServer   = 1

#
# DaemonAutoRestart parameter
#  0 - Do not automatically restart Natural Access Server after crash
#  1 - Automatically restart Natural Access Server after crash
#
DaemonAutoRestart = 0

# OAM Parameter
#  0 Configuration is not for an autonomous board
#  1 Assumes configuration is for autonomous board
AutonomousBoard = 0

#============================================================================
# Natural Access Default Parameter Changes (ctapar)
#
# Valid syntax for changes is:
#   service[.x].category.field = value
#
#============================================================================
[ctapar]

# by default, no changes are made

#============================================================================
[eof]
```

cta.cfg sections

The default *cta.cfg* specifies the following information:

Section	Description
ctasys	Sets the services available to the application. The Natural Access system header is interpreted by Natural Access Server (*ctdaemon*) and by **ctaInitialize**.
application	Sets the default server with which an application works. The default server is either in-process, local, or remote. See *Setting the default server* on page 34. This section also specifies whether or not Natural Access Server (*ctdaemon*) automatically renames contexts to ensure uniqueness. If this option is not activated, Natural Access Server returns an error when a context name is duplicated. The Session Timeout parameter defines the amount of time in seconds in which an application tries to establish communications between the application and a server.
server	Specifies how Natural Access Server (*ctdaemon*) runs. This section includes settings for Natural Access Server, parameter management, traceflags, and remote tracing. To run Natural Access on local and multiple remote servers, set StartCtaServer to 1. The ParmFlags and TraceMode settings affect only the Natural Access Server (*ctdaemon*). All contexts created on this server see the same parameter settings. TraceMode sets the style of tracing for applications. The TraceMode setting has no effect on contexts created on an in-process server. For contexts created on in-process servers, applications must define the style of tracing when they make the call to **ctaInitialize**. To define the style of tracing, set tracefields parameter within the CTA_INIT_PARMS structure. Applications operating with contexts created on servers in different locations (for example, in-process, local, and remote) have tracing characteristics that are a combination of the servers and applications tracing settings. If you set the DaemonAutoRestart parameter to 1, Natural Access Server (*ctdaemon*) starts automatically following some crashes.
ctapar	Specifies Natural Access default parameter overrides. This section is interpreted by Natural Access Server (*ctdaemon*) and by **ctaLoadParameterFile**.

Setting the default server

To set the default server, edit *cta.cfg* by setting the DefaultServer keyword in the application section to one of the following locations:

```
DefaultServer = inproc | localhost | server_name
```

Location	Description
inproc	The Natural Access Server functionality runs within the process address space. Contexts created using inproc are specific to the application and cannot be shared.
localhost	The Natural Access Server functionality runs in a Natural Access Server (*ctdaemon*) process that is separate from the application, but on the same host. Contexts created on Natural Access Server using localhost can be shared.
server_name	The Natural Access Server functionality runs in a Natural Access Server (*ctdaemon*) process that is separate from the application. A ***server_name*** or IP address specifies the host on which the server runs. The application may or may not be running on the same host. Contexts created on Natural Access Server using ***server_name*** can be shared.

Note: If an existing application uses a configuration file that sets the CTAmode statement, CTAmode = 0 or CTAmode = 2 are interpreted as if the DefaultServer statement is set to inproc. CTAmode = 1 is interpreted as a setting of localhost.

When you are finished, save the changes.

Setting up Natural Access applications

Complete the following steps to set up a Natural Access application:

Step	Action
1	Initialize the Natural Access application.
2	Create event queues.
3	Create contexts.
4	Open services on each context.

Setting up a second Natural Access application

Complete the following steps to set up a second Natural Access application that shares a context with the first application:

Step	Action
1	Initialize the Natural Access application.
2	Create event queues.
3	Attach to the existing context that the applications will share.

Initializing Natural Access applications

Use **ctaInitialize** to initialize Natural Access applications. When you initialize Natural Access applications, you

- Register the services that are available to the application.
- Specify the parameter management strategy.
- Enable or disable tracing.
- Enable or disable the wait object notification event when the internal wait object list has changed.

Creating event queues

Call **ctaCreateQueue** to create one or more event queues. Specify the service managers to attach to each queue (if this is different than the defaults specified in the *cta.cfg* file). When you attach or bind a service manager to a queue, you make the service manager available to the queue. After the services are opened, the events generated by the managed services go to the attached queue.

ctaCreateQueue returns an event queue handle (***ctaqueuehd***) that addresses the event queue. You supply the handle to retrieve events from the queue when calling **ctaWaitEvent**.

Applications can invoke **ctaCreateQueue** multiple times within a single process. Each invocation returns a unique queue handle.

After creating the event queue, Natural Access knows which services are available to the event queue (specified by the service managers). You can now create contexts on the event queue.

Creating contexts

To create contexts, call **ctaCreateContext** (or **ctaCreateContextEx** for shared contexts) and provide the queue handle (***ctaqueuehd***) that **ctaCreateQueue** returned. All events for services on the context are received in the specified event queue. **ctaCreateContext** returns a context handle (***ctahd***), which the application uses when invoking Natural Access functions. Events communicated back to the application are also associated with the context.

When creating a context, you can specify the following information:

- Context descriptor (***descriptor***). ***Descriptor*** defines each context by associating a context name and (optionally) the address of the server on which the context is created. To implement shared contexts, specify the context descriptor as a parameter when invoking **ctaCreateContext** or **ctaCreateContextEx**.

  If a server address is not specified in the context descriptor, the context is created on the default server as specified in *cta.cfg*.
- User-defined value (***userid***). Natural Access never interprets ***userid***. The value is echoed back to the application in the event structure when events are delivered. It can be used (instead of the ***ctahd***) to correlate the context with an application-specific data structure.
- The server on which to create the context. If you do not specify a server, the context is created on the default Natural Access Server.

Parameter management

Natural Access manages parameters for services on a context basis. The context maintains a copy of the parameters for all services managed by the dispatcher and specified in the call to **ctaInitialize**. Parameters can be modified for a service not yet opened by **ctaOpenServices** as long as the service is registered in the call to **ctaInitialize**. This feature gives applications the flexibility to open and close services without affecting context parameter values. Refer to *Opening services* on page 42 for more information.

Sharing contexts

Applications can share contexts. To create shared contexts, use **ctaCreateContextEx**. **ctaCreateContextEx** requires the same parameters as **ctaCreateContext** with an additional flag that sets the service sharing mode. The service sharing mode determines how processes attached to a common object can use the service instances opened on that context. When creating shared contexts, ensure that the context names are unique to the specified server.

When a client application destroys a shared context (with **ctaDestroyContext**), it relinquishes control of the context resources. However Natural Access waits until every application using the context invokes **ctaDestroyContext** before destroying the context.

This topic provides information about:

- Service sharing modes
- Creating persistent contexts
- Attaching to shared contexts

Service sharing modes

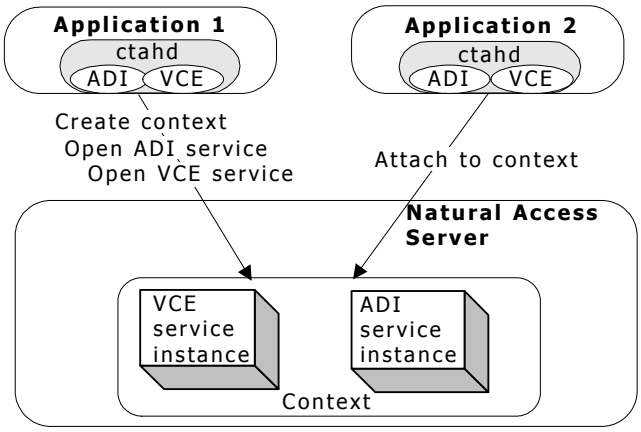
When two applications share a context, they can both use service instances opened on the context. However, depending on the sharing mode in which the context was created, the processes may or may not be able to immediately use functions and receive events from the open services. Natural Access provides two modes of service sharing:

- Common access
- Declared access

Common access

In common access mode (the default mode), service instances opened on a context by one application can be used immediately by any other application that attaches to the context. In common access mode, applications do not need to use **ctaOpenServices** to open service instances already active on the context. An application that closes a service on the context (with **ctaCloseServices**) closes the services for all applications that share the context.

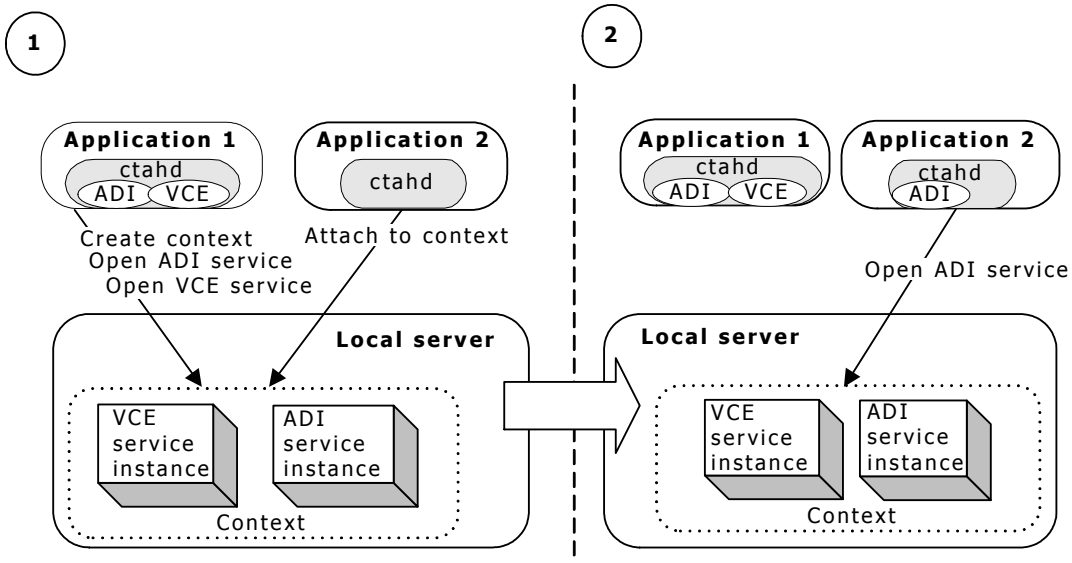
The following illustration shows two applications sharing a context created in common access mode. Application 1 creates the context and opens the ADI service and the Voice Messaging (VCE) service on the context. Application 2 attaches to the context and can immediately access the ADI service and VCE service instances opened on the context.



Declared access

In declared access mode, each application opens and closes its own service instances as needed. When an application attaches to the context, no services are available until the application opens them, even if other applications sharing the context have opened service instances. If the application closes a particular service instance, it is closed only for that application. Any other application that opened the service on the shared context can continue using it.

The following illustration shows two applications sharing a context created in declared access mode. Application 1 creates the context and opens the ADI service and Voice Messaging (VCE) service on the context. Application 2 attaches to the context and opens services on the context.



Comparing common access and declared access modes

Common access mode and declared access mode have the following similarities:

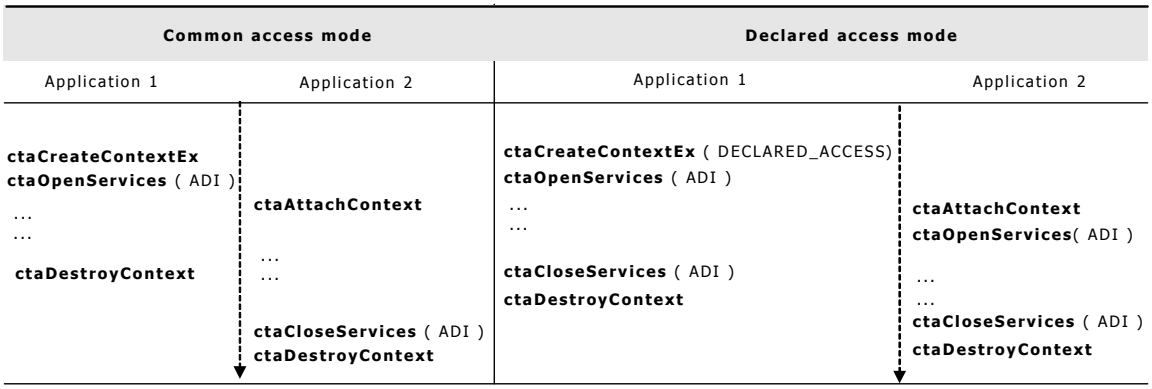
- Single instance of shared service.
- Events are broadcasted to all clients holding handles to the shared context.
- A context is destroyed on the local server only when the last remaining client invokes **ctaDestroyContext**.

The following table describes the differences between common access and declared access modes:

Common access mode	Declared access mode
When calling **ctaCloseServices** from one application, the service is not available to all remaining applications holding handles to the shared context.	When calling **ctaCloseServices** from one application, the service is still available to all remaining applications holding handles to the shared context.
Upon invocation of **ctaAttachContext**, all opened services are immediately available for use (services are inherited).	Upon invocation of **ctaAttachContext**, no services are available for immediate use. Must explicitly invoke **ctaOpenServices**.
Events from all inherited non-masked services are visible.	Only events from services that were explicitly opened are visible.

Examples

The following illustration shows the process for sharing services in common and declared access modes:



The following table describes the process for each mode:

Mode	Process
Common access	Application 1 creates a context. When Application 2 attaches to the context with **ctaAttachContext,** it does not need to explicitly open services already used by Application 1 (in this case, the ADI service). Since closing a service in common access mode closes it for all other applications that share the context, Application 1 uses **ctaDestroyContext** to relinquish control of the context, but does not invoke **ctaCloseServices**. The service remains available to Application 2.
Declared access	Application 1 creates the context. When Application 2 attaches to the context, it must invoke **ctaOpenServices** to use Natural Access services, including services already opened on the context (such as the ADI service). To relinquish control of the context, Application 1 uses **ctaCloseServices** and **ctaDestroyContext**. The ADI service still remains available to Application 2.

Note: In either mode, Natural Access does not actually destroy the context until both Application 1 and Application 2 destroy it.

You can decide the service-sharing mode based on the application's requirements. For example, if you use one application to set up and tear down calls (with the NCC service), and another to perform play and record functions (with the VCE service), create contexts in declared access mode. Each application deals exclusively with a particular Natural Access service and does not need to access other services.

Creating persistent contexts

When a Natural Access application creates a context, other applications can attach to the context. By default, if the originating application is disconnected after creating the context but before a second application attaches to the context, the server destroys the context. If the second application tries to attach to the context, the context does not exist. To enable the second application to attach to the context in this situation, the first application can set the CTA_CONTEXT_PERSISTENT flag in **ctaCreateContextEx**. The shared context continues to exist on the server even when the originating application is disconnected. Connected applications can reattach to the shared context.

When the last application attached to the context calls **ctaDestroyContext**, the context ceases to exist. Unless the last application calls **ctaDestroyContext**, the context remains active indefinitely.

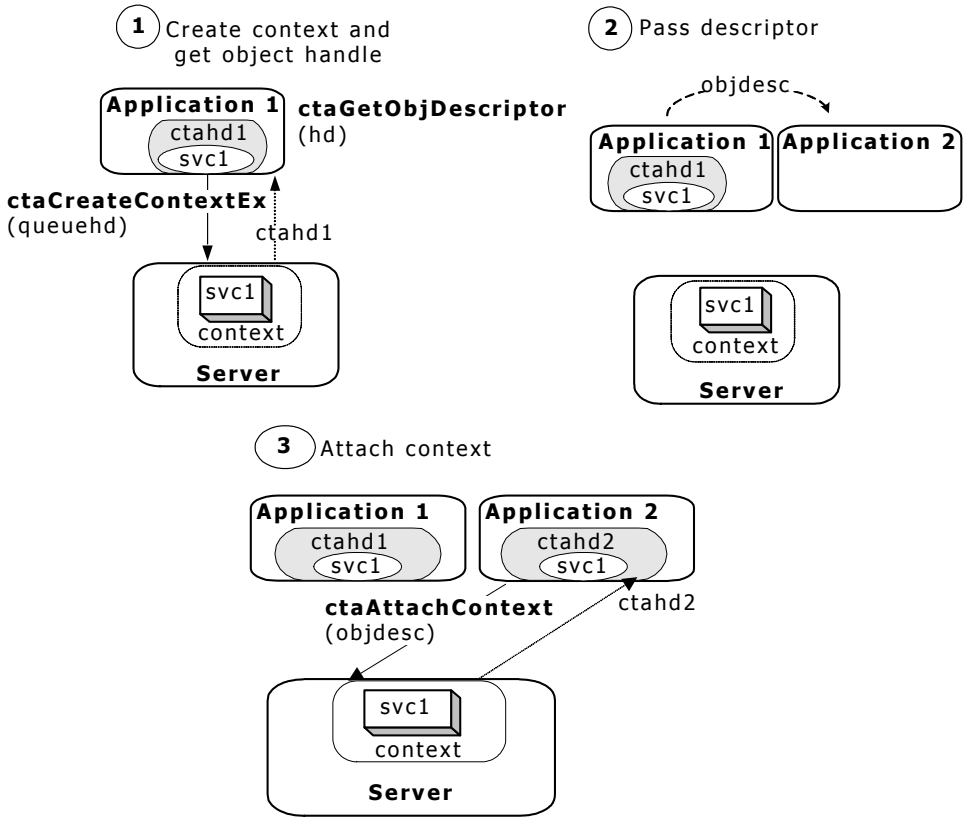
Attaching to shared contexts

Attach to existing contexts with **ctaAttachContext**. When invoking **ctaAttachContext,** specify a ***ctaqueuehd*** returned by **ctaCreateQueue**, and a Natural Access object descriptor. **ctaAttachContext** returns an application specific ***ctahd***.

The following process shows one application creating a context and a second application attaching to the context:

Stage	Description
1	The first application creates a context with **ctaCreateContextEx**.
2	The application retrieves the context-specific object descriptor by invoking **ctaGetObjDescriptor** with the ***ctahd*** returned by **ctaCreateContextEx**.
3	The originating application passes the object descriptor to a second application using one of the following methods: • Application events using the **ctaQueueEvent** function. (Refer to *Service sharing on a local server: csplayrc* on page 195.) Since events are broadcast to all applications that share the context, use the event of a well-known named context to pass the descriptor to all attached applications. • Private IPC channel. Set up a private communication channel to pass information between the applications. • File. Save the context descriptor in a file. Other applications can read the descriptor from the file when they share an existing context.
4	A second application attaches to the context by invoking **ctaAttachContext** and supplying the object descriptor and an event queue handle. **ctaAttachContext** returns an application specific ***ctahd.***

The following illustration shows the process of creating and attaching to a shared context:



Opening services

Applications open services on a context by calling **ctaOpenServices** and passing a context handle and a list of service descriptors. The service descriptor specifies the name of the service, the service manager, and service-specific arguments, for example, MVIP address.

The call to **ctaOpenServices** is asynchronous and returns immediately. When all services are open, CTAEVN_OPEN_SERVICES_DONE returns to the application.

Any application attached to a shared context can open service instances on the shared context. However, whether the applications that share the context can immediately use the service instance depends on the service-sharing mode specified with **ctaCreateContextEx**. See *Sharing contexts* on page 36.

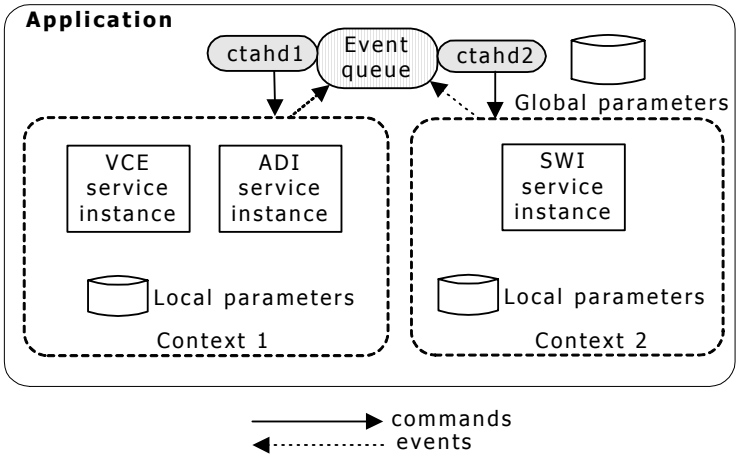
When applications attach to shared contexts in declared mode, they are not required to pass service-specific information to the call to **ctaOpenServices**. For example, the ADI service requires board, stream, and timeslot information when the context is created. Subsequent applications attaching to the context do not need to pass this information. If the information is present, it is ignored.

Opening a service on a context creates a service instance. A service instance holds the data that represents the specific use of the service, for example, the board and stream being used. The context is a bound collection of service instances that associates a server, services, and resources with a state, creates service dependencies, and handles parameter management.

From the context handle (***ctahd***), Natural Access Server (*ctdaemon*) can locate specific service instance information. Applications do not directly address the service; they call a service function with the context handle. For example, the command **vceOpenFile** (***ctahd***...) opens a voice file using the VCE service instance opened on the context.

Natural Access supports opening and closing services on an as-needed basis to optimize the use of scarce resources. Once the services are open, as shown in the following illustration, all paths in Natural Access are defined. You can now use the service API.

Note: Applications can use only the services specified on the server on which the context is created.



Sharing service objects

Once a service is open on a shared context, applications can share the service objects associated with the context. Each application determines how to share a service object depending on the service implementation.

To enable applications to share service objects, each application must attach to the same context, service instance, and service object. Applications can use one of the following methods to share service objects:

Method	Description
Natural Access function calls	Applications exchange object descriptors and then explicitly attach to the service object using a Natural Access function call.
Service-specific function calls	Applications use a service-specific function call to attach to the service object.
Events	Applications are automatically attached to the service object after receiving a service-specific event.

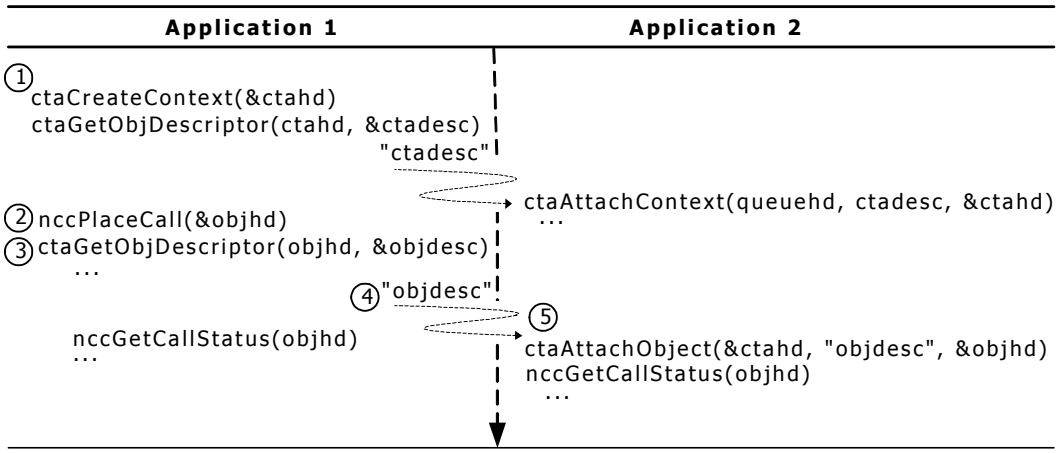
Using Natural Access function calls

Natural Access provides the function **ctaGetObjDescriptor** to retrieve the object descriptor associated with a specific service object. By using any standard transport mechanism, an application then transfers service object descriptors to all peer applications. When an application receives the service object descriptor, it attaches to the service object using the **ctaAttachObject** function call. This function is provided by Natural Access and is available to all services.

The following table describes how to use an object descriptor to attach to a service object.

Step	Action
1	The first application creates a context, and the second application attaches to the same context. Refer to *Attaching to shared contexts* on page 40 for more information.
2	The first application creates a service object on the shared context with a service function such as **nccPlaceCall**.
3	The application retrieves the object-specific descriptor by invoking **ctaGetObjDescriptor** with the object handle (***objhd***) returned by **nccPlaceCall**.
4	The originating application passes the object descriptor to the second application.
5	The second application invokes **ctaAttachObject** and supplies the object descriptor. **ctaAttachObject** returns a service-specific object handle (***objhd***) that is unique from the one given to the originating application. After obtaining the object handle, the second application can access the attached service object as can the first application, and both applications can share the service object.

Each of these steps is numbered in the following illustration:

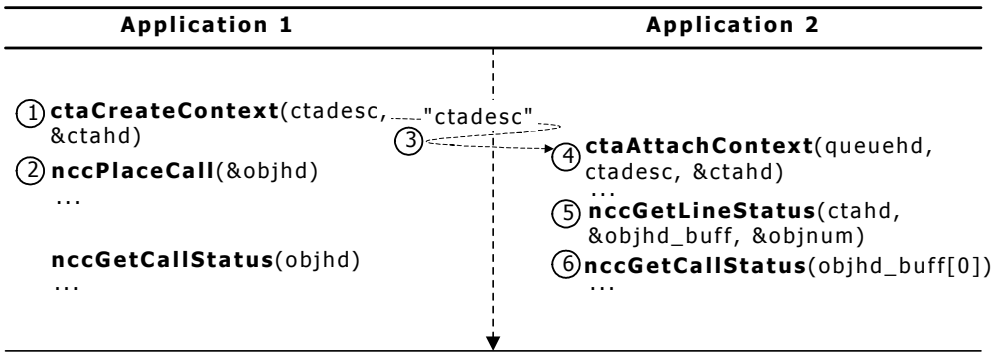


Using service-specific function calls

The following procedure describes how to attach to service objects by invoking service-specific functions:

Step	Action
1	The first application creates a context, and the second application attaches to the same context. Refer to *Attaching to shared contexts* on page 40 for more information.
2	The first application creates a service object on the shared context with a service function such as **nccPlaceCall**.
3	The second application provides a buffer to accept call handles while invoking a service-specific function such as **nccGetLineStatus**.

Each of these steps is numbered in the following illustration:



The Natural Call Control service supports this method. Refer to the *Natural Call Control Service Developer's Reference Manual* for more information.

Using events

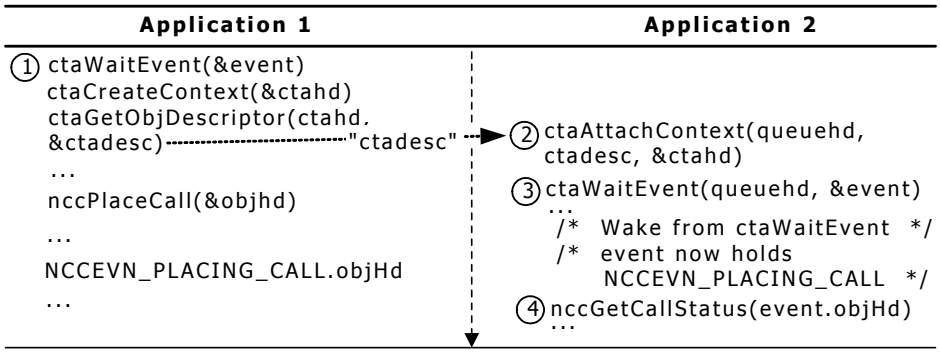
Since applications attached to the context receive all the events regardless of whether they are attached to the service object, an application can receive an unsolicited event containing a service object handle to which it is not currently attached. Upon receiving the event, the service object is automatically attached to the context. (Natural Access and some services provide this support.) Applications can take the object handle from the ***objHd*** field in CTA_EVENT to access the shared object.

The Natural Call Control (NCC) and Digital Trunk Monitor (DTM) services support this method. Refer to the *Natural Call Control Service Developer's Reference Manual* or *Digital Trunk Monitor Service Developer's Reference Manual* for more information.

The following table describes how an application is automatically attached to a service object:

Step	Action
1	The first application creates a context.
2	The second application attaches to the same context. Refer to *Attaching to shared contexts* on page 40 for more information.
3	The second application invokes **ctaWaitEvent** to receive events.
4	When an event arrives, the second application reads the service object handle from the ***objHd*** field in a service-specific event and uses the service object handle to access the attached service object. Both applications now share the same service object. Certain services require parameter settings. Refer to the service-specific manual for more information.

Each of these steps is numbered in the following illustration:



Receiving events on event queues

An event can contain a buffer of data or a scalar. If it contains a buffer, you must allocate memory to store the contents of the buffer. Memory management is typically done in one of the following ways:

- The application has pre-allocated the memory and **ctaWaitEvent** fills in that memory with the data. In this case, the application is responsible for both allocating and freeing its memory. An example is the use of **adiCollectDigits**.
- The application has not pre-allocated the memory. Natural Access or one of the Natural Access services allocated the memory on behalf of the application. The CTA_INTERNAL_BUFFER bit is set in the size field. In this case, the application is responsible for freeing the memory that it did not allocate.

Complete the following steps to process events and release allocated memory:

Step	Action
1	Call **ctaWaitEvent**.
2	On successful return, check to see if the CTA_INTERNAL_BUFFER bit is set.
3	If this bit is set, clear the bit before processing the event (so that the size field specifies the size of the buffer).
4	Upon the completion of the event processing, call **ctaFreeBuffer** if the CTA_INTERNAL_BUFFER bit was originally set.

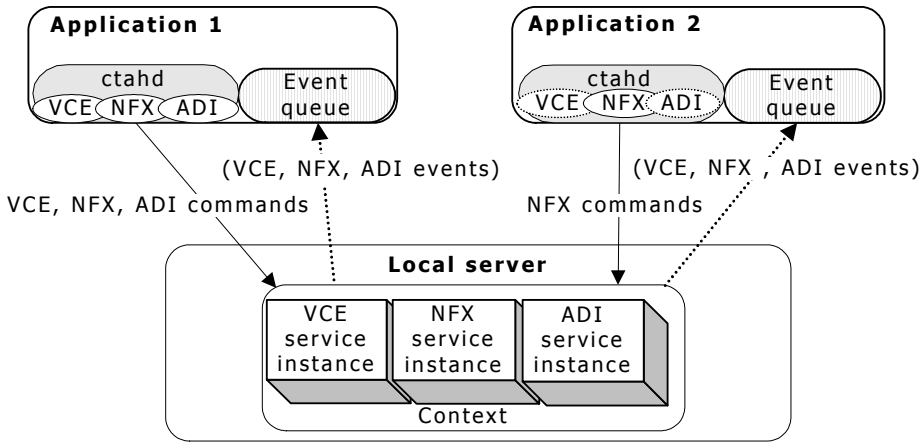
Note: Applications must call **ctaWaitEvent** in a timely manner to ensure prompt processing of time-critical events. Refer to *Using wait objects* on page 49.

Event processing for shared contexts

To share a context, each client application must create at least one event queue. Applications specify the event queue handle (***ctaqueuehd***) when attaching to an existing context.

Any application that attaches to the context immediately receives events for all of the services available on the context (whether or not the application uses functions from these services).

The following illustration shows two applications sharing a context that includes VCE, NFX, and ADI service instances. Each application uses a separate event queue for the events it receives. The illustration shows that, even though Application 2 invokes commands only from the NFX service, it receives events for all services available on the context.

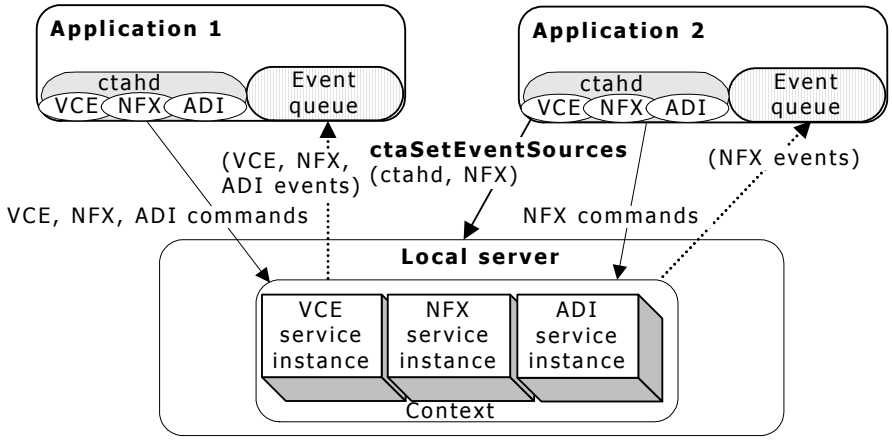


Note: When an application attaches to a context created in declared access mode, the application receives events only for services that it explicitly opens.

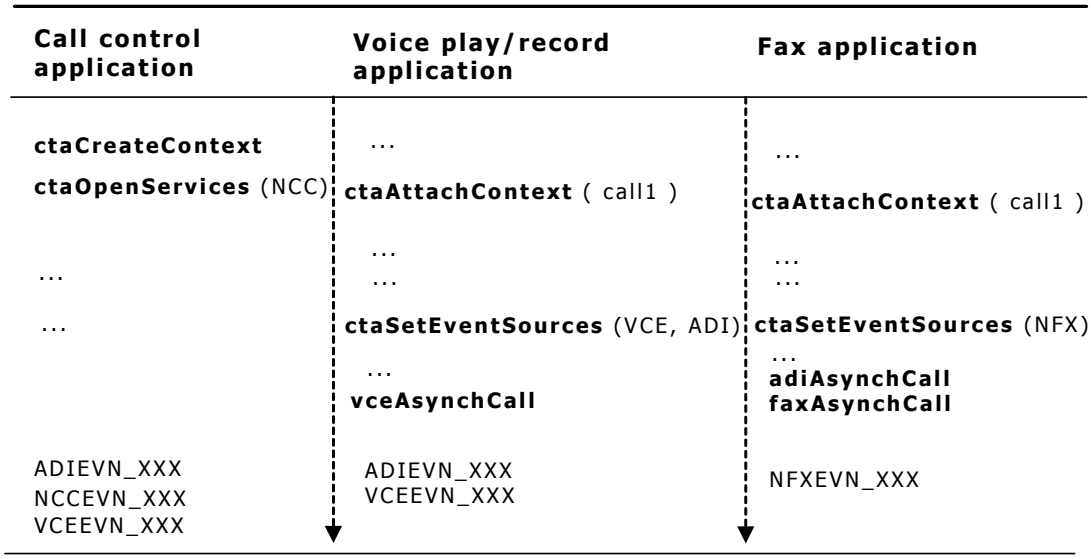
Screening shared service events

ctaSetEventSources specifies the services from which applications receive events. Applications attached to contexts created in common access mode can invoke **ctaSetEventSources** to limit the types of events they receive.

The following illustration shows a fax application attaching to a shared context where the VCE, NFX, and ADI services are open. Initially, the fax application receives events from all three open services. However, by invoking **ctaSetEventSources** (with the appropriate ***ctahd***) and specifying the NFX service, the application effectively screens out all events except those for the NFX service.



The following illustration shows an example of three applications screening events for a context that they all share (a context created in common access mode). The call control application receives events for the ADI service, as well as events for any other services opened by the two other applications. A voice play and record application uses **ctaSetEventSources** to screen NFX events so that it receives only ADI and VCE service events. A fax application uses **ctaSetEventSources** to screen all events except those from the NFX service.



This example illustrates that applications can receive any of the following types of events:

- All events from opened services, even from services that it did not explicitly open
- Only those events from inherited services and services that it explicitly opened
- Only those events that it explicitly opened

Use **ctaGetEventSources** to retrieve a list of services from which a specified context receives events.

Note: In common access mode, you can receive events from unexpected services between the time you invoke **ctaAttachContext** and **ctaSetEventSources**. Code your event handler to deal with these unwanted events.

Closing services

The Natural Access close functions parallel the Natural Access initialization functions:

Use this function...	To...
ctaDetachObject	Disconnect the application from a service object on a server.
ctaCloseServices	Close one or more services on a context. The function is asynchronous and returns immediately. You must wait for the completion event CTAEVN_CLOSE_SERVICES_DONE. If you are closing multiple services and more than one service close fails, the value field of the CTA_EVENT structure (**ctaWaitEvent**) contains the reason the first service failed to close.
ctaDestroyContext	Destroy a context. All currently open services are closed. The function is asynchronous and returns immediately. You must wait for the completion event, CTAEVN_DESTROY_CONTEXT_DONE, before the context is destroyed and resources are released. When using shared or persistent contexts, the context is not actually destroyed until the last application using it calls **ctaDestroyContext**.
ctaDestroyQueue	Destroy the event queue and all contexts associated with the event queue. This function is synchronous. The application remains blocked until all cleanup and close activity is complete. If the services and contexts are already closed, the function returns immediately.

Using wait objects

An asynchronous programming model requires applications to be able to wait for events from multiple sources without blocking on any particular one. Natural Access manages a list of wait objects internally for all of the devices that can generate events. Natural Access exposes two mechanisms for integrating application wait objects with the wait objects that Natural Access manages internally without requiring extra threads of execution.

The first mechanism gives Natural Access control to wait for application and Natural Access wait object signals. For example, an application can wait for a network event to arrive on a socket, input from a user through a keyboard or mouse, and an event from Natural Access. Natural Access allows the application to register the network socket and the keyboard and mouse objects with Natural Access so that the user application is called when there is an event on any of these.

The second mechanism enables the application to access the Natural Access internal wait objects so that the application can use operating system-specific wait functions to wait on the network socket and keyboard and mouse objects, as well as Natural Access internal objects.

Wait objects in Natural Access are operating system specific and are defined to be of the same basic type as the operating system requires in its wait functions.

In UNIX, the wait object is the pollfd structure that is passed to the **poll** system call. It contains a field for the file descriptor that is the same as the MUX_HANDLE type defined in *nmstypes.h*. It also contains extra fields that need to be specified in the call to **poll**.

In Windows, a wait object is equivalent to the MUX_HANDLE type defined in *nmstypes.h*. It is the same type as HANDLE and is passed to either **WaitForSingleObject** or **WaitForMultipleObjects**.

This topic presents:

- Wait objects and events managed by Natural Access
- Wait objects and events managed by the application

Wait objects and events managed by Natural Access

ctaRegisterWaitObject enables applications to add wait objects to the list of wait objects that Natural Access manages internally. This function requires the application to supply the wait object and a function to be called whenever the wait object is signaled by the operating system. The application must also specify a priority for the application events.

Priority	Description
CTA_PRIORITY_HIGH	The application's wait object are given a higher priority than the internal Natural Access wait objects.
CTA_PRIORITY_NORMAL	Natural Access wait objects are given higher priority than the application's wait objects.

After registering the application wait objects with Natural Access, the application calls **ctaWaitEvent**. Whenever the application wait object is signaled, Natural Access gives up the queue lock and calls the application specified callback function to handle the event. The user callback function can call any Natural Access function except **ctaWaitEvent** or **ctaDestroyQueue**. Refer to **ctaRegisterWaitObject** for more information.

The application can call **ctaUnregisterWaitObject** to unregister an application wait object from Natural Access. Natural Access ignores any events on the application wait object after it has been unregistered.

Wait objects and events managed by the application

Natural Access also provides a mechanism for applications to manage wait objects. On initialization, the application can specify a flag of CTA_NOTIFY_UPDATE_WAITOBJS to **ctaInitialize**. This flag tells Natural Access to notify the application through an event whenever Natural Access changes its list of internal wait objects. CTAEVN_UPDATE_WAITOBJS returns to the application whenever there is a change in the list of wait objects being managed by Natural Access. The application calls **ctaQueryWaitObjects** to get the list of wait objects that are used by Natural Access internally. The application subsequently calls the operating system dependent wait functions on their own wait objects as well as Natural Access wait objects returned by **ctaQueryWaitObjects**.

Whenever one of the Natural Access wait objects is signaled, the application detects it and calls **ctaWaitEvent** with a zero (0) timeout. Natural Access processes the event and returns control to the application. The application must ensure that **ctaWaitEvent** is called promptly after a Natural Access wait object is signaled.

Managing parameters

The characteristics for Natural Access services can be altered by modifying associated parameters. Parameters are grouped together in C language structures called parameter structures. Each parameter structure has default values that are sufficient for most configurations. The parameters can be modified to enable or disable features and to adapt the service for special configurations.

Natural Access knows how to store and retrieve the values of named parameters and structures for any service. All services providing the same functionality have the same standard parameters. Services providing the same functionality but implemented on different hardware can have unique extension parameters. In addition to Natural Access named parameters, most service functions allow the application to pass a data structure to override all the parameters.

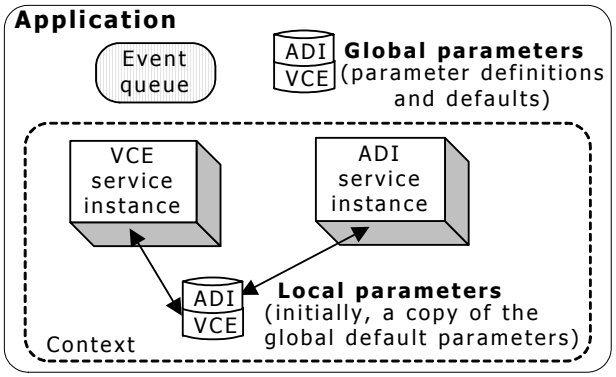
This topic describes:

- Global default parameters
- Shared service parameters
- Parameter definition
- Parameter functions
- Application-specific parameters

Global default parameters

When you initialize Natural Access using **ctaInitialize**, you create and store the service descriptors and values. These are the global default parameters. When you open a service on a context, the global default parameters are copied into the context. You can modify them to define context characteristics. You can also refresh the context parameters with the global default values as needed.

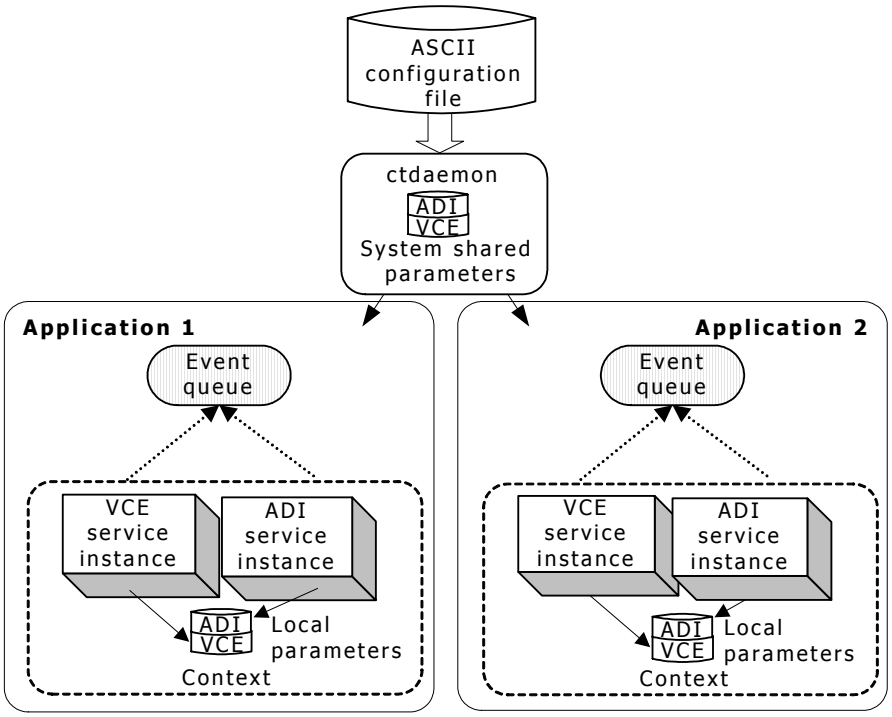
The following illustration shows the parameter scope:



As specified in **ctaInitialize**, Natural Access uses either process local memory or system shared memory to store the global default parameters. *ctdaemon* must be running for default parameters to be shared among processes using system shared memory. In this case, the parameter values can be modified by a configuration file.

ctdaemon is launched with the file name containing changes to the defaults. *ctdaemon* creates the shared memory with the parameters and modifies the defaults. Applications subsequently launched can use the shared memory as the global defaults.

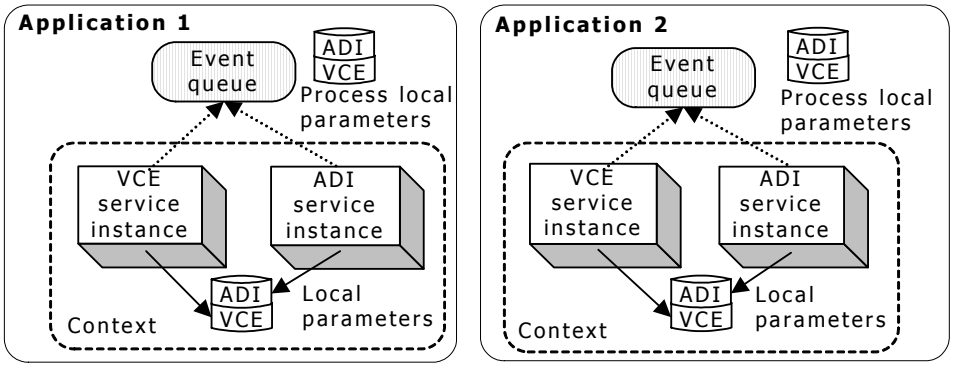
The following illustration show system shared default parameters:



When operating on a local server, parameters are always stored in the system shared memory.

If application local memory is used, changes to parameter values through *ctdaemon* have no effect on the application. You must use the Natural Access parameter access functions, for example, **ctaSetParmByName**, to modify values.

The following illustration shows the local default parameters:



Shared service parameters

When an application sets parameter values for a shared service instance, it changes these parameters for all applications that use the service. Therefore, parameters are essentially global for service instances on shared contexts. When client applications attach to shared contexts, they can use **ctaGetParmInfo** or **ctaGetParmInfoEx** to retrieve parameter settings for open service instances on the context.

Parameter definition

A Natural Access parameter is composed of the following elements:

Element	Description
ASCII name	Parameter name. For example, ADI.COLLECT.intertimeout. See *Parameter naming convention* on page 53.
Data type	You can define a Natural Access parameter as either a scalar data type or a string. For example, the ADI.RECORD.AGCattacktime parameter is a DWORD.
Unit	Unit associated with the parameter, such as dB or dBm. Units are primarily used for display purposes.
Value	Value associated with the parameter that can be set and retrieved. A parameter is initialized with a default value.

Parameter naming convention

Natural Access parameters are named and used according to the following syntax:

- ***SvcName.category.fieldName***
- ***SvcName.category.subStructure.fieldName***

Where:

Name	Description
SvcName	Indicates the service to which the parameter belongs.
Category	Allows multiple, logically related ***fieldNames*** to be grouped.
SubStructure	Allows multiple, logically related ***fieldNames*** to be nested under a category.
FieldName	Indicates the actual parameter.

For example, the ADI service contains multiple categories, each of which contain multiple ***fieldNames*** such as:

- ADI.CALLPROG.busycount
- ADI.CALLPROG.leakagetime
- ADI.CALLPROG.maxbusy
- ADI.COLLECT.firsttimeout
- ADI.COLLECT.intertimeout

Parameter functions

The following table lists the parameter functions. If a valid context handle (***ctahd***) is passed to the function, the context copy of the parameters is retrieved or modified. If the context handle is a void context handle, the global default values are retrieved or modified on the server specified by the void context handle. If the context passed is NULL_CTAHD, the global default values are retrieved or modified on the default server.

To...	Use...
Modify a single field for a given parameter name.	**ctaSetParmByName**
Modify a single field for a given parameter name on a specified Natural Access Server.	**ctaSetParmByNameEx**
Retrieve a single field for a given parameter name.	**ctaGetParmByName**
Retrieve a single field for a given parameter name on a specified Natural Access Server.	**ctaGetParmByNameEx**
Retrieve a parameter field definition.	**ctaGetParmInfo**
Retrieve a parameter field definition on a specified Natural Access Server.	**ctaGetParmInfoEx**
Retrieve parameter values for a given parameter structure.	**ctaGetParms**
Retrieve parameter values for a given parameter structure on a specified Natural Access Server.	**ctaGetParmsEx**
Reset values of all parameters on a context to the global defaults.	**ctaRefreshParms**
Retrieve the parameter ID for a given parameter name.	**ctaGetParmID**
Retrieve the parameter IDs and service ID for the named service and service manager.	**ctaGetParmIds**
Retrieve the parameter IDs and service ID for the named service and service manager on a specified Natural Access Server.	**ctaGetParmIdsEx**
Load and change parameter default values.	**ctaLoadParameterFile**

The Natural Access parameter functions allow you to obtain the parameter information either by specifying the parameter ID, for example, with **ctaGetParmID**, or by specifying the parameter name (for example, with **ctaGetParmByName**). The retrieval process is faster when you specify the parameter ID.

The *ctaparm* utility displays system parameter defaults. Refer to *Natural Access parameter display: ctaparm* on page 236 for more information.

ctdaemon allows you to modify system global default parameters and set the global tracemask. Refer to *Natural Access Server: ctdaemon* on page 238 for more information.

Application-specific parameters

The Parameter Management service (PRM) enables you to manipulate application-defined parameters in the same manner as other Natural Access service parameters. The application parameters are defined in ASCII text files. Use the *pfencode* utility to encode them as binary parameter files. The text or binary (encoded) parameter files are stored in a directory contained in the list of directories specified by the CTA_DPATH environment variable. Load the parameters by specifying the PRM service in the list of services passed to **ctaInitialize**.

Refer to *Defining and using application-specific parameters* on page 265 for more information.

Handling Natural Access errors

All Natural Access functions return one of the following codes:

- SUCCESS (0), indicating that either the function completed (synchronous functions) or that the function was initiated (asynchronous functions).
- An error code indicating that the function failed. Asynchronous functions that return a non-zero value are 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 and the event value field contains an error code.

  Error codes can also appear in the value field of a DONE event. Use the CTA_IS_ERROR macro to determine if a value is an error. The CTA_IS_ERROR macro is applicable only to DONE events.

Each service defines error codes in its header file. Natural Access error codes are defined in the *ctaerr.h* header file and are prefixed with CTAERR_. All services providing the same functionality provide consistent error codes. Use **ctaGetText** to obtain the text representation of an error code.

For a description of the Natural Access error codes, see the *Alphabetical error summary* on page 255.

This topic presents:

- Trapping errors
- Error tracing
- Server tracing

Trapping errors

Use **ctaSetErrorHandler** to define the action taken when an error is returned from any Natural Access functions, including any Natural Access service functions. If Natural Access encounters an error when executing a function, it performs the error action before returning to the application. The error action is undertaken only when Natural Access functions are invoked; it is not undertaken if an error is returned through an event.

By default, there is no error action. You can specify the following error actions:

- Print an error message to stdout and return to the application.
- Print an error message and exit Natural Access.
- Call a user-defined error handler.

Error tracing

Enable or disable tracing in the call to **ctaInitialize**. Services and the application can both log trace messages.

Enable service tracing by calling **ctaSetTraceLevel** and setting a flag to indicate what to trace (for example, events or commands). Based on the flags set, the service logs internal trace information to the *ctdaemon*.

Enable application tracing by calling **ctaLogTrace** and passing a trace severity, a format, and the trace value.

ctdaemon must be running to enable tracing. Using *ctdaemon*, you can specify to send trace messages to a file, to a console in interactive mode, or to a remote display. Refer to *Natural Access Server: ctdaemon* on page 238 for more information.

Backtracing on errors

Natural Access provides a tracing option that allows you to receive a backlog of trace information on the context on which the error occurred.

To enable backtrace on error, change the traceflags field in the CTA_INIT_PARMS structure that is passed to **ctaInitialize**.

Enable this feature if you want to review command and event data leading up to errors. Do not enable this feature if you want to observe trace information for all contexts, whether or not you have errors.

To write all recently logged trace messages on a particular context to the Natural Access daemon, call **ctaLogTrace**, specifying the context you want to analyze, and set the ***traceseverity*** argument to CTA_TRACE_SEVERITY_ERROR.

Server tracing

Applications can use the following functions to start and stop the *rtrace* trace utility, set the global trace mask, and receive trace messages from Natural Access Server.

To...	Use...
Request trace messages from a server	**ctaStartTrace**
Stop receiving trace message from a server	**ctaStopTrace**
Change the server's global trace mask	**ctaSetGlobalTraceMask**

Verifying compatibility

Because Natural Access and initialized Natural Access services are dynamically linked, the application needs a way to determine if it is using a compatible library or server. The compatibility level is incremented when any release includes changes that require applications to be recompiled.

To...	Use...
Check the installed Natural Access core version	**ctaGetVersion** or **ctaGetVersionEx**
Check the initialized service version	**ctaGetServiceVersion** or **ctaGetServiceVersionEx**

ctaGetVersion, ctaGetVersionEx, ctaGetServiceVersion, and **ctaGetServiceVersionEx** return the version, the build date, and the compatibility level of Natural Access library, server, or service.

The *ctavers* utility displays version information about Natural Access services. The version information can be used to report to NMS which revision and build date you are using. Refer to *Service version confirmation: ctavers* on page 237 for more information.

Monitoring server status

Applications receive the server communication error CTAERR_SVR_COMM when the server fails to respond. Any command issued on the local or remote server can return CTAERR_SVR_COMM.

To monitor the continuing status of the server, applications can invoke **ctaWaitEvent** during periods of inactivity. This function enables applications that go to sleep on particular threads to monitor the status of the server. For more information about using **ctaWaitEvent**, refer to *Using wait objects* on page 49.

Controlling the server remotely

ctaShutdown enables applications to shut down and reboot Natural Access Server (*ctdaemon*) running on a remote server, as well as shut down and reboot the host server system. This function requires the application to specify the server name and the action to be taken. For more information, see **ctaShutdown**.

6 Development environment

Header files

Use the following header files to develop applications:

File	Description
ctadef.h	Natural Access library definition.
ctadisp.h	Natural Access dispatcher include file.
ctaerr.h	Natural Access error definitions.
nmstypes.h	Primitive type definitions that promote portability.
nmshw.h	NMS board definitions.
servicename*def.h*	Service API definition files.
servicename*spi.h*	Service SPI include file.

For information on other service-specific include files, see the developer's reference manual for the service.

Library files

The following library files are shipped with Natural Access:

Windows file	UNIX file	Description
cta.lib	*libcta.so*	Natural Access library to which applications link. Includes other Natural Access core DLLs (*cta*.lib*).
servicename*mgr.lib*	***servicename****mgr.so*	Service manager library.
servicename*api.lib*	***servicename****api.so*	Service API library.

Compiling under Windows

This topic describes how to set environment variables and how to compile Natural Access applications in the Windows operating system environment. It includes information about:

- Compiling
- Application linking
- Utility binaries
- Configuration files
- Prompts and voice files

The software is installed in the *\nms* directory.

Compiling

When compiling an application, direct the compiler to *\nms\include* for the Natural Access header files.

Define WIN32 on the compilation command line as follows:

```
-Ic:\nms\include -DWIN32
```

You can add *\nms\include* to your Include environment variable.

Application linking

When linking for Windows, specify the path where the Natural Access import libraries (for example, *cta.lib*, *vcemgr.lib*) reside. The library files are in the *\nms\lib* directory. You can specify the path from the command line or add *\nms\lib* to your Lib environment variable.

When executing Natural Access applications, specify the path where the service manager dynamic link libraries (DLLs) reside. The DLLs are in the *\nms\bin* directory. You can specify the path from the command line or add *\nms\bin* to your Path environment variable by using the Control Panel system dialog box.

Refer to the *demonstration programs* on page 193 for examples of compiling and linking applications.

Utility binaries

Utility binaries are stored in the *\nms\bin* directory. For more information about using utility binaries, refer to *PRM service text parameter file encoder: pfencode* on page 250.

Binaries of demonstration programs are located in the respective subdirectories in *\nms\ctaccess\demos*.

Configuration files

The *nms**ctaccess**cfg* directory contains a sample configuration file (*cta.cfg*) used with the *ctavers* utility and with *ctdaemon*. Refer to the following topics for more information:

- *Natural Access configuration file* on page 31
- *Service version confirmation: ctavers* on page 237
- *Natural Access Server: ctdaemon* on page 238

Prompts and voice files

The *nms**ctaccess**prompts* directory contains prompt tables and voice files used by the demonstration programs.

When executing Natural Access applications, voice files can be found by the path specified by the CTA_DPATH environment variable.

Compiling under UNIX

This topic describes how to set environment variables and how to compile Natural Access applications in the UNIX operating system environment. It includes information about:

- Compiling
- Application linking
- Example compile and link
- Utility binaries
- Configuration files

Compiling

When compiling an application, direct the compiler to */opt/nms/include* for the Natural Access header files. These header files contain conditional compilation for different Natural Access target operating systems.

Define UNIX, UNIX_SVR4, and SOLARIS on the compilation command line, as follows:

```
-I/opt/nms/include -DUNIX -DUNIX_SVR4 -DSOLARIS
```

Application linking

When linking for UNIX, link the Natural Access shared libraries (for example, *libvcemgr.so* or *libcta.so*) and the system dynamic linker, *libdl.so*, with the application program.

Specify the path where the Natural Access shared libraries reside. You can specify the path from the link line with a -L */opt/nms/lib* command or by setting the LD_LIBRARY_PATH to */opt/nms/lib*.

Example compile and link

The following example shows the compiling and linking of *myprog.c*:

```
cc -I/opt/nms/include -DUNIX -DUNIX_SVR4 -DSOLARIS -L/opt/nms/lib -lcta -lvcemgr
-lswimgr -ladimgr -ldl myproc.c
```

Note: When building a multi-threaded application, you must specify the -lthread flag when compiling and linking. This flag generates and links code for multi-threaded applications.

Utility binaries

Utility binaries are stored in */opt/nms/bin*. Details concerning their use are described in *Summary of the utilities* on page 235.

Binaries of demonstration programs are located in the respective subdirectories in */opt/nms/ctaccess/demos*.

Configuration files

The */opt/nms/ctaccess/cfg* directory contains a sample configuration file (*cta.cfg*) used with the *ctavers* utility and with *ctdaemon*. Refer to *Service version confirmation: ctavers* on page 237 and Natural Access Server: ctdaemon for more information. Refer to *Natural Access configuration file* on page 31 for more information about *cta.cfg*.

7 Function summary

Application setup functions

Use the following functions to set up your application:

Function	Synchronous/ Asynchronous	Description
ctaSetDefaultServer	Synchronous	Sets the default server for a Natural Access application.
ctaInitialize	Synchronous	Establishes a list of available services for a Natural Access application.
ctaCreateQueue	Synchronous	Creates an event queue and returns a queue handle (***ctaqueuehandle***).
ctaCreateContext	Synchronous	Creates a context and returns a context handle (***ctahd***).
ctaCreateContextEx	Synchronous	Creates a context or a persistent context, returns a context handle (***ctahd)***, and sets the context sharing mode on local and remote servers.
ctaOpenServices	Asynchronous	Opens one or more services on the specified context (***ctahd***).
ctaAttachContext	Synchronous	Enables applications to use contexts created by other applications on local and remote servers.
ctaGetContextInfo	Synchronous	Returns basic context information specified by the ***ctahd*** parameter.
ctaGetContextInfoEx	Synchronous	Returns extended context information including a list of services opened on the context from a particular Natural Access Server.
ctaGetObjDescriptor	Synchronous	Returns the object descriptor associated with a particular context or service object.
ctaAttachObject	Synchronous	Enables applications to use service objects created by other applications on local and remote servers.

Refer to *Setting up Natural Access applications* on page 35 for more information.

Event processing functions

Use the following functions to access the event queue:

Function	Synchronous/ Asynchronous	Description
ctaQueueEvent	Synchronous	Enqueues an event to an event queue.
ctaWaitEvent	Synchronous	Retrieves an event from the specified queue.
ctaSetEventSources	Synchronous	Specifies the services from which an event queue receives events (local and remote servers only).
ctaGetEventSources	Synchronous	Retrieves a list of services from which the application receives events (local and remote servers only).
ctaAllocBuffer	Synchronous	Allocates memory for data transport with **ctaQueueEvent**.
ctaFreeBuffer	Synchronous	Frees internally allocated buffers or buffers allocated with **ctaAllocBuffer**.

Refer to *Receiving events on event queues* on page 46 for more information.

Shut down functions

Use the following functions to shut down:

Function	Synchronous/ Asynchronous	Description
ctaDetachObject	Synchronous	Enables applications to disconnect from service objects created by other applications on local and remote servers.
ctaCloseServices	Asynchronous	Closes one or more services on a context.
ctaDestroyContext	Asynchronous	Destroys the context and closes all currently opened services. Clears CTA_PERSISTENT_CONTEXT, if set, when the last application attached to the context calls this function.
ctaDestroyQueue	Synchronous	Destroys all contexts created on the event queue, closes all currently opened services, and destroys the event queue.
ctaShutdown	Synchronous	Shuts down and restarts the Natural Access Server (*ctdaemon*), as well as shuts down and reboots the server host system.

Refer to *Closing services* on page 49 for more information.

Parameter modification functions

Use the following functions to obtain or change parameter information:

Function	Synchronous/ Asynchronous	Description
ctaGetParmByName	Synchronous	Retrieves a single field for a given parameter name.
ctaGetParmByNameEx	Synchronous	Retrieves a single field for a given parameter name from a particular Natural Access Server.
ctaSetParmByName	Synchronous	Modifies a single field for a given parameter name.
ctaSetParmByNameEx	Synchronous	Modifies a single field for a given parameter name from a particular Natural Access Server.
ctaGetParmID	Synchronous	Retrieves the parameter category ID for a given parameter name.
ctaGetParmIds	Synchronous	Returns the parameter IDs and service ID for the named service and service manager.
ctaGetParmIdsEx	Synchronous	Returns the parameter IDs and service ID for the named service and service manager that are available on a particular Natural Access Server.
ctaGetParmInfo	Synchronous	Returns parameter values for a given parameter structure.
ctaGetParmInfoEx	Synchronous	Returns parameter values for a given parameter structure that are available on a particular Natural Access Server.
ctaGetParms	Synchronous	Returns parameter values for a given parameter structure.
ctaGetParmsEx	Synchronous	Returns parameter values for a specified parameter structure from a particular Natural Access Server.
ctaLoadParameterFile	Synchronous	Loads and changes parameter default values.
ctaRefreshParms	Synchronous	Resets the values of all context parameters on a context to the global defaults.

Refer to *Managing parameters* on page 51 for more information.

Wait object functions

Use the following functions to establish whether Natural Access or your application manages wait objects:

Function	Synchronous/ Asynchronous	Description
ctaQueryWaitObjects	Synchronous	Obtains an operating system specific array of wait objects that Natural Access is managing internally. This function is used when the application manages wait objects.
ctaRegisterWaitObject	Synchronous	Registers a wait object with Natural Access. This function is used when Natural Access manages wait objects.
ctaUnregisterWaitObject	Synchronous	Unregisters a previously registered wait object. This function is used when Natural Access manages wait objects.

Refer to *Using wait objects* on page 49 for more information.

Error handling functions

Use the following functions to manage errors and other types of codes in Natural Access:

Function	Synchronous/ Asynchronous	Description
ctaGetText	Synchronous	Returns the text representation of an error, reason, event, or command code.
ctaGetTextEx	Synchronous	Returns the text representation of an error, reason, event, or command code from a particular Natural Access Server.
ctaSetErrorHandler	Synchronous	Defines the action taken when an error is returned from Natural Access services.
ctaLogTrace	Synchronous	Allows you to log errors, warnings, or informational messages to the Natural Access trace log.
ctaSetTraceLevel	Synchronous	Sets the trace mask for a service on the specified context.
ctaStartTrace	Synchronous	Starts tracing on a remote server.
ctaSetGlobalTraceMask	Synchronous	Sets the global trace mask for the Natural Access Server specified by a context handle (***ctahd***).
ctaStopTrace	Synchronous	Stops tracing on a remote server.

Refer to *Handling Natural Access errors* on page 55 for more information. Refer to the *Alphabetical error summary* on page 255 for a list of Natural Access errors and to the *Event summary* on page 263 for a list of Natural Access events.

Compatibility verification functions

Use the following functions to return the version, build date, and compatibility level of Natural Access or of a Natural Access service:

Function	Synchronous/ Asynchronous	Description
ctaGetVersion	Synchronous	Retrieves the installed Natural Access version.
ctaGetVersionEx	Synchronous	Retrieves the installed Natural Access version from a specified server.
ctaGetServiceVersion	Synchronous	Retrieves the initialized service version.
ctaGetServiceVersionEx	Synchronous	Retrieves the initialized service version from a specified server.

Refer to *Verifying compatibility* on page 57 for more information.

Miscellaneous functions

Function	Synchronous/ Asynchronous	Description
ctaFindFile	Synchronous	Finds a file using the specified default extension and environment variable.
ctaFindFileEx	Synchronous	Finds a file on a server using the specified default extension and environment variable.
ctaFormatEvent	Synchronous	Formats an event into a string for print diagnostics.
ctaFormatEventEx	Synchronous	Formats an event on a specified server into a string for print diagnostics.
ctaGetTimeStamp	Synchronous	Converts an event timestamp to a count of the number of seconds elapsed since 00:00:00 January 1, 1970.
ctaGetQueueHandle	Synchronous	Obtains the queue handle of the specified context.
ctaQueryServerContexts	Synchronous	Obtains a list of available contexts on the specified server.
ctaQueryServices	Synchronous	Obtains a list of available services on the specified server.

8 Function reference

Using the function reference

This section provides an alphabetical reference to the system functions used by all services in Natural Access. A typical function includes:

Prototype	The prototype is followed by a list of the function arguments. NMS 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.
Return values	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 *Alphabetical error summary* on page 255 and the *Numerical error summary* on page 261 for a list of errors returned by Natural Access functions.
Events	If events are listed, the function is asynchronous and is complete when the DONE event is returned. If there are no events listed, the function is synchronous. Additional information such as reason codes and return values may be provided in the value field of the event.
Example	Example functions that start with Demo are excerpts taken from the demonstration code that is shipped with the product. Example functions that start with my are excerpts taken from sample application programs. The notation /* ... */ indicates additional code that is not shown.

ctaAllocBuffer

Allocates memory for data transport with **ctaQueueEvent**.

Prototype

DWORD **ctaAllocBuffer** (void ****buffer***, unsigned ***size***)

Argument	Description
buffer	Address of the pointer to the returned buffer.
size	Requested size of the buffer to allocate.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid parameter, NULL ***buffer*** pointer, or 0 ***size***.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_OUT_OF_MEMORY	Memory allocation error.

Details

ctaAllocBuffer allocates memory inside Natural Access. The Natural Access memory allocation mechanism can be used to send data in an event.

If an application sends an event with a shared context handle, the event is received by all context clients (applications or threads). The sending application receives an event with the buffer allocated by the application and no flag in the ***size*** field. To free the buffer, the application should use **ctaFreeBuffer**.

All other context clients receive an event with the Natural Access internal buffer in the event and CTA_INTERNAL_BUFFER in the ***size*** field. The clients should use **ctaFreeBuffer** to free memory allocated with **ctaAllocBuffer.**

Example

```
#define APPEVN_MESSAGE CTA_USER_EVENT(0x1)
int DemoSendMessage(CTAHD ctahd, unsigned mode, char* strMessage)
{
     int       ret;
     CTA_EVENT event;

     memset( &event, 0, sizeof(event) );

     event.id = APPEVN_MESSAGE;
     event.ctahd = ctahd;
     event.value = mode;
     event.size = ( strMessage ) ? strlen(strMessage ) : 0;
     if( event.size != 0 )
     {
         event.size += 1;
         ret = ctaAllocBuffer( &event.buffer,event.size);
         if( ret != SUCCESS )
             return ret;
         strcpy( (char*)event.buffer,strMessage );
     }

     return ctaQueueEvent( &event );
}
```

ctaAttachContext

Allows an application to attach to an existing context.

Prototype

DWORD **ctaAttachContext** (CTAQUEUEHD ***ctaqueuehd***, unsigned ***userid***, char ***descriptor***, CTAHD ***ctahd***)

Argument	Description
ctaqueuehd	Event queue handle returned by **ctaCreateQueue**.
userid	User-specified value repeated in all events returned for this context.
descriptor	Pointer to a context descriptor created by or obtained from another application.
ctahd	Pointer to a returned context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Supplied context name is unavailable.
CTAERR_INCOMPATIBLE_SERVICES	On the server on which the context is created, at least one of the available services conflicts with the same service that is open on another server. A client application can use only one instance of a service at a time.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaAttachContext enables applications using local or remote instances of Natural Access Server (*ctdaemon*) to share contexts. The application creating the context must either specify the descriptor with **ctaCreateContextEx** or obtain the descriptor with **ctaGetObjDescriptor**. The application passes the descriptor to another application, which can then use **ctaAttachContext** and the descriptor to attach to the context.

The attaching application can use the returned ***ctahd*** to open services on the specified context, or invoke functions for service instances already opened on the context. Whether the attaching application can immediately use the service instances opened on the context is determined by the context access mode set with **ctaCreateContextEx**.

For more information, see *Sharing contexts* on page 36.

See also

ctaOpenServices, **ctaSetEventSources**

Example

```
typedef struct
{
    unsigned   line;
    unsigned   ag_board;
    unsigned   mvip_stream;
    unsigned   mvip_slot;
    CTAQUEUEHD ctaqueuehd;
    CTAHD      ctahd;
    VCEHD      demovh;
    SWIHD      demoswihd;
} DEMOCONTEXT ;
/*
 * Attach a Natural Access context with its own queue, storing handles in
 * demo context (cx) which was created by the caller.
 */
void DemoAttachContext(unsigned line, DEMOCONTEXT *cx)
{
    char cxname[12];
    int ret;

    cx->line = line;

    /* Context name will be printed in all trace records for this
     * context
     */
    sprintf(cxname, "DEMOCX%04d", cx->line);

    /* Create queue and attach all service managers */
    ctaCreateQueue( NULL, 0, &(cx->ctaqueuehd) );

    /* Attach context with address of demo context as userid */
    ret = ctaAttachContext( NULL, cx->ctaqueuehd, (unsigned) cx,
                      cxname, &(cx->ctahd) );
    if (SUCCESS != ret)
         printf("Unable to attach context\n");
}
```

ctaAttachObject

Attaches to an existing service object.

Prototype

DWORD **ctaAttachObject** (CTAHD ***ctahd**, char ***descriptor**, unsigned ***objhd**)

Argument	Description
ctahd	Pointer to a context handle.
descriptor	Pointer to a service object descriptor created by or obtained from another application.
objhd	Pointer to a returned object handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid object descriptor.
CTAERR_NOT_FOUND	Invalid context.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaAttachObject enables applications using local or remote instances of Natural Access Server (*ctdaemon*) to share service objects. To use this function, the applications must share the context that is associated with the service object.

The application creating the service object must either specify the descriptor or obtain the descriptor with **ctaGetObjDescriptor**. The application passes the descriptor to another application. The second application can then use **ctaAttachObject** and the descriptor to attach to the service object. When **ctaAttachObject** returns, ***objhd*** contains an object handle and ***ctahd*** contains the context handle of the context that is associated with the attached service object. Subsequently, the application can use the objhd to access the service object.

For more information, see *Sharing contexts* on page 36.

See also

ctaDetachObject

Example

```
int DemoAttachDetach(char* contextname, char* objname)
{
     DWORD ret;

        CTAQUEUEHD ctaqueuehd;
        char *mgrlist[] = { "ADIMGR", "ADIMGR", "VCEMGR", "SWIMGR"};
        ret = ctaCreateQueue( mgrlist,
            sizeof(mgrlist)/sizeof(mgrlist[0]),
            &ctaqueuehd);
        if ( ret != SUCCESS )
            return ret;
```

```
        // Attach to existing server context
        CTAHD ctahd;
        ret = ctaAttachContext(ctaqueuehd,0,contextname,&ctahd);
        if ( ret != SUCCESS )
            return ret;

        // Attach to existing call
        unsigned callhd;
        ret = ctaAttachObject(&ctahd,objname,&callhd);
        if ( ret != SUCCESS )
            return ret;

        printf("Attached to call 0x%08X\n",callhd);

        // Get call status
        NCC_CALL_STATUS info;
        ret = nccGetCallStatus( callhd, &info, sizeof( NCC_CALL_STATUS ) );
        if ( ret != SUCCESS )
            return ret;

        printf("Called number: %s\n",info.calledaddr );

        // Detach from the call
        ret = ctaDetachObject(callhd);
        if ( ret != SUCCESS )
            return ret;

        printf("Detached\n");

        return SUCCESS;
}
```

ctaCloseServices

Closes one or more services.

Prototype

DWORD **ctaCloseServices** (CTAHD ***ctahd***, char ****svcname[]***, unsigned ***nsvcs***)

Argument	Description
ctahd	Context handle.
svcname	Pointer to the names of the services to close.
nsvcs	Number of services in ***svcname***.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***svcname*** pointer is NULL.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_SEQUENCE	There is a pending open or close service on the context.
CTAERR_NOT_FOUND	One or more services in ***svcname*** are not open on the context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Events

CTAEVN_CLOSE_SERVICES_DONE

Details

ctaCloseServices closes one or more services on a context. This function is asynchronous and returns immediately. The application must wait for the DONE event before all the services indicated in ***svcname*** are closed.

If multiple services are being closed and more than one service close fails, the value field of the CTA_EVENT structure (specified in **ctaWaitEvent**) contains the failure reason for the first service that failed to close.

Note: To more efficiently identify which service failed to close, close services one at a time.

Refer to *Closing services* on page 49 for more information.

See also

ctaOpenServices

Example

```
/* This is a synchronous function and assumes a single context */
/* per queue programming model has been used.                  */

void DemoCloseServices( CTAQUEUEHD ctaqueuehd, CTAHD ctahd,
                        char *services[], unsigned numservices)
```

```
{
    CTA_EVENT  event;

    /* Close services */
    ctaCloseServices( ctahd, services, numservices );

    /* Wait for the services to be closed asynchronously */
    do
    {
        ctaWaitEvent( ctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while (event.id != CTAEVN_CLOSE_SERVICES_DONE);

    if (event.value != CTA_REASON_FINISHED)
        printf ("Closing services failed: val=0x%x\n", event.value );
}
```

ctaCreateContext

Creates a context or a void context handle for the specified server.

Prototype

DWORD **ctaCreateContext** (CTAQUEUEHD ***ctaqueuehd***, unsigned ***userid***, char ***descriptor***, CTAHD ***ctahd***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.
userid	User-specified value repeated in all events returned for this context.
descriptor	Pointer to a string containing a user-defined context name and (optionally) the address of the server on which the context is created.
ctahd	Pointer to a returned context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid server address
CTAERR_BAD_NAME	Specified ***descriptor*** is in use.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_INCOMPATIBLE_SERVICES	On the server on which the context is created, at least one of the available services conflicts with the same service that is open on another server. A client application can use only one instance of a service at a time.
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_SVR_COMM	Server communication error.

Details

ctaCreateContext creates a context and (optionally) associates a userid and a descriptor with it. The userid is returned in all events for this context. Both the userid and the descriptor also appear in trace records. The context name part of the descriptor is limited to 11 characters (not including a null terminator) and must be unique to the server on which it is created.

If the descriptor does not specify a server, the function uses the default Natural Access Server (*ctdaemon*) specified in *cta.cfg*. Use **ctaSetDefaultServer** to override the default server.

The following table shows some examples of created contexts based on the location of Natural Access Server in relation to the application:

Location	Context example	Description
In-Process	`test_context`	Creates a context named test_context that is specific to the application and cannot be shared.
Local	`svr_context`	Creates a context named svr_context on a local server.
Remote	`localhost/srv_context`	Creates a context named srv_context on the local server.
Remote	`test_context`	Creates a context named test_context on the remote server.

When ***ctaqueuehd*** is NULL_CTAQUEUEHD, **ctaCreateContext** interprets ***descriptor*** as a server descriptor with a host address. If this parameter is a valid Natural Access Server descriptor, **ctaCreateContext** returns the void context handle containing the address of the Natural Access Server (*ctdaemon*) specified by ***descriptor***. Otherwise, the CTAERR_BAD_NAME error code is returned.

The ***descriptor*** parameter specifies the server for which the void context handle is created. The void context handle refers only to the server on which the commands are executed and not to any real context object. When ***descriptor*** is the string

```
cta://localhost
```

the returned void context handle corresponds to the local server.

When ***descriptor*** is the string

```
cta://inproc
```

the returned void context handle corresponds to the process address space.

For more information on descriptors, see *ctaGetObjDescriptor* on page 104**.**

After creating a context, you can open services on it and manage context parameters. Refer to *Creating contexts* on page 36 for more information.

Note: Use **ctaCreateContextEx** to create shared contexts. For more information, see *Sharing contexts* on page 36.

See also

ctaAttachContext, **ctaDestroyContext**, **ctaOpenServices**

Example 1

```
typedef struct
{
    unsigned   line;
    unsigned   ag_board;
    unsigned   mvip_stream;
    unsigned   mvip_slot;
    CTAQUEUEHD ctaqueuehd;
    CTAHD      ctahd;
    VCEHD      demovh;
    SWIHD      demoswihd;
} DEMOCONTEXT ;

/*             Creates a Natural Access context with its own queue, storing */
/*             in the context (cx) that was created by the caller.          */

void DemoCreateContext(unsigned line, DEMOCONTEXT *cx)
{
    char cxname[12];

    cx->line = line;

    /* Context name will be printed in all trace records */
    /* for this context */
    sprintf(cxname, "DEMOCX%04d", cx->line);

    /* Create queue and attach all service managers */
    ctaCreateQueue( NULL, 0, &(cx->ctaqueuehd) );

    /* Create context with line number as userid*/
    ctaCreateContext( cx->ctaqueuehd, line,
                      cxname, &(cx->ctahd) );
}
```

Example 2

```
/*Creates one Natural Access context in the process server and one Natural   */
/*Access context on a local server.                                          */

void DemoCreateContext(
     unsigned userid,
     CTAQUEUHD *ctaqueuehd,
     CTAHD *ctahd_library,
     CTAHD *ctahd_server)
{
     /* Context name for in-process server */
     char cxname_lib[] = "DEMOCX0";
     /* Context name for local server  */
     char cxname_svr[] = "localhost/DEMOCX1";

     /* Create queue and attach all service managers */
     ctaCreateQueue(NULL, 0, ctaqueuehd);

     /* Create a Natural Access context in the process server   */
     ctaCreateContext(*ctaqueuehd, userid, cxname_lib, ctahd_library);

     /* Create a Natural Access context on local server    */
     ctaCreateContext(*ctaqueuehd, userid, cxname_svr, ctahd_server);
}
```

Example 3

```
/* Retrieves global parameter values from the remote server*/
void DemoGetParms(
     unsigned parmid,
     void *buffer,
     unsigned size)
{
     CTAHD void_ctahd;
     /* Server name descriptor  */
     char server_desc[] = "host.nmss.com:2244";

     /* Create a void context handle */
     ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

     /* Retrieve parameter values from host.nmss.com server*/
     ctaGetParms (void_ctahd, parmid, butter, size);
}
```

ctaCreateContextEx

Creates a shared context, sets the context sharing mode, and returns a context handle (***ctahd***).

Prototype

DWORD **ctaCreateContextEx** (CTAQUEUEHD ***ctaqueuehd***, unsigned ***userid***, char ***descriptor***, CTAHD ***ctahd***, char ***flags***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.
userid	User-specified value repeated in all events returned for this context.
descriptor	Pointer to a string containing a user-defined context name and (optionally) the address of the server on which the context is created.
ctahd	Pointer to a returned context handle.
flags	Context sharing mode: • If CTA_CONTEXT_DECLARED_ACCESS is specified, the context is created in Declared Access mode. If the flag is not specified, the context is created in Common Access mode. • If CTA_CONTEXT_PERSISTENT is specified, the returned context is destroyed only if the last application calls **ctaDestroyContext**. Use this parameter to enable an application to reattach to a shared service mode context.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_NAME	Specified ***descriptor*** is in use.
CTAERR_INCOMPATIBLE_SERVICES	On the server on which the context is created, at least one of the available services conflicts with the same service that is open on another server. A client application can use only one instance of a service at a time.
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	This function cannot be used to create an in-process context. It can be used only to create contexts on local or remote instances of Natural Access Server.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaCreateContextEx creates a shared context on a local or remote server and associates a userid and descriptor with it. The userid is returned in all events for this context. Both the userid and the descriptor also display in trace records. The context name part of the descriptor is limited to 11 characters (not including a null terminator) and can be used by other applications to attach to the context.

After creating a context, you can open services on it, manage context parameters, and attach other contexts to it.

The following shared access modes are supported for contexts:

Access mode	Description
Common	Any service instances open on the context are immediately accessible through any contexts attached to it.
Declared	To access any service instances open on the context, each process using the shared context must open that service (with **ctaOpenServices**) on the context.

Note: This function can be used only by applications using local or remote instances of Natural Access Server (*ctdaemon*).

Refer to *Sharing contexts* on page 36 for more information.

See also

ctaAttachContext

Example

```
typedef struct
{
    unsigned   line;
    unsigned   ag_board;
    unsigned   mvip_stream;
    unsigned   mvip_slot;
    CTAQUEUEHD ctaqueuehd;
    CTAHD      ctahd;
    VCEHD      demovh;
    SWIHD      demoswihd;
} DEMOCONTEXT ;

/* Create a Natural Access queue and a context on the default server */
void DemoCreateContext(unsigned line, DEMOCONTEXT *cx)
{
    char cxname[12];

    cx->line = line;

    /* Context name will be printed in all trace records
     * for this context
     */
    sprintf(cxname, "DEMOCX%04d", cx->line);

    /* Create queue and attach all service managers   */
    ctaCreateQueue( NULL, 0, &(cx->ctaqueuehd) );

    /* Create context named "cxname" in restricted
     * access mode
     */
    ctaCreateContextEx( cx->ctaqueuehd, line, cxname, & cx->ctahd,
                    CTA_CONTEXT_DECLARED_ACCESS );
}
```

ctaCreateQueue

Creates an event queue and returns a queue handle.

Prototype

DWORD **ctaCreateQueue** (char ***mgrlist[]**, unsigned ***nmgrs***, CTAQUEUEHD ***ctaqueuehd***)

Argument	Description
mgrlist	Pointer to a list of service managers to attach to the queue. The name of each service manager must match one of the names specified with **ctaInitialize**.
nmgrs	Number of service managers in ***mgrlist.***
ctaqueuehd	Pointer to a returned queue handle.

Return values

Return value	Description
SUCCESS	
CTAERR_ALREADY_DEFINED	There is a duplicate service manager in ***mgrlist.***
CTAERR_BAD_ARGUMENT	***ctaqueuehd*** is invalid or NULL.
CTAERR_NOT_FOUND	One or more of the service managers in ***mgrlist*** is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_DRIVER_OPEN_FAILED	The ADI service driver open failed.

A service-manager-specific error, such as:

Return value	Description
CTAERR_DRIVER_OPEN_FAILED	The ADI service driver open failed.

Details

ctaCreateQueue creates an event queue and returns a handle (***ctaqueuehd***) used to reference the queue. When the application creates a context, the context is associated with this queue.

If ***mgrlist*** is NULL, all service managers registered with **ctaInitialize** are attached to the queue.

Use **ctaWaitEvent** to retrieve events. After creating the event queue, applications that are managing wait objects must set the CTA_NOTIFY_UPDATE_WAITOBJS flag in the ctaflags field of the CTA_INIT_PARMS structure in **ctaInitialize**. Call **ctaQueryWaitObjects** after calling **ctaCreateQueue**.

Refer to *Creating event queues* on page 35 for more information.

Note: When running on a local or a remote server, *cta.cfg* must contain a list of all service and service manager pairs expected to be used by any application. Otherwise, **ctaOpenServices** or **ctaInitialize** return CTAERR_NOT_FOUND.

See also

ctaDestroyQueue

Example

```
DEMOCONTEXT *Cxarray[100];
/* Setup a queue and multiple contexts for a state machine demo. */
void DemoSetup(CTAQUEUEHD *ctaqueuehd)
{
  unsigned   i;
  char      *mgrlist[] = { "ADIMGR", "VCEMGR", "SWIMGR" };

/* Create the Natural Access application queue and attach specified managers */
  ctaCreateQueue( mgrlist,
                  sizeof(mgrlist)/sizeof(mgrlist[0]),
                  ctaqueuehd);

  for (i=0; i < 100; i++)
  {
      char cxname[12];
      DEMOCONTEXT *cx = (DEMOCONTEXT *) malloc ( sizeof(DEMOCONTEXT) );

      cx->line       = i;
      cx->ctaqueuehd = *ctaqueuehd;

      /* Context name will be printed in all trace records for this context */
      sprintf( cxname, "DEMOCX%04d", i );
      /* Create context with index of demo context as userid */
      ctaCreateContext( cx->ctaqueuehd, i, cxname, &(cx->ctahd) );

      Cxarray[i] = cx;
  }
}
```

ctaDestroyContext

Destroys a context.

Prototype

DWORD **ctaDestroyContext** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_SEQUENCE	An open service, close service, or destroy context call is pending on the context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Events

CTAEVN_DESTROY_CONTEXT_DONE

Details

ctaDestroyContext destroys the context and closes all open services on that context. This function is asynchronous and returns immediately. The application must wait for the DONE event before the context is destroyed and any resources associated with it are released.

ctaDestroyContext closes pending open or close service calls. Wait until the pending function completes and then call this function.

If an application sets the CTA_CONTEXT_PERSISTENT flag in **ctaCreateContextEx**, the shared context continues to exist on the server even when the originating application is disconnected. When the last application attached to the context calls **ctaDestroyContext**, the context is destroyed.

Refer to *Closing services* on page 49 for more information.

See also

ctaAttachContext, **ctaCreateContext**

Example

```
/*
 * Destroy the context and the queue. It would be equivalent to just
 * destroy the queue since this implicitly destroys the context and
 * closes all open services.
 */
void DemoDestroyContext(DEMOCONTEXT *cx)
{
    CTA_EVENT event;

    /* Destroy the context */
    ctaDestroyContext( cx->ctahd );

    /* Wait for the services to be closed and the context destroyed */
    do
    {
        ctaWaitEvent( cx->ctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while (event.id != CTAEVN_DESTROY_CONTEXT_DONE);

    ctaDestroyQueue( cx->ctaqueuehd );
}
```

ctaDestroyQueue

Destroys all contexts created on the event queue, closes services, and destroys the queue.

Prototype

DWORD **ctaDestroyQueue** (CTAQUEUEHD ***ctaqueuehd***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAQUEUEHD	Already destroying the queue.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaDestroyQueue destroys all contexts created on the event queue, closes all services, and destroys the queue. Since this is a synchronous function, the application remains blocked until all cleanup and close activity is completed.

If any errors are returned other than those listed in the Return values section, **ctaDestoyQueue** had problems destroying the queue. Resources may be left in an unknown state.

Refer to *Closing services* on page 49 for more information.

See also

ctaDestroyContext

Example

```
/* Shutdown the state machine demo. */
void DemoShutdown(CTAQUEUEHD ctaqueuehd)
{
    unsigned i;

    /* Destroying the queue will destroy all contexts created on the queue. */
    ctaDestroyQueue( ctaqueuehd );

    /* Free all the contexts. */
    for (i=0; i < 100; i++)
    {
        if (Cxarray[i] != NULL)
            free ( Cxarray[i] );
    }
}
```

ctaDetachObject

Closes a service object on a server.

Prototype

DWORD **ctaDetachObject** (unsigned ***objhd***)

Argument	Description
objhd	Object handle returned by **ctaAttachObject**.

Return values

Return value	Description
SUCCESS	
CTA_INVALID_HANDLE	The specified ***objhd*** is not valid.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaDetachObject disconnects the application from the object handle that **ctaAttachObject** obtained.

Example

```
int DemoAttachDetach(char* contextname, char* objname)
{
     DWORD ret;

     CTAQUEUEHD ctaqueuehd;
     char *mgrlist[] = { "ADIMGR", "ADIMGR", "VCEMGR", "SWIMGR"};
     ret = ctaCreateQueue( mgrlist,
          sizeof(mgrlist)/sizeof(mgrlist[0]),
          &ctaqueuehd);
     if ( ret != SUCCESS )
          return ret;

     // Attach to existing server context
     CTAHD ctahd;
     ret = ctaAttachContext(ctaqueuehd,0,contextname,&ctahd);
     if ( ret != SUCCESS )
          return ret;
     // Attach to existing call
     unsigned callhd;
     ret = ctaAttachObject(&ctahd,objname,&callhd);
     if ( ret != SUCCESS )
          return ret;

     printf("Attached to call 0x%08X\n",callhd);

     // Get call status
     NCC_CALL_STATUS info;
     ret = nccGetCallStatus( callhd, &info, sizeof( NCC_CALL_STATUS ) );
     if ( ret != SUCCESS )
          return ret;

     printf("Called number: %s\n",info.calledaddr );

     // Detach from the call
     ret = ctaDetachObject(callhd);
     if ( ret != SUCCESS )
          return ret;

     printf("Detached\n");

     return SUCCESS;
}
```

ctaFindFile

Finds a file using the specified default extension and environment variable.

Prototype

DWORD **ctaFindFile** (char ***filename**, char ***extension**, char ***envvar**, char ***fullname**, unsigned ***size**)

Argument	Description
filename	Pointer to the full or partial file name of the file to find. If a path is specified, only that path is searched. If an extension is not specified in ***filename***, ***extension*** is appended.
extension	Pointer to the three letter extension to append to ***filename*** if one was not specified. If an ***extension*** is not needed, set this to NULL.
envvar	Pointer to the name of the environment variable. If ***filename*** does not include a path, the path specified by ***envvar*** is searched. If no environment variable is to be searched, set this to NULL.
fullname	Pointer to a buffer to receive the complete file name that includes the full path.
size	Size of the ***fullname*** buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***filename*** pointer is NULL.
CTAERR_BAD_SIZE	***size*** is too small to contain the found path.
CTAERR_FILE_NOT_FOUND	The specified file does not exist.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaFindFile finds the specified file. It does not open it or create it.

fullname includes the default ***extension*** if it was appended.

If ***filename*** ends with a period (.), ***extension*** is not appended to the file name.

If the file is...	Then the buffer receives the...
Found	Full path name of the file.
Not found	File name as provided and the optional default extension. This value can be used directly as the complete file name for opening or creating a file.

size should be at least CTA_MAXPATH long to accommodate a long directory path. If ***fullname*** is NULL, file existence is checked.

Example

```
void DemoTestFindFile()
{
    DWORD ret;
    char  name[CTA_MAX_PATH] ;
    char  ext[50];
    char  env[50];
    char  fullpath[CTA_MAX_PATH] ;

    printf( "Enter file name: " );
    gets( name );

    printf( "Enter extension (optional): " );
    gets( ext );

    printf( "Enter search path environment variable: " );
    gets( env );

    ret = ctaFindFile( name, ext, env, fullpath, sizeof(fullpath) );

    if (ret == SUCCESS)
        printf("File found: %s\n", fullpath);
    else
        printf("File not found.");
}
```

ctaFindFileEx

Finds a file on a server using the specified default extension and environment variable.

Prototype

DWORD **ctaFindFileEx** (CTAHD ***ctahd,*** char ****filename***, char ****extension***, char ****envvar***, char ****fullname***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
filename	Pointer to the full or partial file name of the file to find. If a path is specified, only that path is searched. If an extension is not specified in ***filename***, ***extension*** is appended.
extension	Pointer to the 3-letter extension to append to ***filename*** if one was not specified. If an ***extension*** is not needed, set this to NULL.
envvar	Pointer to the name of the environment variable. If ***filename*** does not include a path, the path specified by ***envvar*** is searched. If no environment variable is to be searched, set this to NULL.
fullname	Pointer to a buffer to receive the complete file name that includes the full path.
size	Size of the ***fullname*** buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***filename*** pointer is NULL.
CTAERR_BAD_SIZE	***size*** is too small to contain the found path.
CTAERR_FILE_NOT_FOUND	The specified file does not exist.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communications error.

Details

ctaFindFileEx finds the specified file. It does not open it or create it.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

If ***filename*** ends with a period (.), ***extension*** is not appended to the file name.

If the file is...	Then the buffer receives the...
Found	Full path name of the file.
Not found	File name as provided and the optional default extension. This value can be used directly as the complete file name for opening or creating a file.

If a partial file name is specified, the server specified by ***ctahd*** determines the default directory path.

If the server is...	Then the default directory path is the path of...
In-process	The parent process of the application.
Local	The local server on which Natural Access Server (*ctdaemon*) started.
Remote	The remote server on which Natural Access Server (*ctdaemon*) started.

fullname includes the default ***extension*** if it is appended.

size should be at least CTA_MAXPATH long to accommodate a long directory path. If ***fullname*** is NULL, file existence is checked.

Example

```
void DemoTestFindFile()
{
    DWORD ret;
    char  name[CTA_MAX_PATH] ;
    char  ext[50];
    char  env[50];
    char  fullpath[CTA_MAX_PATH] ;

    CTAHD void_ctahd;
    /* Server name descriptor  */
    char server_desc[] = "host.nmss.com:2244";
    /* Create a void context handle */
    ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

    printf( "Enter file name: " );
    gets( name );

    printf( "Enter extension (optional): " );
    gets( ext );

    printf( "Enter search path environment variable: " );
    gets( env );

    ret = ctaFindFileEx( void_ctahd, name, ext, env, fullpath,
                         sizeof(fullpath) );

    if (ret == SUCCESS)
        printf("File found: %s\n", fullpath);
    else
        printf("File not found.");
}
```

ctaFormatEvent

Formats an event into a string for print diagnostics.

Prototype

DWORD **ctaFormatEvent** (char ***lineprefix**, CTA_EVENT ***ctaevt**, char ***buffer**, unsigned ***size**)

Argument	Description
lineprefix	Pointer to a character string that is placed at the beginning of every new line in the formatted event string.

ctaevt	Pointer to an event structure to be formatted. The CTA_EVENT structure is:
	`typedef struct`
	`{`
	`  DWORD  id;  /* Event code and source service ID */`
	`  CTAHD  ctahd;  /* Natural Access context handle */`
	`  DWORD  timestamp;  /* Timestamp */`
	`  DWORD  userid;  /* Userid (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 object handle */`
	`} CTA_EVENT;`

buffer	Pointer to the buffer to receive the event string.
size	Size of the buffer in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is not big enough to fit the formatted event.
CTAERR_NOT_FOUND	The service to format the event is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaFormatEvent creates a detailed textual description of the event.

Note: If you are working with OAM service events, refer to the *NMS OAM Service Developer's Reference Manual* for information on formatting events.

See also

ctaFormatEventEx, **ctaGetText**

Example

```
void DemoShowEvent( CTA_EVENT *event )
{
    char format_buffer[CTA_MAX_FORMAT_SIZE];
    char *prefix = "\t\t\t";  /* default demo indent */

    format_buffer[0] = '\0';

    ctaFormatEvent( prefix, event, format_buffer, CTA_MAX_FORMAT_SIZE );

    printf( "%s", format_buffer );

}
```

ctaFormatEventEx

Formats an event on a specified server into a string for print diagnostics.

Prototype

DWORD **ctaFormatEventEx** (CTAHD ***ctahd***, char ****lineprefix***, CTA_EVENT ****ctaevt***, char ****buffer***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
lineprefix	Pointer to a character string that is placed at the beginning of every new line in the formatted event string.

ctaevt	Pointer to an event structure to be formatted. The CTA_EVENT structure is
	`typedef struct`
	`{`
	`  DWORD  id;  /* Event code and source service ID */`
	`  CTAHD  ctahd;  /* Natural Access context handle */`
	`  DWORD  timestamp;  /* Timestamp */`
	`  DWORD  userid;  /* Userid (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 object handle */`
	`} CTA_EVENT;`

buffer	Pointer to the buffer to receive the event string.
size	Size of the buffer in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is not big enough to fit the formatted event.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_NOT_FOUND	Service to format the event is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaFormatEventEx creates a detailed textual description of the event.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

Use **ctaCreateContext** to obtain the void context handle that corresponds to a server.

Note: If you are working with OAM service events, refer to the *NMS OAM Service Developer's Reference Manual* for information on formatting events.

See also

ctaCreateContextEx, **ctaGetText**

Example

```
void DemoShowEvent( CTA_EVENT *event )
{
    char format_buffer[CTA_MAX_FORMAT_SIZE];
    char *prefix = "\t\t\t";  /* default demo indent */

    CTAHD void_ctahd;
    /* Server name descriptor  */
    char server_desc[] = "host.nmss.com:2244";
    /* Create a void context handle */
    ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

    format_buffer[0] = '\0';

    ctaFormatEventEx( void_ctahd, prefix, event, format_buffer,
                      CTA_MAX_FORMAT_SIZE );

    printf( "%s", format_buffer );

}
```

ctaFreeBuffer

Frees internally allocated buffers or buffers allocated with **ctaAllocBuffer**.

Prototype

DWORD **ctaFreeBuffer** (void ***buffer***)

Argument	Description
buffer	Pointer to a buffer allocated with **ctaAllocBuffer**, or to an internal buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	If ***buffer*** is NULL.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaFreeBuffer frees internally allocated buffers and memory allocated with **ctaAllocBuffer**.

If an event received by **ctaWaitEvent** contains data, the buffer can be allocated inside Natural Access (data allocated and sent from a Natural Access service). In this case, the flag CTA_INTERNAL_BUFFER is set in the ***size*** field. Use **ctaFreeBuffer** to free the event buffer.

Example

```
CTAHD commCtahd = NULL_CTAHD,  mediaCtahd = NULL_CTAHD;

void EventLoop()
{
    DWORD ret;

    for(;;)
    {
        CTA_EVENT   event = { 0 };
        int         ctaOwnsBuf = 0;

        event.ctahd = commCtahd;
        if( ret = ctaWaitEvent( commCtahd, &event ) != SUCCESS )
        {
            printf("ctaWaitEvent returns 0x%x -- exiting\n", "ret );
            exit( -1 );
        }

        /* In Natural Access server mode, certain event buffers are allocated by
        *  Natural Access and then must be released back to Natural Access.
        */
        if ( event.buffer )
        {
           if ( ctaOwnsBuf = event.size & CTA_INTERNAL_BUFFER )
           {
               /* This buffer is owned by Natural Access and will need to be
               *  released back to Natural Access after processing of the
               *  buffer is complete. Clear the Natural Access flags from the
               *  size field.
               */
               event.size &= ~CTA_INTERNAL_BUFFER;
           }
        }

               /* Event processing */
        switch ( event.id )
        {
            case APPEVN_EXCHANGE_NAME:
                          . . .
            default:
                break;

        }

        /* Ensure proper release of Natural Access owned buffers. */
        if ( ctaOwnsBuf )
        {
            ctaOwnsBuf = 0;
            ctaFreeBuffer( event.buffer );
        }
    }
}
```

ctaGetContextInfo

Obtains basic context information.

Prototype

DWORD **ctaGetContextInfo** (CTAHD ***ctahd***, CTA_CONTEXT_INFO **** info***, unsigned ***size***)

Argument	Description
ctahd	Context handle.

info	Pointer to the CTA_CONTEXT_INFO structure:
	`typedef struct`
	`{`
	`  unsigned size;  /* Size of returned structure */`
	`  CTAQUEUEHD ctaqueuehd;  /* Natural Access queue handle */`
	`  unsigned userid;  /* Userid passed back in events */`
	`  char contextname[CTA_CONTEXT_NAME_LEN];`
	`  /* Context name */`
	`}CTA_CONTEXT_INFO;`

size	Size of the CTA_CONTEXT_INFO structure.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid ***info***, ***size***, or ***ctahd*** parameter.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetContextInfo obtains basic information for the context specified by the ***ctahd*** parameter.

See also

ctaGetContextInfoEx

Example

```
int DemoContextInfo(CTAHD ctahd)
{
        DWORD ret;
        CTA_CONTEXT_INFO info;
        ret = ctaGetContextInfo(ctahd,&info,sizeof(info));
        if ( ret != SUCCESS )
                return ret;
        printf("Context name: %s \n", info.contextname);
        return SUCCESS;
}
```

ctaGetContextInfoEx

Obtains extended context information.

Prototype

DWORD **ctaGetContextInfoEx** (CTAHD ***ctahd***, CTA_CONTEXT_INFOEX ***** ***info***)

Argument	Description
ctahd	Context handle.

info	Pointer to the CTA_CONTEXT_INFOEX structure:
	`typedef struct`
	`{`
	`  CTAQUEUEHD ctaqueuehd;  /* Natural Access queue handle */`
	`  unsigned userid;  /* Userid passed back in events */`
	`  char contextname[CTA_CONTEXT_NAME_LEN]; /* Context name */`
	`  char* svcnames;  /* Returned service names */`
	`  /* terminated by 0, with an */`
	`  /* extra 0 at the end */`
	`  unsigned size;  /* Size of svcnames buffer */`
	`}  CTA_CONTEXT_INFOEX;`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid ***info*** or ***ctahd*** parameter.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetContextInfoEx obtains extended information including a list of services opened on the context specified by the ***ctahd*** parameter.

See also

ctaGetContextInfo

Example

```
int DemoContextInfoEx(CTAHD ctahd)
{
        DWORD ret;
        unsigned i;
        char* p;
        CTA_CONTEXT_INFOEX info;
        info.svcnames = (char*) malloc(1024);
        info.size = 1024;
        ret = ctaGetContextInfoEx(ctahd,&info);
        if ( ret != SUCCESS )
        {
              free(info.svcnames);
              return ret;
        }
        printf("Context name: %s \n", info.contextname);
        p = info.svcnames;

        for (i = 0; i < info.size; i++)
        {
              if (*(info.svcnames + i) == 0)
              {
                     if (*(info.svcnames + i + 1) == 0 ||
                           i == 0)
                          break;
                     printf("Service: %s\n",p);
                     p = info.svcnames + i + 1;
              }
        }
        free(info.svcnames);
        return SUCCESS;
}
```

ctaGetEventSources

Retrieves a list of service names from which the specified context receives events.

Prototype

DWORD **ctaGetEventSources** (CTAHD ***ctahd***, char ****svcname*[]**, unsigned ***nsvcs***)

Argument	Description
ctahd	Context handle.
svcname	Pointer to a buffer to return a list of services sending events to the context.
nsvcs	Number of services in the list.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Buffer is NULL.
CTAERR_BAD_SIZE	Size of the buffer is too small for the list of service names.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetEventSources retrieves the list of services from which the context receives events. Use **ctaSetEventSources** to specify or change the list of services from which the context receives events.

Refer to *Receiving events on event queues* on page 46 for more information.

See also

ctaAttachContext, **ctaCreateContextEx**

Example

```
char svcnamebuffer[SVC_NAME_BUFFER_SIZE];
ret = ctaGetEventSources(ctahd, svcname[], nsvcs);
if (ret == SUCCESS)
{
     printf("The services which are masked are:\n");
     PrintStringofStrings(svcnamebuffer);
}
```

ctaGetObjDescriptor

Retrieves the Natural Access object descriptor associated with the specified context or object handle.

Prototype

DWORD **ctaGetObjDescriptor** (CTAHD ***hd***, char ****descriptor***, DWORD ***size***)

Argument	Description
hd	Service object handle or context handle
descriptor	Pointer to a buffer to receive the Natural Access object descriptor in text format.
size	Size of the descriptor buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Buffer is NULL, or ***size*** is 0.
CTAERR_BAD_SIZE	Descriptor buffer is too small.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaGetObjDescriptor returns the full path of the object descriptor associated with the specified context or object handle.

A descriptor is a character string that uniquely identifies a Natural Access processing resource (for example, a context or a service object). Applications attach to existing contexts by invoking **ctaAttachContext** and specifying the descriptor. Applications attach to existing service objects by invoking **ctaAttachObject** and specifying the descriptor.

When creating the context as a short cut for the descriptor, the application can specify a context name of up to 11 characters long (for example, common). Natural Access can automatically generate the descriptor and the application can obtain the descriptor later using **ctaGetObjDescriptor**. The following table shows the format of the returned descriptor:

Context object	Service object
[prot://][host]ctx_name	***[prot://][host]ctx_name/svc_obj_name***

The following table provides examples of valid descriptors:

Descriptor example	Description
`cta://localhost/common`	***ctx_name*** (common) is user-specified.
`cta://127.0.0.1/0x80010001`	***ctx_name*** (0x80010001) is automatically generated by Natural Access.
`context`	Descriptor contains only the context name. The host part of the descriptor defaults depending on the application mode.
`localhost/context_name`	A context created in a Natural Access Server process that is separate from the application, but on the same host.

To access a shared resource, applications can transfer descriptors from one application to another. Refer to *Attaching to shared contexts* on page 40 and *Sharing service objects* on page 43 for more information.

See also

ctaCreateContextEx

Example

```
case PAIR( 'D', 'H' ):              /* GET CONTEXT DESCRIPTOR */
    {
        char descriptor[128];

        if ( ctaGetObjDescriptor(ctahd, descriptor, 128) == SUCCESS )
            printf("Descriptor = %s\n", descriptor);
        else
            printf("Unable to get descriptor for ctahd 0x%x\n", ctahd);

        break;
    }
```

ctaGetParmByName

Retrieves the values for a given parameter.

Prototype

DWORD **ctaGetParmByName** (CTAHD ***ctahd***, char ****parmname***, void ****buffer***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmname	Pointer to a string containing the parameter name to retrieve.
buffer	Pointer to a buffer to receive parameter values.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***buffer*** is not big enough to hold the parameter field data.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmname*** not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmByName retrieves the values for the parameter identified as ***parmname***. If ***ctahd*** is a valid context, this function retrieves the parameters from the context defaults.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

If a void context handle is used for the ***ctahd*** parameter, this function retrieves parameters from the global defaults and is executed on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

Natural Access writes the values to the buffer. The ***size*** argument specifies the amount of available memory at the ***buffer*** and must be large enough to contain the returned parameter value.

The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***]. The service name and parameter structure name must be supplied, but the extension specifier and field name are optional. If the parameter or parameter structure are extensions to the standard service parameters, the literal text .x must be included in ***parmname***. If the field name is omitted, the complete default parameter structure is returned.

An example of ***parmname*** is vce.play.gain.

This function is intended primarily for utility applications. For example, use **ctaGetParmByName** to display the default value of a parameter based on its name. Use **ctaGetParms** to retrieve default parameter structures by ID.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetParmByNameEx, **ctaGetParmInfo**, **ctaGetParms**, **ctaSetEventSources**

Example

```
void DemoPrintPlayGain(CTAHD ctahd)
{
    INT32 playgain;

    ctaGetParmByName( ctahd, "vce.play.gain",
                      &playgain, sizeof(playgain) );
    printf( "Default vce.play.gain is %d dB\n", playgain);
}
```

ctaGetParmByNameEx

Retrieves the values for a given parameter from a specified Natural Access Server.

Prototype

DWORD **ctaGetParmByNameEx** (char ***addr***, CTAHD ***ctahd***, char ****parmname***, void ****buffer***, unsigned ***size***)

Argument	Description
addr	Natural Access Server address.
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmname	Pointer to a string containing the parameter name to retrieve.
buffer	Pointer to a buffer to receive parameter values.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***buffer*** is not big enough to hold the parameter field data.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmname*** not found.
CTAERR_NOT_IMPLEMENTED	This function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmByNameEx retrieves the values for the parameter identified as ***parmname***. If ***ctahd*** is a valid context, this function retrieves the parameters from the server associated with the context.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle.

If a void context handle is used for the ***ctahd*** parameter, this function retrieves parameters from the global defaults and is executed on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

If the context handle is equal to NULL_CTAHD, the function retrieves the parameters from the server specified by ***addr***. If ***addr*** is equal to NULL or is an empty string, the function retrieves parameters from the default server.

Natural Access writes the values to the buffer. The ***size*** argument specifies the amount of available memory at the ***buffer*** and must be large enough to contain the returned parameter value.

The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***]. The service name and parameter structure name must be supplied, but the extension specifier and field name are optional. If the parameter or parameter structure are extensions to the standard service parameters, the literal text .x must be included in ***parmname***. If the field name is omitted, the complete default parameter structure is returned. An example of ***parmname*** is vce.play.gain.

ctaGetParmByNameEx is intended primarily for utility applications. For example, use **ctaGetParmByNameEx** to display the default value of a parameter based on its name. Use **ctaGetParms** to retrieve default parameter structures by ID.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetParmByName, **ctaGetParmID**, **ctaGetParmIds**, **ctaGetParmIdsEx**, **ctaGetParmInfo**, **ctaGetParms**, **ctaGetParmsEx**, **ctaSetParmByName**, **ctaSetParmByNameEx**

Example

```
void DemoPrintPlayGain(CTAHD ctahd)
{
    INT32 playgain;

    ctaGetParmByNameEx( "localhost", ctahd, "vce.play.gain",
                        &playgain, sizeof(playgain) );
    printf( "Default vce.play.gain is %d dB\n", playgain);
}
```

ctaGetParmID

Retrieves the parameter ID for a given parameter descriptor name.

Prototype

DWORD **ctaGetParmID** (CTAHD ***ctahd***, char ****parmname***, unsigned ****parmid***)

Argument	Description
ctahd	Context handle.
parmname	Pointer to the parameter name. Must be a valid parameter name.
parmid	Pointer to the parameter ID. Value cannot be NULL.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***parmname*** is NULL or ***parmid*** is NULL.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	Parameter name is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmID retrieves the parameter ID for a given parameter descriptor name. This function enables you to take advantage of the faster processing of parameters by ID without needing to hard code the parameter ID numbers.

Parameter IDs are fixed in Natural Access. The IDs are defined in the header files.

Note: Set ***ctahd*** to NULL_CTAHD if **ctaGetParmID** must be called before any contexts are created with **ctaCreateContext**.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmInfo**, **ctaGetParms**, **ctaLoadParameterFile**, **ctaRefreshParms**, **ctaSetEventSources**

Example

```
void DemoRecordFile(DEMOCONTEXT *cx, char *filename, unsigned encoding)
{
    VCEHD            vh;
    CTA_EVENT        event;
    VCE_RECORD_PARMS rparms;
    DWORD            vcerecordid;
    unsigned         maxtime = 0;  /* no cap on length of recording */
    /*
     * Record the caller, overriding default parameters for
     * quick recognition of silence once voice input has begun.
     */
    ctaGetParmID( cx->ctahd, "vce.record", &vcerecordid );
    ctaGetParms( cx->ctahd, vcerecordid, &rparms, sizeof(rparms) );
    rparms.silencetime = 500; /* 1/2 sec */
    vceOpenFile ( cx->ctahd, filename, VCE_FILETYPE_FLAT, VCE_PLAY_RECORD,
                  encoding, &vh );
    vceRecordMessage( vh, 0, maxtime, &rparms);
    /* Wait for recording to end */
    do
    {
        ctaWaitEvent( cx->ctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while ( event.id != VCEEVN_RECORD_DONE );
    printf ( "Record ended with reason: val=0x%x\n", event.value );

    vceClose (vh);
}
```

ctaGetParmIds

Returns the parameter IDs and service ID for the named service and service manager.

Prototype

DWORD **ctaGetParmIds** (char ***svcname**, char ***svcmgrname**, unsigned ***retsvcid**, unsigned ***buffer**, unsigned ***maxsize**, unsigned ***retsize**)

Argument	Description
svcname	Pointer to the name of the service for which parameter IDs and service IDs are requested.
svcmgrname	Pointer to the associated service manager name (optional).
retsvcid	Pointer to returned service ID or NULL.
buffer	Pointer to returned array of parameter IDs.
maxsize	Number of available entries in the buffer.
retsize	Pointer to the returned value of the number of entries used in the buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***svcname***, ***buffer***, or ***retsize*** is NULL.
CTAERR_BAD_SIZE	Size of the buffer is too small.
CTAERR_NOT_FOUND	Service is not available to Natural Access.
CTAERR_SVR_COMM	Application is unable to communicate with the server.

Details

ctaGetParmIds retrieves all defined parameter IDs for a given named service. There can be different service managers associated with a service interface and it is optional whether you specify the associated service manager.

The parameter IDs are defined in the various service specific header files of Natural Access.

If ***retsvcid*** is not NULL, the service ID value for the named service is returned.

Service names passed to **ctaGetParmIds** must display in the list of services known to the default Natural Access Server. The list of services is located in the [ctasys] section of the Natural Access configuration file, *cta.cfg*.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmIdsEx**, **ctaGetParmInfo**, **ctaGetParms**, **ctaLoadParameterFile**, **ctaRefreshParms**, **ctaSetEventSources**

Example

```
static CTA_SERVICE_DESC pServicesWithNcc[] =  /* for ctaOpenServices */
{   { {"ADI", NULL},     { 0 }, { 0 }, { 0 } },
    { {"NCC", NULL},     { 0 }, { 0 }, { 0 } },
    { {"SWI", "SWIMGR"}, { 0 }, { 0 }, { 0 } },
    { {"VCE", "VCEMGR"}, { 0 }, { 0 }, { 0 } }
};

static CTA_SERVICE_DESC* pServices = pServicesWithNcc;
static unsigned SizeofServices = sizeof(pServicesWithNcc);

void ShowDefaultParm( CTAHD ctahd, char *name )
{
    unsigned      i;
    unsigned      ret;
    CTA_PARM_INFO info;

    unsigned i, j;

    for( i = 0; i < SizeofServices/sizeof(CTA_SERVICE_DESC); i++)
    {
        unsigned ids[100];
        unsigned numids = 0;
        unsigned serviceID = 0;

        ctaGetParmIds( pServices[i].name.svcname,
                       pServices[i].name.svcmgrname,
                       &serviceID,
                       ids, 100, &numids );

        for (j=0; j < numids; j++)
        {
            /* Display each of the parameter structure names */
            ctaGetParmInfoEx( ctahd, ids[j], NULL, 0, &info );
            printf( "%s\n", info.structname );
        }
    }
}
```

ctaGetParmIdsEx

Returns the parameter IDs and service ID for the named service and service manager that are available on a particular Natural Access Server.

Prototype

DWORD **ctaGetParmIdsEx** (CTAHD ***ctahd***, char ****svcname***, char ****svcmgrname***, unsigned ****retsvcid***, unsigned ****buffer***, unsigned ***maxsize***, unsigned ****retsize***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
svcname	Pointer to the name of the service for which parameter IDs and service IDs are requested.
svcmgrname	Pointer to the associated service manager name (optional).
retsvcid	Pointer to the returned service ID or NULL.
buffer	Pointer to the returned array of parameter IDs.
maxsize	Number of available entries in the buffer.
retsize	Pointer to the returned value of the number of entries used in the buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***svcname***, ***buffer***, or ***retsize*** is NULL.
CTAERR_BAD_SIZE	Size of the buffer is too small.
CTAERR_NOT_FOUND	Service is not available to the Natural Access server.
CTAERR_SVR_COMM	Application is unable to communicate with the server.

Details

ctaGetParmIdsEx retrieves all defined parameter IDs for a given named service. There can be different service managers associated with a service interface, and it is optional whether you specify the associated service manager.

The parameter IDs are defined in the various service-specific header files of Natural Access.

If ***retsvcid*** is not NULL, the service ID value for the named service is returned.

The service names passed to **ctaGetParmIdsEx** must display in the list of services known to the specified Natural Access server. The list of services is included in the [ctasys] section of the Natural Access configuration file, *cta.cfg*. Use ***ctahd*** to indicate which Natural Access server to process this command.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmIds**, **ctaGetParmInfo**, **ctaGetParms**, **ctaLoadParameterFile**, **ctaRefreshParms**, **ctaSetEventSources**

Example

```
static CTA_SERVICE_DESC pServicesWithNcc[] =  /* for ctaOpenServices */
{   { {"ADI", NULL},     { 0 }, { 0 }, { 0 } },
    { {"NCC", NULL},     { 0 }, { 0 }, { 0 } },
    { {"SWI", "SWIMGR"}, { 0 }, { 0 }, { 0 } },
    { {"VCE", "VCEMGR"}, { 0 }, { 0 }, { 0 } }
};

static CTA_SERVICE_DESC* pServices = pServicesWithNcc;
static unsigned SizeofServices = sizeof(pServicesWithNcc);
void ShowDefaultParm( CTAHD ctahd, char *name )
{
    CTAHD         voidHandle;
    unsigned      i;
    unsigned      ret;
    CTA_PARM_INFO info;

    unsigned i, j;

    /* Get the void context handle for the remote Natural Access
    *  server running on host NMS001
    */
    ctaCreateContext( NULL_CTAQUEUEHD, 0, "//NMS001", &voidHandle );

    for( i = 0; i < SizeofServices/sizeof(CTA_SERVICE_DESC); i++)
    {
        unsigned ids[100];
        unsigned numids = 0;
        unsigned serviceID = 0;

        ctaGetParmIdsEx( voidHandle,
                         pServices[i].name.svcname,
                         pServices[i].name.svcmgrname,
                         &serviceID,
                         ids, 100, &numids );

        for (j=0; j < numids; j++)
        {
            /* Display each of the parameter structure names */
            ctaGetParmInfoEx( ctahd, ids[j], NULL, 0, &info );
            printf( "%s\n", info.structname );
        }
    }
}
```

ctaGetParmInfo

Retrieves a parameter field definition.

Prototype

DWORD **ctaGetParmInfo** (unsigned ***parmid***, char ****parmname***, unsigned ***index***, CTA_PARM_INFO ****buffer***)

Argument	Description
parmid	Parameter structure ID.
parmname	Pointer to the parameter name (use NULL if the ***parmid*** specifies a valid parameter ID).
index	Relative field within the parameter structure.

buffer	Pointer to a buffer to receive a copy of CTA_PARM_INFO for the specified parameters. The CTA_PARM_INFO structure is:
	`typedef struct`
	`{`
	`  char  structname[40];`
	`  char  fieldname[40];`
	`  DWORD offset;`
	`  DWORD size;`
	`  DWORD format;`
	`  DWORD units;`
	`} CTA_PARM_INFO;`
	See the Details section for a description of these fields.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL, or ***parmname*** is NULL and ***parmid*** is zero (0).
CTAERR_NOT_FOUND	***parmname*** not found, ***parmid*** is invalid, or ***index*** is out of range.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmInfo retrieves a field definition conveyed as a CTA_PARM_INFO structure. The parameter structure can be specified either by ***parmname*** or ***parmid***.

If ***parmname*** is non-NULL, Natural Access searches for the parameter structure therein. The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***]. The service name and parameter structure name must be supplied, but the extension specifier and field name are optional. If the parameter or parameter structure are extensions to the standard service parameters, the literal text .x must be included in ***parmname***.

If the field name is not supplied in ***parmname***, Natural Access uses ***index*** as the field selector. The first field within the structure is indexed as zero (0), the second as one (1), and so on.

If ***parmname*** is NULL, ***parmid*** is used to retrieve the parameter structure and ***index*** is used as the field selector. The parameter IDs are defined in the service header file (for example, *vcedef.h*).

The IDs are usually of the following form:
SERVICENAME[***_X***]***_STRUCTNAME_PARMID***.

The service name is the same name referenced in **ctaInitialize** and **ctaOpenServices**. If the structure references non-standard service extension parameters, the ***_X*** is used to distinguish these non-standard structures. An example of a standard structure ***parmid*** is VCE_PLAY_PARMID.

The returned CTA_PARM_INFO structure contains the following fields:

Field	Description
structname	The parameter's structure name. Use when using ***parmid***.
fieldname	The parameter's field name.
offset	The value's offset in the parameter structure.
size	The parameter's value size (bytes).
format	The parameter's value format. Refer to the format table.
units	The parameter's value units. Refer to the unit table.

The following formats are supported:

Format	Description
CTA_FMT_WORD	Field is a 16-bit unsigned quantity.
CTA_FMT_DWORD	Field is a 32-bit unsigned quantity.
CTA_FMT_INT16	Field is a 16-bit signed quantity.
CTA_FMT_INT32	Field is a 32-bit signed quantity.
CTA_FMT_STRING	Field is a NULL-terminated string.
CTA_FMT_UNKNOWN	Field contains an unknown format.

The following units are supported:

Format	Description
CTA_UNITS_INTERNAL	No engineering units available.
CTA_UNITS_INTEGER	Field is a count or an index.
CTA_UNITS_COUNT	Field is a number of something.
CTA_UNITS_MASK	Field is a bit mask.
CTA_UNITS_HZ	Field is a measurement of frequency, in Hz.
CTA_UNITS_MS	Field is measured in milliseconds.
CTA_UNITS_DB	Field is a relative signal level, measured in dB.
CTA_UNITS_DBM	Field is an absolute signal level.
CTA_UNITS_IDU	Field is in internal DSP units (NMS internal representation for amplitude).
CTA_UNITS_STRING	Field contains a string.
CTA_UNITS_PERCENT	Field contains a percentage (where 100 is 100 percent).

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmInfoEx**, **ctaGetParms**, **ctaSetEventSources**

Example

```
void myShowParmInfo( char *parmname )
{
    CTA_PARM_INFO info;
    BYTE temp[100];     /* temp storage for parm value */
    union
    {
        WORD  w;
        DWORD W;
        INT16 i;
        INT32 I;
        char  s[80];
    } *pdata =( void* ) temp;
    char *punits;

    if( ctaGetParmInfo( 0, parmname, 0, &info ) != SUCCESS
        || ctaGetParmByName(NULL_CTAHD, parmname, &temp, sizeof( temp) )
        != SUCCESS )
        {
            /* display error */
            return;
        }

    switch( info.units )
        {
        case CTA_UNITS_INTERNAL: punits = "Internal";     break;
        case CTA_UNITS_INTEGER : punits = "Integer";      break;
        case CTA_UNITS_COUNT   : punits = "Count";        break;
        case CTA_UNITS_MASK    : punits = "Mask";         break;
        case CTA_UNITS_HZ      : punits = "Hz";           break;
        case CTA_UNITS_MS      : punits = "ms";           break;
        case CTA_UNITS_DB      : punits = "dB";           break;
        case CTA_UNITS_DBM     : punits = "dBm";          break;
        case CTA_UNITS_IDU     : punits = "Internal DSP"; break;
        case CTA_UNITS_STRING  : punits = "String";       break;
        case CTA_UNITS_PERCENT : punits = "Percent";      break;
        default                : punits = "Undefined";    break;
        }

    switch( info.format )
        {
        case CTA_FMT_WORD:
            printf( "%s.%-20s = %5u\t\t# (0x%04x) WORD  (%s)\n",
                info.structname, info.fieldname, pdata->w, pdata->w, punits );
            break;
        case CTA_FMT_DWORD:
            printf( "%s.%-20s = %5u\t\t# (0x%04x) DWORD (%s)\n",
                info.structname, info.fieldname, pdata->W, pdata->W, punits );
            break;
        case CTA_FMT_INT16:
            printf( "%s.%-20s = %5d\t\t# (0x%04x) INT16 (%s)\n",
                info.structname, info.fieldname, pdata->i, pdata->i, punits );
            break;
        case CTA_FMT_INT32:
            printf( "%s.%-20s = %5d\t\t# (0x%04x) INT32 (%s)\n",
                info.structname, info.fieldname, pdata->I, pdata->I, punits );
            break;
        case CTA_FMT_STRING:
            printf( "%s.%-20s = \"%s\"\t\t# STRING[%d]\n",
                info.structname, info.fieldname, pdata->s, info.size );
            break;
        default:
            printf( "Error! Unknown data type: '%c'\n", info.format );
            break;
        }
        return;
}
```

Sample run

For the parameter vce.play.gain, the output is:

```
vce.play.gain                          0                          # (0x0000) INT32 (dB)
```

ctaGetParmInfoEx

Retrieves a parameter field definition on a particular Natural Access Server.

Prototype

DWORD **ctaGetParmInfoEx** (CTAHD ***ctahd***, unsigned ***parmid***, char ****parmname***, unsigned ***index***, CTA_PARM_INFO ****buffer***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmid	Parameter structure ID.
parmname	Pointer to the parameter name (use NULL if the ***parmid*** specifies a valid parameter ID).
index	Relative field within the parameter structure.

buffer	Pointer to a buffer to receive a copy of CTA_PARM_INFO for the specified parameters. The CTA_PARM_INFO structure is:
	`typedef struct`
	`{`
	`  char  structname[40];`
	`  char  fieldname[40];`
	`  DWORD offset;`
	`  DWORD size;`
	`  DWORD format;`
	`  DWORD units;`
	`} CTA_PARM_INFO;`
	See the Details section for a description of these fields.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL, or ***parmname*** is NULL and ***parmid*** is zero (0).
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_NOT_FOUND	***parmname*** is not found, ***parmid*** is invalid, or ***index*** is out of range.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmInfoEx retrieves a field definition conveyed as a CTA_PARM_INFO structure. The parameter structure can be specified either by ***parmname*** or ***parmid***.

If ***parmname*** is non-NULL, Natural Access searches for the parameter structure therein. The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***]. The service name and parameter structure name must be supplied, but the extension specifier and field name are optional. If the parameter or parameter structure are extensions to the standard service parameters, the literal text .x must be included in ***parmname***.

If the field name is not supplied in ***parmname***, Natural Access uses ***index*** as the field selector. The first field within the structure is indexed as zero (0), the second as one (1), and so on.

If ***parmname*** is NULL, ***parmid*** is used to retrieve the parameter structure and ***index*** is used as the field selector. The parameter IDs are defined in the service header file (for example, *vcedef.h*).

The IDs are usually of the following form:
SERVICENAME[***_X***]***_STRUCTNAME_PARMID***.

The service name is the same name referenced in **ctaInitialize** and **ctaOpenServices**. If the structure references non-standard service extension parameters, the ***_X*** is used to distinguish these non-standard structures. An example of a standard structure ***parmid*** is VCE_PLAY_PARMID.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

The returned CTA_PARM_INFO structure contains the following fields:

Field	Description
structname	The parameter's structure name. Use when using ***parmid***.
fieldname	The parameter's field name.
offset	The value's offset in the parameter structure.
size	The parameter's value size (bytes).
format	The parameter's value format. Refer to the format table.
units	The parameter's value units. Refer to the unit table.

The following formats are supported:

Format	Description
CTA_FMT_WORD	Field is a 16-bit unsigned quantity.
CTA_FMT_DWORD	Field is a 32-bit unsigned quantity.
CTA_FMT_INT16	Field is a 16-bit signed quantity.
CTA_FMT_INT32	Field is a 32-bit signed quantity.
CTA_FMT_STRING	Field is a NULL-terminated string.
CTA_FMT_UNKNOWN	Field contains an unknown format.

The following units are supported:

Format	Description
CTA_UNITS_INTERNAL	No engineering units available.
CTA_UNITS_INTEGER	Field is a count or an index.
CTA_UNITS_COUNT	Field is a number of something.
CTA_UNITS_MASK	Field is a bit mask.
CTA_UNITS_HZ	Field is a measurement of frequency, in Hz.
CTA_UNITS_MS	Field is measured in milliseconds.
CTA_UNITS_DB	Field is a relative signal level, measured in dB.
CTA_UNITS_DBM	Field is an absolute signal level.
CTA_UNITS_IDU	Field is in internal DSP units (NMS internal representation for amplitude).
CTA_UNITS_STRING	Field contains a string.
CTA_UNITS_PERCENT	Field contains a percentage (where 100 is 100 percent).

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParms**, **ctaSetEventSources**

Example

```
void myShowParmInfo( char *parmname )
{
    CTA_PARM_INFO info;
    BYTE temp[100];     /* temp storage for parm value */
    union
    {
        WORD  w;
        DWORD W;
        INT16 i;
        INT32 I;
        char  s[80];
    } *pdata =( void* ) temp;
    char *punits;

    CTAHD void_ctahd;
    /* Server name descriptor  */
    char server_desc[] = "host.nmss.com:2244";
    /* Create a void context handle */
    ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

    if( ctaGetParmInfoEx( void_ctahd, 0, parmname, 0, &info ) != SUCCESS
        || ctaGetParmByName(NULL_CTAHD, parmname, &temp, sizeof( temp) )
        != SUCCESS )
        {
            /* display error */
            return;
        }

    switch( info.units )
        {
        case CTA_UNITS_INTERNAL: punits = "Internal";     break;
        case CTA_UNITS_INTEGER : punits = "Integer";      break;
        case CTA_UNITS_COUNT   : punits = "Count";        break;
        case CTA_UNITS_MASK    : punits = "Mask";         break;
        case CTA_UNITS_HZ      : punits = "Hz";           break;
```

```
        case CTA_UNITS_MS      : punits = "ms";           break;
        case CTA_UNITS_DB      : punits = "dB";           break;
        case CTA_UNITS_DBM     : punits = "dBm";          break;
        case CTA_UNITS_IDU     : punits = "Internal DSP"; break;
        case CTA_UNITS_STRING  : punits = "String";       break;
        case CTA_UNITS_PERCENT : punits = "Percent";      break;
        default                : punits = "Undefined";    break;
        }



    switch( info.format )
        {
        case CTA_FMT_WORD:
            printf( "%s.%-20s = %5u\t\t# (0x%04x) WORD  (%s)\n",
                info.structname, info.fieldname, pdata->w, pdata->w, punits );
            break;
        case CTA_FMT_DWORD:
            printf( "%s.%-20s = %5u\t\t# (0x%04x) DWORD (%s)\n",
                info.structname, info.fieldname, pdata->W, pdata->W, punits );
            break;
        case CTA_FMT_INT16:
            printf( "%s.%-20s = %5d\t\t# (0x%04x) INT16 (%s)\n",
                info.structname, info.fieldname, pdata->i, pdata->i, punits );
            break;
        case CTA_FMT_INT32:
            printf( "%s.%-20s = %5d\t\t# (0x%04x) INT32 (%s)\n",
                info.structname, info.fieldname, pdata->I, pdata->I, punits );
            break;
        case CTA_FMT_STRING:
            printf( "%s.%-20s = \"%s\"\t\t# STRING[%d]\n",
                info.structname, info.fieldname, pdata->s, info.size );
            break;
        default:
            printf( "Error! Unknown data type: '%c'\n", info.format );
            break;
        }
        return;
}
```

Sample run

For the parameter vce.play.gain, the output is:

```
vce.play.gain                       0                       # (0x0000) INT32 (dB)
```

ctaGetParms

Returns parameter values for a specified parameter structure.

Prototype

DWORD **ctaGetParms** (CTAHD ***ctahd***, unsigned ***parmid***, void ****buffer***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmid	Parameter structure ID.
buffer	Pointer to a buffer to receive parameter values.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***parmid*** is zero (0), or ***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is less than the parameter size.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmid*** is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParms retrieves the parameter values for the structure identified by its ***parmid***. If ***ctahd*** is a valid context, this function retrieves the parameters from the context defaults.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

If a void context handle is used for ***ctahd***, this function retrieves parameters from the global defaults and is executed on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

The ***size*** argument specifies the amount of available memory at the ***buffer*** and must be large enough to contain the returned parameter structure. The parameter structure's size can be retrieved using **ctaGetParmInfo**.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmsEx**, **ctaSetEventSources**

Example

```
void DemoRecordFile(DEMOCONTEXT *cx, char *filename, unsigned encoding)
{
    VCEHD            vh;
    CTA_EVENT        event;
    VCE_RECORD_PARMS rparms;
    unsigned         maxtime = 0;  /* no cap on length of recording */

    /*
     * Record the caller, overriding default parameters for
     * quick recognition of silence once voice input has begun.
     */
    ctaGetParms( cx->ctahd, VCE_RECORD_PARMID, &rparms, sizeof(rparms) );

    rparms.silencetime = 500; /* 1/2 sec */

    vceOpenFile ( cx->ctahd, filename, VCE_FILETYPE_FLAT, VCE_PLAY_RECORD,
                  encoding, &vh );

    vceRecordMessage( vh, 0, maxtime, &rparms);

    /* Wait for recording to end */
    do
    {
        ctaWaitEvent( cx->ctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while ( event.id != VCEEVN_RECORD_DONE );

    printf ( "Record ended with reason: val=0x%x\n", event.value );

    vceClose (vh);
}
```

ctaGetParmsEx

Returns parameter values for a specified parameter structure from a particular Natural Access Server.

Prototype

DWORD **ctaGetParmsEx** (char ***addr***, CTAHD ***ctahd***, unsigned ***parmid***, void ***buffer***, unsigned ***size***)

Argument	Description
addr	Natural Access Server address.
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmid	Parameter structure ID.
buffer	Pointer to a buffer to receive parameter values.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***parmid*** is zero (0) or ***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is less than the parameter size.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmid*** is not found.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetParmsEx retrieves the parameter values for the structure identified by its ***parmid***. If ***ctahd*** is a valid context, this function retrieves the parameters from the server associated with the context.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle.

If a void context handle is used for ***ctahd***, **ctaGetParmsEx** retrieves parameters from the global defaults and is executed on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

If the context handle is equal to NULL_CTAHD, the function retrieves parameters from the server specified by ***addr***. If ***addr*** is equal to NULL or is an empty string, the function retrieves parameters from the default server.

The ***size*** argument specifies the amount of available memory at the ***buffer*** and must be large enough to contain the returned parameter structure. The parameter structure's size can be retrieved using **ctaGetParmInfo**.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetParmByName, **ctaGetParmByNameEx**, **ctaGetParmID**, **ctaGetParmIds**, **ctaGetParmIdsEx**, **ctaGetParms**, **ctaSetParmByName**, **ctaSetParmByNameEx**

Example

```
void DemoRecordFile(DEMOCONTEXT *cx, char *filename, unsigned encoding)
{
    VCEHD            vh;
    CTA_EVENT        event;
    VCE_RECORD_PARMS rparms;
    unsigned         maxtime = 0;  /* no cap on length of recording */

    /*
     * Record the caller, overriding default parameters for
     * quick recognition of silence once voice input has begun.
     */
    ctaGetParmsEx( "localhost", cx->ctahd, VCE_RECORD_PARMID, &rparms,
          sizeof(rparms) );

    rparms.silencetime = 500; /* 1/2 sec */

    vceOpenFile ( cx->ctahd, filename, VCE_FILETYPE_FLAT, VCE_PLAY_RECORD,
                  encoding, &vh );

    vceRecordMessage( vh, 0, maxtime, &rparms);

    /* Wait for recording to end */
    do
    {
        ctaWaitEvent( cx->ctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while ( event.id != VCEEVN_RECORD_DONE );

    printf ( "Record ended with reason: val=0x%x\n", event.value );

    vceClose (vh);
}
```

ctaGetQueueHandle

Obtains the queue handle of the specified context.

Prototype

DWORD **ctaGetQueueHandle** (CTAHD ***ctahd***, CTAQUEUEHD ****ctaqueuehd***)

Argument	Description
ctahd	Context handle.
ctaqueuehd	Pointer to a returned queue handle associated with the context handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***ctaqueuehd*** is NULL.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaGetQueueHandle obtains the queue handle associated with the context.

Because there is overhead involved in calling **ctaGetQueueHandle**, use it mainly with the single context per queue model. Applications should save both the context and the queue handles to avoid calling this function on every call to **ctaWaitEvent**.

See also

ctaCreateContext

Example

```
DWORD NMSSTDCALL DemoErrorHandler( CTAHD ctahd, DWORD errorcode,
                                   char *errortext, char *func )
{
    if (ctahd == NULL_CTAHD)
    {
        printf( "\007Error in %s [ctahd=null]: %s (%#x)\n",
                func, errortext, errorcode );
    }
    else
    {
        CTAQUEUEHD ctaqueuehd;

        ctaGetQueueHandle(ctahd, &ctaqueuehd);

        printf( "\007Error in %s [ctahd=%#x, ctaqueuehd=%#x]: %s (%#x)\n",
                func, ctahd, ctaqueuehd, errortext, errorcode );
    }

    exit( errorcode );
}
```

ctaGetServiceVersion

Retrieves the version of the specified Natural Access service.

Prototype

DWORD **ctaGetServiceVersion** (CTA_SERVICE_NAME ***svcname***, CTA_REV_INFO ***pinfo***, unsigned ***size***)

Argument	Description

svcname	Pointer to a structure specifying a service. The CTA_SERVICE_NAME structure is:
	`typedef struct`
	`{`
	`  char *svcname;  /* Service name (eg. "ADI", "SWI" ...)*/`
	`  char *svcmgrname;  /* Service manager providing service  */`
	`} CTA_SERVICE_NAME;`

pinfo	Pointer to a buffer to receive the CTA_REV_INFO structure, as shown:
	`typedef struct`
	`{`
	`  DWORD size;  /* Size of the returned structure  */`
	`  DWORD majorrev;  /* Major revision of service  */`
	`  DWORD minorrev;  /* Minor revision of service  */`
	`  char builddate [12];  /* Build date, "Mmm dd yyyy\0"  */`
	`  DWORD compatlevel;  /* Compatibility level of API  */`
	`} CTA_REV_INFO;`
	See the Details section for field descriptions.

size	Size of the CTA_REV_INFO data structure.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***pinfo*** is NULL.
CTAERR_BAD_SIZE	***size*** is smaller than the size of DWORD.
CTAERR_NOT_FOUND	Specified service or service manager set is not initialized.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetServiceVersion performs programmatic checking of the installed and initialized service version. Use the information returned in the CTA_REV_INFO structure to check the minimum required service revision and compatibility level.

The CTA_REV_INFO structure contains the following fields:

Field	Description
size	Number of bytes written at the address pointed to by ***pinfo***.
majorrev	Major revision number of the initialized service. This value changes when a service release contains major enhancements such as new APIs.
minorrev	Minor revision number of the initialized service. This value changes when a service release contains minor enhancements, bug fixes, or both.
builddate	A NULL-terminated string indicating the build date of the service, in ***mm dd yyyy*** format.
compatlevel	Compatibility level of the specified service. The compatibility level changes when the release includes enhancements requiring an application to be recompiled because the changes are not backward compatible, for example, API signature change.

See also

ctaGetServiceVersionEx

Example

```
int CheckRequiredVersion( CTA_SERVICE_NAME *svcname, DWORD compatlevel )
{
    CTA_REV_INFO info;

    ctaGetServiceVersion( svcname, &info, sizeof info );
    if ( info.compatlevel < compatlevel )
    {
        return FAILURE;
    }
    printf( "(%s) service version (%d.%d) installed.\n", svcname->svcname,
            info.majorver, info.minorver );
    return SUCCESS;
}
```

ctaGetServiceVersionEx

Retrieves the version of the specified Natural Access service from a specified server.

Prototype

DWORD **ctaGetServiceVersionEx** (CTAHD ***ctahd***, CTA_SERVICE_NAME ***svcname***, CTA_REV_INFO ***pinfo***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.

svcname	Pointer to a structure specifying a service. The CTA_SERVICE_NAME structure is:
	`typedef struct`
	`{`
	`  char *svcname;  /* Service name (eg. "ADI", "SWI" ...)*/`
	`  char *svcmgrname;  /* Service manager providing service */`
	`} CTA_SERVICE_NAME;`

pinfo	Pointer to a buffer to receive the CTA_REV_INFO structure, as shown:
	`typedef struct`
	`{`
	`  DWORD size;  /* Size of the returned structure */`
	`  DWORD majorrev;  /* Major revision of service */`
	`  DWORD minorrev;  /* Minor revision of service */`
	`  char builddate [12];  /* Build date, "Mmm dd yyyy\0" */`
	`  DWORD compatlevel;  /* Compatibility level of API */`
	`} CTA_REV_INFO;`
	See the Details section for field descriptions.

size	Size of the CTA_REV_INFO data structure.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***pinfo*** is NULL.
CTAERR_BAD_SIZE	***size*** is smaller than the size of DWORD.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_NOT_FOUND	Specified service or service manager set is not initialized.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetServiceVersionEx retrieves the version of an initialized service. ***ctahd*** can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

ctaGetServiceVersionEx performs programmatic checking of the installed and initialized service version. Use the information returned in the CTA_REV_INFO structure to check the minimum required service revision and compatibility level.

The CTA_REV_INFO structure contains the following fields:

Field	Description
size	Number of bytes written at the address pointed to by ***pinfo***.
majorrev	Major revision number of the initialized service. This value changes when a service release contains major enhancements such as new APIs.
minorrev	Minor revision number of the initialized service. This value changes when a service release contains minor enhancements, bug fixes, or both.
builddate	A NULL-terminated string indicating the build date of the service, in ***mm dd yyyy*** format.
compatlevel	Compatibility level of the specified service. The compatibility level changes when the release includes enhancements that require an application to be re-compiled because the changes are not backward compatible, for example, API signature change.

Example

```
int CheckRequiredVersion( CTA_SERVICE_NAME *svcname, DWORD compatlevel )
{
    CTA_REV_INFO info;

    CTAHD void_ctahd;
    /* Server name descriptor  */
    char server_desc[] = "host.nmss.com:2244";
    /* Create a void context handle */
    ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

    ctaGetServiceVersionEx( void_ctahd, svcname, &info, sizeof info );
    if ( info.compatlevel < compatlevel )
    {
        return FAILURE;
    }
    printf( "(%s) service version (%d.%d) installed.\n", svcname->svcname,
            info.majorver, info.minorver );
    return SUCCESS;
}
```

ctaGetText

Returns the text representation of an error, reason, event, or command code.

Prototype

DWORD **ctaGetText** (CTAHD ***ctahd***, unsigned ***code***, char ****buffer***, unsigned ***size***)

Argument	Description
ctahd	Context handle. Use a void context handle if the text is not associated with a context or a context is not available.
code	Natural Access code (either a general code or a code defined by a Natural Access service).
buffer	Pointer to a buffer to receive the text string. ***buffer*** should contain a minimum of 40 characters.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is too small to receive the text.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	Service to translate ***code*** is not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetText returns a text string in ***buffer*** corresponding to the Natural Access error, reason, event, or command code.

Refer to *Handling Natural Access errors* on page 55 for more information.

See also

ctaFormatEvent, **ctaGetTextEx**

Example

```
void DemoShowError( char *preface, DWORD errcode )
{
    char text[40];

    ctaGetText( NULL_CTAHD, errcode, text, sizeof(text) );
    printf( "\t%s: %s\n", preface, text );
}
```

ctaGetTextEx

Returns the text representation of an error, reason, event, or command code from a particular Natural Access Server.

Prototype

DWORD **ctaGetTextEx** (char ***addr***, CTAHD ***ctahd***, unsigned ***code***, char ****buffer***, unsigned ***size***)

Argument	Description
addr	Natural Access Server address.
ctahd	Context handle. Use a void context handle if the text is not associated with a context or a context is not available.
code	Natural Access code (either a general code or a code defined by a Natural Access service).
buffer	Pointer to a buffer to receive the text string. ***buffer*** should contain a minimum of 40 characters.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***size*** is too small to receive the text.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	Service to translate ***code*** is not found.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetTextEx returns a text string in ***buffer*** corresponding to the Natural Access error, reason, event, or command code. If ***ctahd*** is a valid context, this function retrieves the text string from the server associated with the context.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle.

If a void context handle is used for ***ctahd***, **ctaGetTextEx** retrieves the text string from the global defaults and executes on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

If the context handle is equal to NULL_CTAHD, the function retrieves the text string from the server specified by ***addr***. If ***addr*** is equal to NULL or is an empty string, the function retrieves the text string from the default server.

The ***size*** argument specifies the amount of available memory at the ***buffer*** and must be large enough to contain the returned text string.

Refer to *Handling Natural Access errors* on page 55 for more information.

See also

ctaFormatEvent, **ctaGetText**

Example

```
void DemoShowError( char *preface, DWORD errcode )
{
    char text[40];

    ctaGetTextEx( "localhost", NULL_CTAHD, errcode, text, sizeof(text) );
    printf( "\t%s: %s\n", preface, text );
}
```

ctaGetTimeStamp

Converts an event timestamp to a count of the seconds elapsed since January 1, 1970.

Prototype

DWORD **ctaGetTimeStamp** (DWORD ***msgtime***, unsigned long ****timesec***, unsigned ****timems***)

Argument	Description
msgtime	The event timestamp.
timesec	Pointer to the returned seconds.
timems	Pointer to the returned milliseconds.

Return values

SUCCESS

Details

ctaGetTimeStamp converts an event timestamp to a count of the number of seconds elapsed since 00:00:00 January 1, 1970. The ***msgtime*** is the CTA_EVENT timestamp value, which is in millisecond units. This function converts the ***msgtime*** into ***timesec*** seconds and ***timems*** milliseconds since midnight 1/1/70.

Note: Because the event timestamp is 32 bits, it wraps every 2^{32} milliseconds (about 49 days). **ctaGetTimeStamp** assumes the event occurred within 24 days.

Example

```
#include <time.h>

void myShowTime( CTA_EVENT *event )
{
struct tm *ptime;
unsigned long timesec;
unsigned timems;

ctaGetTimeStamp( event->timestamp, &timesec, &timems );
ptime = localtime( &timesec );
printf( "%02d:%02d:%02d.%03d\n",
ptime->tm_hour, ptime->tm_min, ptime->tm_sec, timems );
}
```

ctaGetVersion

Returns the Natural Access version, build date, and compatibility level.

Prototype

DWORD **ctaGetVersion** (CTA_REV_INFO ***pinfo***, DWORD ***size***)

Argument	Description

pinfo	Pointer to the memory area. The CTA_REV_INFO structure is:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD majorrev;`
	`  DWORD minorrev;`
	`  char  builddate [12];`
	`  DWORD compatlevel;`
	`} CTA_REV_INFO;`
	See the Details section for a description of these fields.

size	Size of the data structure.

Return values

Return value	Description
CTAERR_BAD_SIZE	***size*** is not valid.
CTAERR_BAD_ARGUMENT	***pinfo*** is NULL.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetVersion returns the Natural Access version, build date, and compatibility level.

The CTA_REV_INFO structure contains the following fields:

Field	Description
size	Number of bytes written at the address pointed to by ***pinfo***.
majorrev	Major revision number of Natural Access. This value changes when a Natural Access release contains major enhancements such as new APIs or new services.
minorrev	Minor revision of Natural Access. This value changes when a Natural Access release contains minor enhancements, bug fixes, or both.
builddate	A NULL-terminated string indicating the build date of the revision, in ***mm dd yyyy*** format.
compatlevel	Compatibility level of the specified library. The compatibility level changes when the release includes enhancements that require applications to be recompiled because the changes are not backward compatible.

Refer to *Verifying compatibility* on page 57 for more information.

See also

ctaGetVersionEx

Example

```
/* Abort execution if NMS delivers version with compatibility level changed: */

void DemoCheckVersion( void )
{
    CTA_REV_INFO info;

    ctaGetVersion( &info, sizeof(info) );
    if ( info.compatlevel != CTA_COMPATLEVEL )
    {
        printf("Natural Access compatibility level does not match.
        Re-compile.\n");
        exit(-1);
    }
}
```

ctaGetVersionEx

Returns the Natural Access Server version, build date, and compatibility level.

Prototype

DWORD **ctaGetVersionEx** (CTAHD ***ctahd***, CTA_REV_INFO ***pinfo***, DWORD ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.

pinfo	Pointer to the memory area. The CTA_REV_INFO structure is:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD majorrev;`
	`  DWORD minorrev;`
	`  char  builddate [12];`
	`  DWORD compatlevel;`
	`} CTA_REV_INFO;`
	See the Details section for a description of these fields.

size	Size of the data structure.

Return values

Return value	Description
CTAERR_BAD_SIZE	***size*** is not valid.
CTAERR_BAD_ARGUMENT	***pinfo*** is NULL.
CTAERR_INVALID_CTAHD	Context handle is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaGetVersionEx returns the Natural Access Server version, build date, and compatibility level.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

The CTA_REV_INFO structure contains the following fields:

Field	Description
size	Number of bytes written at the address pointed to by ***pinfo***.
majorrev	Major revision number of Natural Access. This value changes when a Natural Access release contains major enhancements such as new APIs or new services.
minorrev	Minor revision of Natural Access. This value changes when a Natural Access release contains minor enhancements, bug fixes, or both.
builddate	A NULL-terminated string indicating the build date of the revision, in ***mm dd yyyy*** format.
compatlevel	Compatibility level of the specified server. The compatibility level changes when the release includes enhancements that require applications to be recompiled because the changes are not backward compatible.

Refer to *Verifying compatibility* on page 57 for more information.

Example

```
/* Abort execution if NMS delivers version with compatibility level changed: */

void DemoCheckVersion( void )
{
    CTA_REV_INFO info;

    CTAHD void_ctahd;
    /* Server name descriptor  */
    char server_desc[] = "host.nmss.com:2244";
    /* Create a void context handle */
    ctaCreateContext(NULL_CTAQUEUEHD, 0, server_desc, &void_ctahd);

    ctaGetVersionEx( void_ctahd, &info, sizeof(info) );

    if ( info.compatlevel != CTA_COMPATLEVEL )
    {
        printf("Natural Access compatibility level does not match.
        Re-compile.\n");
        exit(-1);
    }
}
```

ctaInitialize

Establishes a list of services available to a Natural Access application.

Prototype

DWORD **ctaInitialize** (CTA_SERVICE_NAME ***svcname[]**, unsigned ***nsvcs***, CTA_INIT_PARMS ***initparms***)

Argument	Description

svcname	Pointer to an array of service names to make available to the application. The CTA_SERVICE_NAME structure is:
	`typedef struct`
	`{`
	`  char *svcname;  /* Name of service  */`
	`  char *svcmgrname;  /* Name of service manager  */`
	`} CTA_SERVICE_NAME;`

nsvcs	Number of services in ***svcname***.

initparms	Pointer to a structure containing initialization parameters. The CTA_INIT_PARMS structure is:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD parmflags;`
	`  DWORD traceflags;`
	`  DWORD daemonflags;`
	`  DWORD ctaflags;`
	`  char  *filename;`
	`  DWORD ctacompatlevel;`
	`  DWORD reserved;`
	`} CTA_INIT_PARMS;`
	See Details for a description of these fields.

Return values

Return value	Description
SUCCESS	
CTAERR_ALREADY_DEFINED	Duplicate service or service manager exists in ***svcname.***
CTAERR_ALREADY_INITIALIZED	A Natural Access service was already initialized.
CTAERR_BAD_SIZE	***initparms*** size field is not valid.
CTAERR_FILE_NOT_FOUND	The file name specified in CTA_INIT_PARMS is not found or *cta.cfg* is not found.
CTAERR_FUNCTION_ACTIVE	Two separate threads called **ctaInitialize** at the same time. The first thread executed the function correctly. The second thread received this error.
CTAERR_INCOMPATIBLE_PARMS	The same services provided by two or more service managers have different standard parameters.
CTAERR_INCOMPATIBLE_REVISION	One of the specified services or ***svcnames*** is incompatible with the Natural Access dispatcher.
CTAERR_INVALID_LIB	The specified action (usually setting the ctaflags bits) could not be performed because the application is not linked to the appropriate Natural Access library.
CTAERR_LOAD_LIB	An error occurred while loading the Natural Access core library.
CTAERR_NOT_FOUND	One or more services are not found or listed services are not found in the configuration file.
CTAERR_SHAREMEM_ACCESS	One of the following conditions occurred: • Shared system parameters or tracing is enabled but *ctdaemon* is not running. • CTA_PARM_MGMT_SHARED bit is set in parmflags but one or more of the specified services are not being managed within *ctdaemon*. Update *cta.cfg* to contain all required services and restart *ctdaemon.*
CTAERR_WRONG_COMPAT_LEVEL	Installed Natural Access compatibility level is different from the application-specified compatibility level.

Details

ctaInitialize finds all service managers and services that the application needs. Call this function only once at application initialization.

You can register services in one of the following ways:

- Invoke **ctaInitialize** and specify the service and service manager names of all the services you plan to use. The application can use only the services specified in the call to **ctaInitialize**.
- Invoke **ctaInitialize**, pass NULL for ***svcname***, and pass a user-defined Natural Access configuration file name. *cta.cfg* is the default configuration file name if a file name is not specified. **ctaInitialize** looks in the [ctasys] header section of the configuration file for the list of service and service manager names.

Note: If you want to use a remote server but pass NULL for ***svcname*** and pass the configuration file name *cta.cfg*, *cta.cfg* on the local server may not specify the same available services as the remote server.

After calling **ctaInitialize**, you can modify and retrieve global default parameters, and create and destroy event queues and contexts.

The CTA_INIT_PARMS structure contains the following fields:

Field	Description
size	Indicates the size of the passed structure.
parmflags	Specifies where the global default parameters are stored. Valid values are: • CTA_PARM_MGMT_LOCAL (default) Global default parameters are stored in process local memory. • CTA_PARM_MGMT_SHARED Global default parameters are stored in shared system memory. *ctdaemon* must be running and the shared memory created. An error is returned if *ctdaemon* is not running. • CTA_PARM_MGMT_SHARED_IF_PRESENT Global default parameters are stored in shared system memory and an error is not returned if *ctdaemon* is not running. If this setting is chosen and *ctdaemon* is not running, the default parameter management setting is local to the process. If you are using a local or remote server, the system ignores these flags. Parameter management applies to the entire system and cannot be overridden. Refer to *Global default parameters* on page 51 for more information on parameter storage.
traceflags	Provides a backlog of trace information on a context when an error condition occurs. To observe trace information for all contexts, do not enable this feature. • Set the CTA_TRACE_ENABLE bit to enable tracing. *ctdaemon* must be running and maintaining the system-wide trace buffer for all Natural Access applications. Since an error is not returned if *ctdaemon* crashes, traceflags cannot be used to check *ctdaemon.* • Set the CTA_TRACE_ENABLE and CTA_TRACE_ON_ERROR bits to enable the backtracing on error option. To write all recently logged trace messages on a particular context to *ctdaemon*, call **ctaLogTrace**, specifying the context you want to analyze, and set the ***traceseverity*** argument to CTA_TRACE_SEVERITY_ERROR. If you are using a local or remote server, the system ignores these flags.
daemonflags	Set to zero (0).

Field	Description
ctaflags	Specifies the context and queue settings. Valid values are: • CTA_NOTIFY_UPDATE_WAITOBJS (0x1) Natural Access sends CTAEVN_UPDATE_WAITOBJS when there is a change to the internal array of Natural Access wait objects. Set this flag when wait objects are managed by the application. Do not set this flag if wait objects are managed by Natural Access. Refer to **ctaRegisterWaitObject** to allow Natural Access to manage the application wait objects. • CTA_COMMS_SERIALIZE (0x2)(default) Commands from all queues sent to the server are serialized. This mode requires fewer system resources, allowing more event queues to be created. This flag is ignored if the application is using Natural Access functionality in the process address space (inproc). **Note:** When this bit is set under UNIX, a process cannot fork. Each process must create its own session to the server. • CTA_COMMS_NOTSERIALIZE (0x4) Commands from all queues sent to the server are not serialized. This mode requires more system resources, limiting the number of event queues that can be created. This flag is ignored if the application is using Natural Access functionality in the process address space (inproc). • CTA_MODE_LIBRARY (0x8) The Natural Access Server functionality runs in the process address space. Contexts created using this flag are specific to the application and cannot be shared. This flag corresponds to the DefaultServer=inproc parameter in *cta.cfg*. • CTA_MODE_SERVER (0x10) The Natural Access Server functionality runs in a Natural Access Server (*ctdaemon*) that is separate from the application, but on the same host. Contexts created using this flag can be shared. This flag corresponds to the DefaultServer=localhost parameter in *cta.cfg*. • CTA_CONTEXTNAME_REQUIRE_UNIQUE (0x20) Natural Access cannot modify the context name to ensure uniqueness. If two applications specify the same context name, the second application receives an error. This flag is ignored if the application is using Natural Access functionality in the process address space (inproc). • CTA_CONTEXTNAME_CREATE_UNIQUE (0x40) The local server can modify context names specified by client applications so that context names are not duplicated (default). This flag is ignored if the application is using Natural Access functionality in the process address space (inproc).
filename	Determines the file to use to obtain the service names. • To use a user-defined file, pass NULL for ***svcname*** and pass the user-defined file name. **ctaInitialize** looks in the [ctasys] header section of the specified file for the service names. If there is a spelling error in the [ctasys] section of the file, the service and service manager are not initialized. • To use *cta.cfg*, pass NULL for ***svcname*** and NULL for filename. **ctaInitialize** looks in the [ctasys] header section of *cta.cfg* for the service names. • If you pass a ***svcname*** and a file name, the file name is ignored. • If filename does not contain a path, the path specified in the environment variable CTA_DPATH is searched.

Field	Description
ctacompatlevel	Verifies Natural Access compatibility. Valid values are: • CTA_COMPATLEVEL Verifies Natural Access compatibility. • Zero (0) Omits compatibility checking. The compatibility level is changed if the release includes changes requiring applications to be recompiled. Refer to *Verifying compatibility* on page 57 for more information.
reserved	Reserved for future use. Set this value to zero (0).

Refer to *Initializing Natural Access applications* on page 35 for more information.

Refer to *Natural Access configuration file* on page 31 for information on *cta.cfg*. Refer to *Natural Access Server: ctdaemon* on page 238 for more information on *ctdaemon*.

See also

ctaQueryWaitObjects, **ctaSetTraceLevel**

Example

```
static CTA_SERVICE_NAME InitServices[] = { { "ADI", "ADIMGR" },
                                           { "SWI", "SWIMGR" },
                                           { "VCE", "VCEMGR" } };

void DemoInitialize()
{
    DWORD ret;
    CTA_INIT_PARMS initparms = { 0 };

    /* Initialize size of init parms structure */
    initparms.size = sizeof(CTA_INIT_PARMS);

    /* Use process global default parameters */
    initparms.parmflags |= CTA_PARM_MGMT_LOCAL;

    /* If daemon running then initialize tracing */
    initparms.traceflags = CTA_TRACE_ENABLE;

    /* Notify when Natural Access changes its internal list of wait objects*/
    initparms.ctaflags = CTA_NOTIFY_UPDATE_WAITOBJS;

    /* Verify Natural Access compatibility */
    initparms.ctacompatlevel = CTA_COMPATLEVEL;

    /* Initialize Natural Access */
    ctaInitialize(InitServices,
                  sizeof(InitServices)/sizeof(InitServices[0]),
                  &initparms);
}
```

Note: For QX boards, replace ADIMGR with QDIMGR.

ctaLoadParameterFile

Loads and changes parameter default values.

Prototype

DWORD **ctaLoadParameterFile** (CTAHD ***ctahd***, char ****filename***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
filename	Pointer to the full or partial file name of the file. If a path is specified, only that path is searched. If the ***filename*** is specified without a path, the path specified in the CTA_DPATH environment variable is searched.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***filename*** is NULL or is too long.
CTAERR_FILE_NOT_FOUND	***filename*** is not found.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_SYNTAX	Syntax error occurred in the parameter file. To debug the error, use *ctdaemon.*
CTAERR_NOT_FOUND	Specified parameter name is not found in the file. To debug the error, use *ctdaemon.*
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaLoadParameterFile loads and changes parameter default values specified in ***filename*** to the specified ***ctahd***. Default values can be changed on a context specific-basis or on an application-wide basis anytime after a service is initialized, the event queue and the context are created, and services are opened.

The file parameter default syntax is: ***ServiceName.ParameterName.FieldName = Value***.

For example:

```
adi.play.speed = 125
adi.play.gain = 50
```

If a file contains a section header, default parameter overrides must be preceded by [ctapar]. Use this value for placing parameter overrides in *cta.cfg*. For example:

```
[ctasys]
Service = adi, adimgr
Service = swi, swimgr
Service = vce, vcemgr
[ctapar]
adi.play.speed = 125
adi.play.gain = 50
```

If a file does not contain a section header, the entire file is considered to be a default parameter override file. For example:

```
adi.play.mask = 0x0FC0
adi.play.gain = 50
adi.play.speed = 125
adi.play.maxspeed = 200
```

Refer to *Natural Access configuration file* on page 31 for more information about *cta.cfg*. If ***ctahd*** is a valid context, this function modifies the parameters from the context defaults.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

If a void context handle is used for ***ctahd***, this function retrieves parameters from the global defaults and executes on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

If the application needs to change default values on all contexts, call **ctaLoadParameterFile** with NULL as ***ctahd*** before calling **ctaCreateContext**. Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmID**, **ctaGetParmInfo**, **ctaGetParms**, **ctaRefreshParms**, **ctaSetEventSources**

Example

```
     [ctapar]
     vce.play.gain = 10
     vce.play.speed = 200
     vce.play.maxspeed = 200
```

myPlayVoxFileWithCfg plays the content of a VOX file at twice normal speed with 10 dB gain.

```
/*
*   play a message at the speed and
*   gain specified in filename.
*   Also wait for completion.
*/
extern CTAHD      CtaHd;
extern CTAQUEUEHD CtaQueueHd;

void myPlayVoxFileWithCfg( char *voxfile, DWORD encoding, char *cfgfile)
{
    VCEHD vh;
    CTA_EVENT event;
    unsigned message = VCE_ALL_MESSAGES;

    /* Change vce play defaults specified in filename */
    ctaLoadParameterFile( CtaHd,  cfgfile );

    vceOpenFile(CtaHd, voxfile, VCE_FILETYPE_VOX, VCE_PLAY_ONLY, encoding,
                                                                    &vh);
    vcePlayMessage (vh, message, NULL) ;
    do
    {
        ctaWaitEvent( CtaQueueHd, &event, CTA_WAIT_FOREVER);
    } while (event.id != VCEEVN_PLAY_DONE); /* Ignore other events */
}
```

ctaLogTrace

Logs a trace record.

Prototype

DWORD **ctaLogTrace** (CTAHD ***ctahd***, CTA_TRACE_SEVERITY ***traceseverity***, unsigned ***tracetag***, void ***tracevalue***, unsigned ***tracesize***)

Argument	Description
ctahd	Handle returned by **ctaCreateContext**. Use CTA_NULL_CTAHD if the trace message is not associated with a context.
traceseverity	Type of trace message. Valid values are: CTA_TRACE_SEVERITY_INFO CTA_TRACE_SEVERITY_WARNING CTA_TRACE_SEVERITY_ERROR
tracetag	Type of parameter passed by reference in ***tracevalue***. Valid values are: CTA_TRACETAG_API_EVENT CTA_TRACETAG_DWORD CTA_TRACETAG_INT32 CTA_TRACETAG_STRING CTA_TRACETAG_ERROR
tracevalue	Pointer to a location containing the byte values to be written to the log file.
tracesize	Size of ***tracevalue*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_OUT_OF_MEMORY	Tracing is enabled and there is not enough memory for the trace messages.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaLogTrace logs errors, warnings, or informational messages to the Natural Access trace buffer maintained by the Natural Access Server (*ctdaemon*). Enable tracing by setting ***traceflags*** to CTA_TRACE_ENABLE in **ctaInitialize**.

ctdaemon allows viewing trace messages and logging to a file. *ctdaemon* translates trace records received from applications into a readable format based on the ***tracetag*** passed to this function.

Refer to *Error tracing* on page 56 for more information.

See also

ctaSetTraceLevel

Example

```
void DemoLogString(CTAHD ctahd, char *string)
{
    ctaLogTrace(ctahd,
                CTA_TRACE_SEVERITY_INFO,
                CTA_TRACETAG_STRING,
                string, strlen(string) + 1);
}

void DemoLogAppEvent(CTA_EVENT *event)
{
    ctaLogTrace(event->ctahd,
                CTA_TRACE_SEVERITY_INFO,
                CTA_TRACETAG_API_EVENT,
                event, sizeof(*event));
}
```

ctaOpenServices

Opens one or more services on the specified context.

Prototype

DWORD **ctaOpenServices** (CTAHD ***ctahd***, CTA_SERVICE_DESC ***svclist[]***, unsigned ***nsvcs***)

Argument	Description
ctahd	Context handle.

svclist	List of services to open on the context. The CTA_SERVICE_DESC structure is
	`typedef struct`
	`{`
	`  CTA_SERVICE_NAMEname;`
	`  CTA_SERVICE_ADDRsvcaddr;`
	`  CTA_SERVICE_ARGSsvcargs;`
	`  CTA_MVIP_ADDR mvipaddr;`
	`} CTA_SERVICE_DESC;`
	The CTA_SERVICE_NAME structure is:
	`typedef struct`
	`{`
	`  char *svcname;`
	`  char *svcmgrname;`
	`} CTA_SERVICE_NAME;`
	The CTA_SERVICE_ADDR structure is reserved and should be initialized to zero (0).
	The CTA_SERVICE_ARGS structure is:
	`typedef struct`
	`{`
	`  charstrarg [40];`
	`  DWORD args [10];`
	`} CTA_SERVICE_ARGS;`
	The CTA_MVIP_ADDR structure is:
	`typedef struct`
	`{`
	`  DWORD board;`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`  DWORD mode;`
	`} CTA_MVIP_ADDR;`

nsvcs	Number of services in ***svclist***.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_SEQUENCE	An open or close service is pending on the context.
CTAERR_NOT_FOUND	One of the following conditions occurred: • One or more of the services in ***svclist*** are not registered with Natural Access. • Application requested a service on a local or remote server that is not specified in the server configuration file (for example, *cta.cfg*). • Service manager is not attached to the event queue.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SERVICE_IN_USE	One or more of the services are already open on the context.
CTAERR_SVR_COMM	Server communication error.

Events

CTAEVN_OPEN_SERVICES_DONE

The event value field contains the open services completion status. If the value is CTA_REASON_FINISHED, all services successfully opened.

If the event value field is not CTA_REASON_FINISHED, one of the services failed to open. The resources associated with all of the services listed in **svclist** are de-allocated and the context is returned to the state it was in just before the call to **ctaOpenServices**.

When opening the ADI service, the following errors can be returned in the event value field:

Error	Description
ADIERR_CANNOT_CREATE_CHANNEL	Out of resources on board.
CTAERR_BAD_ARGUMENT	Invalid MVIP stream, timeslot, or mode.
CTAERR_INVALID_BOARD	Invalid board number. The board was not configured by the configuration utility.
CTAERR_OUTPUT_ACTIVE	DSP timeslot is in use.

Details

ctaOpenServices opens one or more services on the specified context (***ctahd***). If any service fails to open, all services successfully opened in this call are closed and a service-specific error is returned in the value field of the CTAEVN_OPEN_SERVICES_DONE event.

Note: Ensure that the services you open on a context are services that are specified in the server configuration file.

ctaOpenServices is asynchronous and returns immediately. When all services are successfully opened, a DONE event returns to the application by way of the application event queue.

See also

ctaAttachContext

Example

```
void DemoOpenPort(
    unsigned    userid,             /* for ctaCreateContext               */
    char       *contextname,        /* for ctaCreateContext               */
    CTA_SERVICE_DESC services[],    /* for ctaOpenServices                */
    unsigned    numservices,        /* number of "services"               */
    CTAQUEUEHD *pctaqueuehd,        /* returned Natural Access queue handle*/
    CTAHD      *pctahd)             /* returned Natural Access context hd   */
{
    CTA_EVENT event = { 0 };

    /* Open the Natural Access application queue, attaching all defined
     * service managers.
     */
    ctaCreateQueue( NULL, 0, pctaqueuehd );

    /* Create a Natural Access context */
    ctaCreateContext( *pctaqueuehd, userid, contextname, pctahd );

    /* Open services */
    ctaOpenServices( *pctahd, services, numservices );

    /* Wait for the service manager and services to be opened asynchronously */
    do
    {
        ctaWaitEvent( *pctaqueuehd, &event, CTA_WAIT_FOREVER );
    } while (event.id != CTAEVN_OPEN_SERVICES_DONE);

    if (event.value != CTA_REASON_FINISHED)
        printf ("Opening services failed: val=0x%x\n", event.value );
}

void DemoStartLine(DEMOCONTEXT *cx)
{
    char cxname[12];

    CTA_SERVICE_DESC services[] =        /* for ctaOpenServices */
    {   { {"ADI", "ADIMGR"}, { 0 }, { 0 }, { 0 } },
        { {"SWI", "SWIMGR"}, { 0 }, { 0 }, { 0 } },
        { {"VCE", "VCEMGR"}, { 0 }, { 0 }, { 0 } }
    };

    /* Fill in ADI service (index 0) MVIP address information */
    services[0].mvipaddr.board    = cx->ag_board;
    services[0].mvipaddr.stream   = cx->mvip_stream;
    services[0].mvipaddr.timeslot = cx->mvip_slot;
    services[0].mvipaddr.mode     = ADI_FULL_DUPLEX;

/* Context name will be printed in all trace records for this context */
    sprintf( cxname, "DEMOCX%04d", cx->line );

    DemoOpenPort( 0, "DEMOCONTEXT", services,
                  sizeof(services)/sizeof(services[0]),
                  &(cx->ctaqueuehd), &(cx->ctahd) );
}
```

Note: For QX boards, replace ADIMGR with QDIMGR.

ctaQueryServerContexts

Obtains a list of available contexts on a server.

Prototype

DWORD **ctaQueryServerContexts** (CTAHD ***ctahd***, CTA_CNXT_INFO ***buffer***, unsigned ***bufsize***, unsigned ****ncontexts***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.

buffer	Pointer to the CTA_CNXT_INFO structure for the context information:
	`typedef struct`
	`{`
	`  CTAHD ctahd;`
	`  CTAQUEUEHD ctaqueuehd;`
	`  char contextname[CTA_CONTEXT_NAME_LEN];`
	`  DWORD flags;`
	`  DWORD nclients;`
	`  DWORD nsvcs;`
	`} CTA_CNXT_INFO;`
	See the Details section for a description of these fields.

bufsize	Size of the CTA_CNXT_INFO array passed. Set to zero (0) to determine the number of contexts created on a Natural Access Server.
ncontexts	Pointer to the returned number of contexts.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid server address.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaQueryServerContexts obtains a list of contexts available on the Natural Access Server (*ctdaemon*) specified by ***ctahd***. CTA_CNXT_INFO is returned for every context. If the size of ***buffer*** is less than the current number of contexts on a server (but not zero), the maximum possible number of contexts are returned.

The returned CTA_CNXT_INFO structure contains the following fields:

Field	Description
ctahd	Server context handle.
ctaqueuehd	Server queue handle.
flags	Context flags.
nclients	Number of active clients attached to the context.
nsvcs	Number of services open on the context.

See also

ctaAttachContext, **ctaCreateContext**

Example

```
int DemoQueryContexts(CTAHD ctahd)
{
     unsigned size, count;
     DWORD ret;
     /* Obtain the number of contexts on the server */
     ret = ctaQueryServerContexts(ctahd,NULL,0,&size);
     if ( ret != SUCCESS )
          return ret;
     CTA_CNXT_INFO* buffer = (CTA_CNXT_INFO*)
          malloc(size * sizeof(CTA_CNXT_INFO));

     /* Retrieve the list of contexts */
     ret = ctaQueryServerContexts(ctahd,buffer,size,&count);
     if ( ret != SUCCESS )
          return ret;
     for (unsigned i = 0;i < count; i++)
            printf("Context %s with %d clients and %d services\n",

     buffer[i].contextname,buffer[i].nclients,buffer[i].nsvcs);
     free(buffer);
     return SUCCESS;
}
```

ctaQueryServices

Obtains a list of available services.

Prototype

DWORD **ctaQueryService** (CTAHD ***ctahd***, CTA_SERVICE_NAME ***svclist[]***, unsigned ***size***, unsigned ****count***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.

svclist	Pointer to a list of Natural Access service names. The CTA_SERVICE_NAME_BUF structure is:
	`typedef struct`
	`{`
	`  char svcname[CTA_MAXSVCNAME_LEN + 1];`
	`  char svcmgrname[CTA_MAXSVCNAME_LEN + 4];`
	`}  CTA_SERVICE_NAME_BUF;`

size	The size of the CTA_SERVICE_NAME_BUF array passed. Set to 0 (zero) to determine the number of services available on a Natural Access Server (*ctdaemon*).
count	Pointer to the returned number of service names.

Return values

Return value	Description
SUCCESS	
CTA_BAD_ARGUMENT	Invalid server address.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaQueryServices obtains a list of services available on the Natural Access Server (*ctdaemon*) specified by ***ctahd***. If the size of ***svclist*** is less than the current number of services on a server (but not zero), the maximum possible number of services are returned.

See also

ctaOpenServices

Example

```
int DemoQueryServices(CTAHD ctahd)
{
     unsigned size, count;
     DWORD ret;

     ret = ctaQueryServices(ctahd,NULL,0,&size);
     if ( ret != SUCCESS )
          return ret;

     CTA_SERVICE_NAME_BUF* svclist = (CTA_SERVICE_NAME_BUF*)
          malloc(size * sizeof(CTA_SERVICE_NAME_BUF));

     ret = ctaQueryServices(ctahd,svclist,size,&count);
     if ( ret != SUCCESS )
          return ret;

     for (unsigned i = 0;i< count; i++)
               printf("Service: %s, Service manager: %s\n",
                    svclist[i].svcname,svclist[i].svcmgrname);

     free(svclist);

     return SUCCESS;
}
```

ctaQueryWaitObjects

Obtains an operating-system-specific array of the wait objects that Natural Access is managing internally. This function is used when the application is managing wait objects.

Prototype

DWORD **ctaQueryWaitObjects** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ****waitobjs[]***, unsigned ***size***, unsigned ****count***)

Argument	Description
ctaqueuehd	Context handle.

waitobjs	Pointer to the CTA_WAITOBJ array that is filled in with the wait objects.
	For Windows, the type is:
	`typedef MUX_HANDLE CTA_WAITOBJ;`
	For UNIX, the type is:
	`struct pollhd { int fd; int events; int revents; } typedef struct pollhd CTA_WAITOBJ;`

size	The size of the CTA_WAITOBJ array passed. Set to zero (0) to determine the number of wait objects used by Natural Access.
count	Pointer to the returned number of the CTA_WAITOBJ array elements.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaQueryWaitObjects obtains an operating-system-specific array of the wait objects that Natural Access is managing internally. If the application is managing its own wait objects, the application needs to be notified about Natural Access wait objects so that the Natural Access wait objects and the application-specific wait objects can be passed to the operating system wait functions.

If the application is managing its own wait objects and waiting for operating system events, call **ctaQueryWaitObjects** immediately after **ctaCreateQueue**, and when the application receives CTAEVN_UPDATE_WAITOBJS. Applications receive CTAEVN_UPDATE_WAITOBJS only if they specify the CTA_NOTIFY_UPDATE_WAITOBJS flag in the ***ctaflags*** field of the ***initparms*** parameter to **ctaInitialize**. If this flag is not specified, the list of Natural Access wait objects may not be complete.

The application must manage all wait objects including those of Natural Access. This requires an application to:

- Dynamically create an array of wait objects to contain both Natural Access and application-specific wait objects.
- Identify when a Natural Access wait object is signaled and notify Natural Access accordingly.

Complete the following steps to create an array of wait objects:

Step	Action
1	Call **ctaQueryWaitObjects** with a ***size*** value of zero (0). This call returns the actual number of wait objects used by Natural Access in the ***count*** field without actually returning the wait objects themselves.
2	Allocate an array of application-maintained wait objects large enough to hold both the Natural Access wait objects and the application-specific wait objects.
3	Call **ctaQueryWaitObjects** again to retrieve the Natural Access wait objects. Place these wait objects at the beginning of the newly allocated array. The remaining wait objects are available for use by the application.

Complete the following steps to process signals on a wait object in the array:

Step	Action
1	Determine if you classified the wait object as a Natural Access wait object.
2	If so, immediately call **ctaWaitEvent** with a ***timeout*** of zero (0). Wait events may return with no event if the event is absorbed internally by Natural Access.

Note: Do not use **ctaQueryWaitObjects** if you register wait objects with **ctaRegisterWaitObject**.

Refer to *Using wait objects* on page 49 for more information.

See also

ctaUnregisterWaitObject

Example

```
#include "nmstypes.h"
#include "ctademo.h"
#include "ctadef.h"

CTA_WAITOBJ *waitobjs;
unsigned numwaitobjs;

void query_ctaccess_wait_objects(CTAQUEUEHD ctaqueuehd)
{
    DWORD ret;

    /* Find out how many Natural Access wait objects there are: */
    if ((ret = ctaQueryWaitObjects( ctaqueuehd, NULL, 0, &numwaitobjs))
        != SUCCESS)
    {
        exit( -1 );
    }
    else
    {
        unsigned size = sizeof(CTA_WAITOBJ)*numwaitobjs;

        /* Allocate one more than the number returned by Natural Access. The
           additional one (waitobjs[0]), will be used for the application's
           stdin wait object */
        waitobjs = (CTA_WAITOBJ *)malloc(size + sizeof(CTA_WAITOBJ));
        if (waitobjs == NULL)
            exit( -1 );

        /* Get the Natural Access wait objects in an array */
        if ((ret = ctaQueryWaitObjects( ctaqueuehd, &(waitobjs[1]), size,
                                        &numwaitobjs)) != SUCCESS)
        {
            exit( -1 );
        }
    }

    numwaitobjs++; /* Total number of wait objects include app wait object */
}
```

ctaQueueEvent

Enqueues an event to an event queue in the same process.

Prototype

DWORD **ctaQueueEvent** (CTA_EVENT ***ctaevt***)

Argument	Description

ctaevt	Pointer to the event structure to be enqueued. The CTA_EVENT structure is:
	`typedef struct`
	`{`
	`  DWORD  id;  /* Event code and source service ID */`
	`  CTAHD  ctahd;  /* Natural Access context handle */`
	`  DWORD  timestamp;  /* Timestamp */`
	`  DWORD  userid;  /* Userid(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 object handle */`
	`} CTA_EVENT;`

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaQueueEvent routes the event to the event queue of the context specified by the ctahd field in CTA_EVENT.

Applications can create their own events using the valid code identifiers of 0x20002000 through 0x20002FFF. Use the CTA_USER_EVENT macro to convert a user event code in the range of 0 through 0xFFF to a valid Natural Access event code.

See also

ctaWaitEvent

Example - sending

```
/* Code fragment for a program responsible for performing call control
*  on a Natural Access context. Media functions will be performed by another
*  program which attaches to a shared context, i.e., the one on which call
*  control is being performed by this program.
*/
CTAQUEUEHD ctaqueuehd = NULL_CTAQUEUEHD;
CTAHD      CallCtrlCtahd;
void ExchangeContextName( CTAHD ctahd, char *pNewCtxtName )
{
    unsigned  userid = 0;
    int       ret ;
    /* Attach to agreed upon context for exchange of application events */
    if ((ret = ctaAttachContext( ctaqueuehd, userid, "Common", &ctahd ))
         != SUCCESS)
    {
        printf("Unable to attach to shared context used with media program\n");
        exit(-1);
    }
    /* Create new context based upon the name provided to the application in the
    *  startup of the program, i.e., "pNewCtxName".  This is the new context
    *  on which call control and media functions will be performed.
    */
    if (ret = ctaCreateContextEx( ctaqueuehd, userid, pNewCtxtName,
                                  &CallCtrlCtahd, 0 ) != SUCCESS)
    {
        printf("Unable to create context %s\n", pNewCtxtName );
        exit(-1);
    }
    else
    {
        CTA_EVENT event = { 0 };

        /* Fill in the event to be sent to the media program. */
        event.id = APPEVN_EXCHANGE_NAME;
        event.ctahd = ctahd;
        event.value = SUCCESS;

        /* Send text buffer with the event */
        event.size = strlen( pNewCtxtName ) + 1;
        event.buffer = pNewCtxtName;

        if ( (ret = ctaQueueEvent( &event )) != SUCCESS )
        {
            printf("ctaQueueEvent failed with 0x%x", ret );
            exit(-1);
        }
    }
}
```

Example - receiving

```
/*  Code fragment from a program responsible for performing media functions
*   on a Natural Access context shared with another program that performs call
*   control.
*/
CTAHD commCtahd = NULL_CTAHD,  mediaCtahd = NULL_CTAHD;
CTAQUEUEHD ctaqueuehd;
void SetupAndExchange()
{
    DWORD ret;

    /* Create agreed to context on which exchange of application events
    *  will occur.
    */
    if (ret = ctaCreateContextEx( ctaqueuehd, 0, "Common",
                                  &commCtahd, 0 ) != SUCCESS)
    {
        printf("Unable to create context %s\n", "Common" );
        exit(-1);
    }

    /* Wait for the call control program to attach to this "common" context
    *  and send an event message indicating the name of the context on which
    *  media should be played.
    */
    for(;;)
    {
        CTA_EVENT   event = { 0 };
        int         ctaOwnsBuf = 0;

        event.ctahd = commCtahd;
        if((ret = ctaWaitEvent( ctaqueuehd, &event,
                                CTA_WAIT_FOREVER )) != SUCCESS )
        {
            printf("ctaWaitEvent returns 0x%x -- exiting\n", "ret );
            exit( -1 );
        }

        /* In Natural Access Server mode, certain event buffers are allocated by
        *  Natural Access and then must be released back to Natural Access.
        */
        if ( event.buffer )
        {
           if ( ctaOwnsBuf = event.size & CTA_INTERNAL_BUFFER )
             /* This buffer is owned by Natural Access and will need to be
              * released back to Natural Access after processing of the buffer
              * is complete. Clear the Natural Access flags from the size field.
             */
               event.size &= ~CTA_INTERNAL_BUFFER;
        }
         switch ( event.id )
        {
            case APPEVN_EXCHANGE_NAME:
                /* This event message contains the name of the context
                *  to which we should now attach.
                */
                if ( event.buffer )
                {
                    if ((ret = ctaAttachContext( ctaqueuehd, userid,
                                                 event.buffer,  &mediaCtahd)))
                             != SUCCESS)
                    {
                        printf("Unable to attach to context %s\n",event.buffer);
                        exit( -1 );
                    }
                }
                break;
            case APPEVN_PLAY_MSG:
                /* The call control program indicates which message to play
                *  by sending the message number in the event's size field.
```

```
                */
                int msgNum = event.size;
                DemoPlayMessage( mediaCtahd, "ctademo", msgNum,
                                 0, NULL );
                break;
            default:
                break;
        }
        /* In Natural Access Client/Server mode, ensure proper release of
        * Natural Access owned buffers.
        */
        if ( ctaOwnsBuf )
        {
            ctaOwnsBuf = 0;
            ctaFreeBuffer( event.buffer );
        }
    }
}
```

ctaRefreshParms

Resets the values of all parameters on a context to the global defaults. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

Prototype

DWORD **ctaRefreshParms** (CTAHD ***ctahd***)

Argument	Description
ctahd	Handle returned by **ctaCreateContext**.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaRefreshParms resets the values of all parameters on a context to the global defaults. Use this function if another application process updates the global defaults and the application process needs these values.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetParms, **ctaSetEventSources**

Example

```
#define APPEVN_REFRESH_PARMS CTA_USER_EVENT(0x1)

void DemoSendRefreshParms(CTAHD ctahd)
{
    CTA_EVENT event;

    memset( &event, 0, sizeof( event ) );

    event.id    = APPEVN_REFRESH_PARMS;
    event.ctahd = ctahd;

    ctaQueueEvent( &event );
}

void DemoProcessEvents(CTAQUEUEHD ctaqueuehd, CTAHD ctahd)
{
    CTA_EVENT event;

    for(;;)
    {
        ctaWaitEvent(ctaqueuehd, &event, CTA_WAIT_FOREVER);

        switch ( event.id )
        {
        case APPEVN_REFRESH_PARMS:
            ctaRefreshParms( event.ctahd );
            break;

        case ADIEVN_CALL_DISCONNECTED:
            printf( "Caller hung up." );
            break;

        /* . . . */

        default:
            break;
        }
    }
}
```

ctaRegisterWaitObject

Registers a wait object so that Natural Access waits on application wait objects and Natural Access internal objects. Use this function when Natural Access is managing wait objects.

Prototype

DWORD **ctaRegisterWaitObject** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ****waitobj***, CTA_WAITFN ***waitfn***, void ****arg***, unsigned ***priority***)

Argument	Description
ctaqueuehd	Context handle.

waitobj	Pointer to the handle of the application wait object.
	For Windows, the type is:
	`typedef MUX_HANDLE CTA_WAITOBJ;`
	For UNIX, the type is:
	`struct pollhd { int fd; int events; int revents; } typedef struct pollhd CTA_WAITOBJ;`

waitfn	Callback function to be called when ***waitobj*** wait object is signaled. Refer to the Details section for the wait function prototype.
arg	Pointer to the argument to be passed to the callback function ***waitfn***.
priority	Relative priority of the wait object compared to Natural Access wait objects. Refer to the Details section for the valid priorities.

Return values

Return value	Description
SUCCESS	
CTAERR_DUPLICATE_WAITOBJ	Attempted to register a wait object that was already registered.
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaRegisterWaitObject registers a wait object with Natural Access so that **ctaWaitEvent** waits on application wait objects and Natural Access internal objects. A wait object can be registered only once with Natural Access. An attempt to register a wait object a second time returns CTAERR_DUPLICATE_WAITOBJ.

Natural Access calls the function ***waitfn.*** Whenever the wait object ***waitobj*** is signaled, Natural Access passes the argument ***arg*** to the function ***waitfn***. Natural Access gives up the queue lock before calling the function so that the function can call other Natural Access functions without a deadlock. On return from the function, Natural Access reacquires the queue lock.

Note: ctaDestroyQueue and **ctaWaitEvent** cannot be called from inside the function ***waitfn***.

The function prototype is:

DWORD NMSSTDCALL ***yourwaitfn*** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ***waitobj***, void ***arg***)

priority can be specified as either of the following values:

Value	Description
CTA_PRIORITY_HIGH	Natural Access checks the application's wait objects before checking its own wait objects.
CTA_PRIORITY_NORMAL	Natural Access checks its own wait objects before checking the application's wait objects.

Note: Do not use **ctaRegisterWaitObject** if the application is managing and waiting on wait objects outside of **ctaWaitEvent** (in **ctaInitialize**, ctaflags field in ***initparms*** is set to CTA_NOTIFY_UPDATE_WAITOBJS).

Call **ctaUnregisterWaitObject** to unregister the wait object. Refer to *Using wait objects* on page 49 for more information.

See also

ctaQueryWaitObjects

Example

```
#include "nmstypes.h"
#include "ctademo.h"
#include "ctadef.h"
extern DWORD NMSSTDCALL MyKbdEventHandler( CTAQUEUEHD  ctaqueuehd,
                                           CTA_WAITOBJ *waitobj,
                                           void       *arg);

CTA_WAITOBJ waitobj;
CTAHD kbdhd;

void register_stdin_waitobj(CTAQUEUEHD ctaqueuehd)
{
    DWORD ret;

#if defined (WIN32)
    /* Get a handle for stdin */
    HANDLE hConsoleInput = GetStdHandle( STD_INPUT_HANDLE ) ;
    if (hConsoleInput == INVALID_HANDLE_VALUE)
        exit( -1 );
#endif

#if defined (WIN32)
    waitobj = hConsoleInput;
#elif defined (UNIX)
    /* Use the STDIN file descriptor */
    waitobj.fd = STDIN_FILENO;
    waitobj.events = POLLIN;
    waitobj.revents = 0;
#endif

    /* Register the wait object */
    if ((ret=ctaRegisterWaitObject(ctaqueuehd,
                                   &waitobj,
                                   MyKbdEventHandler,
                                   (void *)kbdhd,
                                   CTA_PRIORITY_NORMAL)) != SUCCESS)
    {
```

```
        exit( -1 );
    }
}
```

ctaSetDefaultServer

Specifies the location of the default Natural Access Server (*ctdaemon*) with which the client application communicates.

Prototype

DWORD **ctaSetDefaultServer** (char ***address**)

Argument	Description
address	Pointer to the server address or object descriptor string.

Return values

Return value	Description
SUCCESS	
CTA_BAD_ARGUMENT	Invalid server address.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaSetDefaultServer specifies the location of the default Natural Access Server (*ctdaemon*) with which the client application communicates. This function overrides the default server assigned in the *cta.cfg* file.

Example

```
int DemoSetDefaultServer()
{
      int ret;
          ret = ctaSetDefaultServer("cta://nms-server.company.com/");
          return ret;
}
```

ctaSetErrorHandler

Defines the action taken when an error is returned from any Natural Access functions, including any Natural Access service functions.

Prototype

DWORD **ctaSetErrorHandler** (CTA_ERROR_HANDLER ***handler***, CTA_ERROR_HANDLER ****oldhandler***)

Argument	Description
handler	Application error action. See the Details section for valid values.
oldhandler	Pointer to the returned previous application error action.

Return values

SUCCESS

Details

ctaSetErrorHandler defines how the application program traps programming errors. If an error is encountered when executing any Natural Access or Natural Access service command, Natural Access performs the error action before returning to the application.

Note: Do not call any functions from the application's error handler unless the code first sets a different error handler.

The error action is undertaken only when Natural Access functions or service functions are invoked; it is not undertaken if an error is returned by way of a CTA_EVENT structure.

By default, there is no error action. The application can specify any of the following error actions:

Error	Action
NULL	Restores the error handling action to the default of no action.
CTA_REPORT_ERRORS	Prints a brief message to stdout. The error code is then returned from the function to the application.
CTA_EXIT_ON_ERRORS	Prints the error as defined in CTA_REPORT_ERRORS and then invokes exit with a -1 exit code.
yourhandler	Pointer to an application call back function called by Natural Access.

The error handling function prototype is:

DWORD NMSSTDCALL ***yourhandler*** (CTAHD ***ctahd***, DWORD ***error***, char ****errtxt***, char ****func***)

Argument	Description
ctahd	Context handle.
error	Error code as detected by Natural Access.
errtxt	Pointer to an error text description (see **ctaGetText**).
func	Pointer to the function name in which the error occurred.

Natural Access returns to the application whatever is returned from ***yourhandler***. It is recommended that the ***errcode*** received is returned.

Refer to *Handling Natural Access errors* on page 55 for more information.

Example

```
DWORD NMSSTDCALL myErrorHandler( CTAHD  ctahd, DWORD errorcode,
                                 char *errtxt, char *func )
{
    /* print out the error and exit */
    printf( "myErrorHandler: %s on ctahd=%#x, %s\n", errtxt, ctahd, func );
    exit(errorcode);
}
/* This makes more sense if we assume the main program called
 * ctaSetErrorHandler(myErrorHandler, NULL); before calling this function. */
void DemoInitTrace()
{
    DWORD             ret;
    CTA_INIT_PARMS    initparms = { 0 };
    CTA_ERROR_HANDLER hdlr;

    /* Initialize size of init parms structure */
    initparms.size = sizeof(CTA_INIT_PARMS);

    /* If daemon running then initialize tracing */
    initparms.traceflags = CTA_TRACE_ENABLE;

    /* Set error handler to NULL and remember old handler */
    ctaSetErrorHandler(NULL, &hdlr);

    if ((ret = ctaInitialize(InitServices,
                             sizeof(InitServices)/sizeof(InitServices[0]),
                             &initparms)) != SUCCESS)
    {
        initparms.traceflags = 0; /* clear trace flags */

        ctaSetErrorHandler(hdlr, NULL); /* restore error handler */

        /* try to initialize again - will exit on failure */
        ctaInitialize(InitServices,
                      sizeof(InitServices)/sizeof(InitServices[0]),
                      &initparms);

        printf("Tracing disabled. Check that the daemon is running.");
    }
    else
        ctaSetErrorHandler(hdlr, NULL);  /* restore error handler */

    /* . . . */
}
```

ctaSetEventSources

Provides a list of services from which the specified context receives events.

Prototype

DWORD **ctaSetEventSources** (CTAHD ***ctahd***, char ****svcname***, unsigned ***nsvcs***)

Argument	Description
ctahd	Context handle.
svcname	Pointer to an array containing a list of services from which the application receives events.
nsvcs	Number of services in the ***svcname*** list.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid ***svcname*** was specified.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaSetEventSources specifies the types of events Natural Access sends to an event queue associated with a shared context. Each client application that uses the shared context can keep its own list of services. Natural Access sends to the event queue only events corresponding to entries in the ***svcname*** list.

Note: By default, if a context is created in Declared Access mode, applications that attach to it receive service events only for services they explicitly open (even if other services are opened by other client applications). If a context is created in Common Access mode, all applications that attach to it receive events for all service instances opened on the context.

The specified ***svcname*** list can include service instances not yet opened on the shared context. Reset the event mask by invoking **ctaSetEventSources** again and specifying a different ***svcname*** list.

Applications can retrieve the current list of ***svcnames*** in the existing event mask by invoking **ctaGetEventSources**.

Refer to *Receiving events on event queues* on page 46 for more information.

See also

ctaAttachContext, **ctaCreateContextEx**

Example

```
case PAIR('S', 'M' ):               /* Set Event Mask */
     {
     char *svcarr[MAX_SERVICES];
     int nsvcs;
     if (!Ioffline)
        printf("Enter list of services to mask; Press Enter to end list\n")
      nsvcs = PromptForStrings("Enter service name", svcarr,
                               MAX_SVC_NAME_LEN);
      ret = ctaSetEventSources(ctahd, svcname, nsvcs);
      for ( ; nsvcs >= 0; nsvcs--)
      {
        free(svcarr[nsvcs]);
      } break;
     }
```

ctaSetGlobalTraceMask

Sets the global trace mask.

Prototype

DWORD **ctaSetGlobalTraceMask** (CTAHD ***ctahd,*** unsigned ***tracemask***)

Argument	Description
ctahd	Context handle for the server on which the global trace mask is changed. ***ctahd*** can be a void context handle.
tracemask	The new global trace mask.

Returned Values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaSetGlobalTraceMask sets the global trace mask for the Natural Access Server (*ctdaemon*) specified by ***ctahd***. CTA_TRACEMASK macros accepted by **ctaSetTraceLevel** can be used for the ***tracemask*** argument.

See also

ctaSetTraceLevel

Example

```
int DemoRemoteTrace(CTAQUEUEHD ctaqueuehd, CTAHD ctahd)
{
     DWORD ret;

          /* Start tracing */
          ret = ctaStartTrace(ctahd);
          if ( ret != SUCCESS )
             return ret;

          /* Set tracemask on server */
          ret = ctaSetGlobalTraceMask(ctahd,CTA_TRACEMASK_ALL_COMMANDS);
          if ( ret != SUCCESS )
             return ret;

          for (;!kbhit();)
          {
               // Retrieve an event from the event queue.
               CTA_EVENT cta_event;
               ret = ctaWaitEvent( ctaqueuehd, &cta_event, 1000 );

          // Check if buffer is owned by Natual Access and must be freed by
          // us below.
              bool const bCtaOwnsBuf = (cta_event.buffer != NULL &&
```

```
              (cta_event.size & CTA_INTERNAL_BUFFER));
              if (bCtaOwnsBuf)
              {
                  cta_event.size &= ~CTA_INTERNAL_BUFFER;
                     // clear flag from size
              }
              if (cta_event.id == CTAEVN_TRACE_MESSAGE)
                     // if it's a trace event
              {
                  if (!cta_event.buffer)
                  {
                      continue;
                  }

                   const char * const pszText = (char*)cta_event.buffer;

                   printf( "%s", pszText );
               }

               if (bCtaOwnsBuf)
               {
                   ctaFreeBuffer( cta_event.buffer );
                      // our reponsibility to free
               }
          }

         /* Stop tracing */
         ret = ctaStopTrace( ctahd );
         if ( ret != SUCCESS )
             return ret;

          return SUCCESS;

}
```

ctaSetParmByName

Modifies a given parameter.

Prototype

DWORD **ctaSetParmByName** (CTAHD ***ctahd***, char ****parmname***, void ****buffer***, unsigned ***size***)

Argument	Description
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmname	Pointer to a string that specifies the name of the parameter. See the Details section for the ***parmname*** format.
buffer	Pointer to a buffer that contains the new values of the named parameter.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***buffer*** size does not match the size of the parameter field data.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmname*** was not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

Use **ctaSetParmByName** to set the values of the named parameter. If ***ctahd*** is a valid context, this function modifies the parameters from the context defaults.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle. If the void context handle is equal to NULL_CTAHD, it refers to the default server.

If ***ctahd*** is a void context handle, this function retrieves parameters from the global defaults and executes on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***].

An example of ***parmname*** is vce.play.gain.

You can retrieve the values of parameters using **ctaGetParms** or **ctaGetParmByName**.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetEventSources, **ctaGetParmInfo**, **ctaSetParmByNameEx**

Example

```
void DemoSetContextPlayGain(CTAHD ctahd, INT32 newgain)
{
    INT32 oldgain;

    ctaGetParmByName( ctahd, "vce.play.gain",
                      &oldgain, sizeof(oldgain) );

    ctaSetParmByName( ctahd, "vce.play.gain",
                      &newgain, sizeof(newgain) );

    printf( "CTAHD=%x vce.play.gain is now %d dB (old value = %d dB)\n",
            ctahd, newgain, oldgain);
}
```

ctaSetParmByNameEx

Modifies a given parameter on a particular Natural Access Server.

Prototype

DWORD **ctaSetParmByNameEx** (char ***addr***, CTAHD ***ctahd***, char ****parmname***, void ****buffer***, unsigned ***size***)

Argument	Description
addr	Natural Access Server address.
ctahd	Context handle that specifies the server on which commands are executed. ***ctahd*** can be a void context handle.
parmname	Pointer to a string that specifies the name of the parameter. See the Details section for the ***parmname*** format.
buffer	Pointer to a buffer that contains the new values of the named parameter.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL.
CTAERR_BAD_SIZE	***buffer*** size does not match the size of the parameter field data.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_FOUND	***parmname*** was not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

Use **ctaSetParmByNameEx** to set the values of the named parameter. If ***ctahd*** is a valid context, this function modifies the parameters from the context defaults.

ctahd can be a void context handle. A void context handle refers only to the server on which the commands are executed and not to any real context object. Each server has a unique void context handle.

If ***ctahd*** is a void context handle, this function retrieves parameters from the global defaults and executes on the server specified by the void context handle. Global defaults are either statically defined values maintained within a process or dynamically defined values maintained by the Natural Access Server.

If the context handle is equal to NULL_CTAHD, the function retrieves parameters from the server specified by ***addr***. If ***addr*** is equal to NULL or is an empty string, the function retrieves parameters from the default server.

The ***parmname*** format is ***svcname***[.x].***structname***[.***fieldname***].

An example of ***parmname*** is vce.play.gain.

You can retrieve the values of parameters using **ctaGetParmsEx** or **ctaGetParmByNameEx**.

Refer to *Managing parameters* on page 51 for more information.

See also

ctaGetParmByName, **ctaGetParmID**, **ctaGetParmIds**, **ctaGetParmIdsEx**, **ctaGetParmInfo**, **ctaGetParmInfoEx**, **ctaGetParms**, **ctaSetParmByName**

Example

```
void DemoSetContextPlayGain(CTAHD ctahd, INT32 newgain)
{
    INT32 oldgain;

    ctaGetParmByNameEx( "localhost", ctahd, "vce.play.gain",
                        &oldgain, sizeof(oldgain) );

    ctaSetParmByNameEx( "localhost", ctahd, "vce.play.gain",
                        &newgain, sizeof(newgain) );

    printf( "CTAHD=%x vce.play.gain is now %d dB (old value = %d dB)\n",
            ctahd, newgain, oldgain);
}
```

ctaShutdown

Shuts down the Natural Access Server (*ctdaemon*) or the server's host system and restarts the server or the server's host system.

Prototype

DWORD **ctaShutdown** (CTAHD ***ctahd***, unsigned ***action***)

Argument	Description
ctahd	Context handle of the server to be shut down. ***ctahd*** can be a void context handle.
action	Action to be performed. See the Details section for valid actions.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid ***action***.
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_IMPLEMENTED	Function is not available in the execution mode associated with the specified context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaShutdown shuts down the Natural Access Server (*ctdaemon*) or the server's host system and restarts the server or the server's host system. ***action*** can be the following values:

Value	Description
CTA_SERVER_RESTART	Restarts the Natural Access Server (*ctdaemon*).
CTA_SERVER_SHUTDOWN	Shuts down the Natural Access Server (*ctdaemon*).
CTA_SYSTEM_SHUTDOWN	Shuts down the server's host system.
CTA_SYSTEM_REBOOT	Reboots the server's host system.

Example

```
int DemoShutdown(char* servername)

{
         DWORD ret;

         ret = ctaSetDefaultServer(servername);
         if ( ret != SUCCESS )
            return ret;

         // Create void context on default server
         CTAHD ctahd;
         ret = ctaCreateContext(NULL_CTAQUEUEHD,0,NULL,&ctahd);
         if ( ret != SUCCESS )
            return ret;

         // Shutdown Natural Access Server
         ret = ctaShutdown(ctahd,CTA_SERVER_SHUTDOWN);

         return ret;
}
```

ctaSetTraceLevel

Sets the trace mask for a service on the specified context.

Prototype

DWORD **ctaSetTraceLevel** (CTAHD ***ctahd***, char ****svcname***, unsigned ***tracemask***)

Argument	Description
ctahd	Context handle. ***ctahd*** cannot be NULL.
svcname	Pointer to the name of the service for which you will change the ***tracemask***. Use CTA to modify the Natural Access trace mask.
tracemask	Trace mask of the service. See the Details section for valid values.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	
CTAERR_NOT_FOUND	Service is not open on the context.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

ctaSetTraceLevel sets the trace mask for a service on a specified context. If ***svcname*** is NULL, the function sets the trace masks of all services opened within the ***ctahd*** to the new ***tracemask***.

Use the following macros from *ctadef.h* for the ***tracemask*** argument:

Value	Description
CTA_TRACEMASK_DRIVER_COMMANDS	Trace commands to devices.
CTA_TRACEMASK_DRIVER_EVENTS	Trace events to devices.
CTA_TRACEMASK_DISP_COMMANDS	Trace service-to-service commands.
CTA_TRACEMASK_DISP_EVENTS	Trace service-to-service events.
CTA_TRACEMASK_API_COMMANDS	Trace application commands.
CTA_TRACEMASK_API_EVENTS	Trace application events.
CTA_TRACEMASK_API_ERRORS	Trace application errors.
CTA_TRACEMASK_SVC_ERRORS	Trace internal service errors.
CTA_TRACEMASK_DEBUG_BIT0	Trace service-specific messages.
CTA_TRACEMASK_DEBUG_BIT1	Trace service-specific messages.
CTA_TRACEMASK_DEBUG_BIT2	Trace service-specific messages.
CTA_TRACEMASK_DEBUG_BIT3	Trace service-specific messages.

Use the following macros to unify related groups of ***tracemask*** values:

Value	Description
CTA_TRACEMASK_ALL_COMMANDS	Trace all commands.
CTA_TRACEMASK_ALL_EVENTS	Trace all events.
CTA_TRACEMASK_ALL	Trace everything.

ctdaemon must be running and *y*ou must have set the global ***tracemask*** in *ctdaemon* for tracing to work. This setting affects all processes (and all contexts within the process) that have enabled tracing with **ctaInitialize***.* For more information, refer to *Natural Access Server: ctdaemon* on page 238.

Based on the flags set in ***tracemask***, the specified service logs internal trace information to *ctdaemon*. Each service that manages a device can respond to the CTA_TRACEMASK_DRIVER_COMMANDS or CTA_TRACEMASK_DRIVER_EVENTS bits in ***tracemask***.

CTA_TRACEMASK_DISP_COMMANDS, CTA_TRACEMASK_API_COMMANDS, CTA_TRACEMASK_API_EVENTS, and CTA_TRACEMASK_API_ERRORS are used to enable Natural Access dispatcher tracing. Use CTA as the service name to enable this capability.

The debug bits are service specific. Refer to the documentation for the service for more information.

Refer to *Error tracing* on page 56 for more information.

See also

ctaLogTrace

Example

```
/*
 * These functions will enable/disable tracing dispatcher and API events on the
 * given context. To trace driver commands each service must be specified
 * separately. To set the global trace mask, use the ctdaemon. This enables
 * tracing on all contexts in processes that have enabled tracing.
 */

void DemoTraceEventsOn(CTAHD ctahd)
{
    ctaSetTraceLevel( ctahd, "CTA", CTA_TRACEMASK_ALL_EVENTS );
}

void DemoTraceEventsOff(CTAHD ctahd)
{
    ctaSetTraceLevel( ctahd, "CTA", 0 );
}
```

ctaStartTrace

Starts tracing on a remote server.

Prototype

DWORD **ctaStartTrace** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle for the server on which tracing is started. ***ctahd*** cannot be a void context handle because void contexts do not have queues.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Events

Event	Description
CTAEVN_TRACE_MESSAGE	This event passes trace messages from the server to the client applications. The ***buffer*** field of the CTA_EVENT structure points to an array containing the message text. Trace messages are text in the same format as they appear in the Natural Access Server (*ctdaemon*) log file.

Details

After an application initiates a call to **ctaStartTrace**, the trace messages are sent as CTAEVN_TRACE_MESSAGE events to the queue on which the ***ctahd*** context was created.

See also

ctaSetGlobalTraceMask, **ctaStopTrace**

Example

```
int DemoRemoteTrace(CTAQUEUEHD ctaqueuehd, CTAHD ctahd)
{
     DWORD ret;

          /* Start tracing */
          ret = ctaStartTrace(ctahd);
          if ( ret != SUCCESS )
             return ret;

          /* Set tracemask on server */
          ret = ctaSetGlobalTraceMask(ctahd,CTA_TRACEMASK_ALL_COMMANDS);
          if ( ret != SUCCESS )
             return ret;

          for (;!kbhit();)
          {
               // Retrieve an event from the event queue.
               CTA_EVENT cta_event;
               ret = ctaWaitEvent( ctaqueuehd, &cta_event, 1000 );

          // Check if buffer is owned by Natual Access and must be freed by
          // us below.
              bool const bCtaOwnsBuf = (cta_event.buffer != NULL &&
              (cta_event.size & CTA_INTERNAL_BUFFER));
              if (bCtaOwnsBuf)
              {
                  cta_event.size &= ~CTA_INTERNAL_BUFFER;
                     // clear flag from size
              }
              if (cta_event.id == CTAEVN_TRACE_MESSAGE)
                     // if it's a trace event
              {
                  if (!cta_event.buffer)
                  {
                      continue;
                  }

                   const char * const pszText = (char*)cta_event.buffer;

                   printf( "%s", pszText );
               }

               if (bCtaOwnsBuf)
               {
                   ctaFreeBuffer( cta_event.buffer );
                      // our reponsibility to free
               }
          }

         /* Stop tracing */
         ret = ctaStopTrace( ctahd );
         if ( ret != SUCCESS )
             return ret;

          return SUCCESS;

}
```

ctaStopTrace

Stops tracing on a remote server.

Prototype

DWORD **ctaStopTrace** (CTAHD ***ctahd***)

Argument	Description
ctahd	Context handle for the server on which tracing is stopped. ***ctahd*** cannot be a void context handle because void contexts do not have queues.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.

Details

After an application initiates a call to **ctaStopTrace**, the server specified by ***ctahd*** no longer receives trace messages.

See also

ctaSetGlobalTraceMask, **ctaStartTrace**

Example

```
int DemoRemoteTrace(CTAQUEUEHD ctaqueuehd, CTAHD ctahd)
{
     DWORD ret;

          /* Start tracing */
          ret = ctaStartTrace(ctahd);
          if ( ret != SUCCESS )
             return ret;

          /* Set tracemask on server */
          ret = ctaSetGlobalTraceMask(ctahd,CTA_TRACEMASK_ALL_COMMANDS);
          if ( ret != SUCCESS )
             return ret;

          for (;!kbhit();)
          {
               // Retrieve an event from the event queue.
               CTA_EVENT cta_event;
               ret = ctaWaitEvent( ctaqueuehd, &cta_event, 1000 );

          // Check if buffer is owned by Natual Access and must be freed by
          // us below.
              bool const bCtaOwnsBuf = (cta_event.buffer != NULL &&
              (cta_event.size & CTA_INTERNAL_BUFFER));
              if (bCtaOwnsBuf)
              {
                  cta_event.size &= ~CTA_INTERNAL_BUFFER;
                     // clear flag from size
              }
              if (cta_event.id == CTAEVN_TRACE_MESSAGE)
                     // if it's a trace event
              {
                  if (!cta_event.buffer)
                  {
                      continue;
                  }

                   const char * const pszText = (char*)cta_event.buffer;

                   printf( "%s", pszText );
               }

               if (bCtaOwnsBuf)
               {
                   ctaFreeBuffer( cta_event.buffer );
                      // our reponsibility to free
               }
          }

         /* Stop tracing */
         ret = ctaStopTrace( ctahd );
         if ( ret != SUCCESS )
             return ret;

          return SUCCESS;

}
```

ctaUnregisterWaitObject

Unregisters a previously registered wait object. Use this function when Natural Access is managing wait objects.

Prototype

DWORD **ctaUnregisterWaitObject** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ***waitobj***)

Argument	Description
ctaqueuehd	Context handle.
waitobj	Pointer to the handle of the application wait object.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***waitobj*** is NULL.
CTAERR_INVALID_CTAQUEUEHD	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_NOT_FOUND	***waitobj*** was not found.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.

Details

ctaUnregisterWaitObject unregisters a wait object from Natural Access. The wait object was registered by a previous call to **ctaRegisterWaitObject**. On return from this function, Natural Access ignores any signals on the wait object.

Note: Do not use **ctaUnregisterWaitObject** if the application is managing and waiting on wait objects outside of **ctaWaitEvent** (in **ctaInitialize**, ctaflags field in ***initparms*** is set to CTA_NOTIFY_UPDATE_WAITOBJS).

Refer to *Using wait objects* on page 49 for more information.

Example

```
#include "nmstypes.h"
#include "ctademo.h"
#include "ctadef.h"

void unregister_waitobj(CTAQUEUEHD ctaqueuehd, CTA_WAITOBJ *waitobj)
{
    DWORD ret;

    ret = ctaUnregisterWaitObject(ctaqueuehd, waitobj);
    if (ret != SUCCESS)
        printf("UnregisterWaitObject failed...\n");
}
```

ctaWaitEvent

Retrieves an event from the specified queue.

Prototype

DWORD **ctaWaitEvent** (CTAQUEUEHD ***ctaqueuehd***, CTA_EVENT ****ctaevt***, unsigned ***timeout***)

Argument	Description
ctaqueuehd	Context handle.

ctaevt	Pointer to the event structure to be filled. The CTA_EVENT structure is:
	`typedef struct`
	`{`
	`  DWORD  id;  /* Event code and source service ID */`
	`  CTAHD  ctahd;  /* Natural Access context handle */`
	`  DWORD  timestamp;  /* Timestamp */`
	`  DWORD  userid;  /* Userid(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 object handle */`
	`} CTA_EVENT;`

timeout	Time to wait (in milliseconds) for an event before returning control to the caller. Use CTA_WAIT_FOREVER for an infinite timeout.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAQUEUEHD	Queue is being destroyed or has already been destroyed.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized. Call **ctaInitialize** first.
CTAERR_SVR_COMM	Server communication error.
CTAERR_WAIT_FAILED	Operating-system-specific wait function returned an error.
CTAERR_WAIT_PENDING	Another thread already called **ctaWaitEvent** on the same queue.

Events

CTAEVN_WAIT_TIMEOUT

Details

Call **ctaWaitEvent** to retrieve an event from the application event queue.

If no events are already queued, the function blocks for the specified timeout, waiting for events to appear. If the timeout is exceeded, the event structure returns with a CTAEVN_WAIT_TIMEOUT code indicating that no events were pending and that the timeout condition occurred. In this case, the function returns SUCCESS.

If another thread destroys the queue, this function returns CTAERR_INVALID_CTAQUEUEHD. Subsequent calls to this function fail with the same error code.

An application can wait on wait objects outside of **ctaWaitEvent** by setting the **ctaInitialize** ctaflags field in ***initparms*** to CTA_NOTIFY_UPDATE_WAITOBJS. If wait objects are signaled in this instance, call **ctaWaitEvent** with ***timeout*** set to zero (0) to process and retrieve a waiting event. CTAEVN_WAIT_TIMEOUT is returned if there is no application event available as a result of wait object processing.

Refer to *Using wait objects* on page 49 for more information.

See also

ctaFormatEvent, **ctaQueryWaitObjects**, **ctaQueueEvent**

Example

```
extern CTAQUEUEHD ctaqueuehd;
void myGetEvent( CTAHD ctahd, CTA_EVENT *event )
{
    DWORD ctaret ;
    BOOL ctaOwnsBuf = FALSE;

    ctaret = ctaWaitEvent( ctaqueuehd, event, CTA_WAIT_FOREVER );
    if (ctaret != SUCCESS)
    {
        printf( "\007ctaWaitEvent returned %x\n", ctaret);
        exit( 1 );
    }
    if ( event->buffer )
    {
       ctaOwnsBuf = event->size & CTA_INTERNAL_BUFFER;
       if ( ctaOwnsBuf )
       {
           /* This buffer is owned by Natural Access and will need to be
            * released back to Natural Access after processing of the buffer
            * is complete. Clear the Natural Access flags from the size field.
            */
           event->size &= ~CTA_INTERNAL_BUFFER;
       }
    }

    /* ... Process incoming event here .... */

    /* In Natural Access Client/Server mode, ensure proper release of
     * Natural Access owned buffers.
     */
    if ( ctaOwnsBuf )
    {
        ctaFreeBuffer( event->buffer );
    }
    return ;
}
```

9 Demonstration programs

Summary of the demonstration programs

Each demonstration program is shipped as an executable program with its source and makefiles.

Before you start the demonstration programs, ensure that

- Natural Access is properly installed.
- The board is executing.
- CT bus switching is correctly configured.

Refer to the board-specific installation and developer's manual for details on installing the boards.

The following demonstration programs are provided with Natural Access:

Program	Description
boardinf	Retrieves and displays configurations for all boards using Natural Access in the host computer.
csplayrc	Demonstrates context sharing on a local server.
ctatest	Demonstrates Natural Access functions.
incta	Demonstrates inbound calls.
inoutcta	Places and receives calls using CAS protocols.
outcta	Demonstrates outbound calls.
takecmd	Controls *takeover* demonstration program.
takeover	Demonstrates sharing control of Natural Access service objects.
testdm	Demonstrates using the Natural Access functionality running within the process address space and using Natural Access Server (*ctdaemon*) on a local and remote host.

The *trunklog* demonstration program, provided by the Digital Trunk Monitor service, displays alarm events from digital trunks. Refer to the *Digital Trunk Monitor Service Developer's Reference Manual* for more information.

Natural Access also provides utilities. For more information, see *Summary of the utilities* on page 235.

Board configuration: boardinf

Retrieves and displays the hardware and MVIP configuration for all boards that use Natural Access in the host computer.

Usage

```
boardinf
```

There are no command-line options.

Featured functions

adiGetBoardInfo, **adiGetBoardSlots**

Description

boardinf opens a board driver under the Natural Access environment and retrieves up to 16 board configurations. The program displays the configuration and a list of available DSP timeslots for each board, formatting this array into timeslot ranges before displaying it.

Procedure

Ensure that you have initialized the boards using the configuration utility.

Run *boardinf* by entering boardinf at the prompt.

Note: You can use **adiGetBoardInfo** and **adiGetBoardSlots** to retrieve system configuration information when initializing your applications.

Service sharing on a local server: csplayrc

Demonstrates context sharing on a local server. Use *csplayrc* to

- Attach and share contexts.
- Hand off Natural Access object descriptors or context names from application to application.
- Queue and receive events through shared communication contexts.

Usage

```
csplayrc [options]
```

where ***options*** include:

Option	Use this option to specify...
-m ***contextname***	The context name for the communication context. Default: commn-cxt **Note:** This name must be the same communication context name as specified with *incta*.
-F ***filename***	The name of the Natural Access configuration file. Default: *cta.cfg*

Featured functions

ctaAttachContext, **ctaCreateContextEx**, **ctaQueueEvent**, **vceCreateFile**, **vceOpenFile**, **vcePlayList**, **vceRecordMessage**

Description

csplayrc provides the same functionality as *incta*, but executes commands through a context shared by both *csplayrc* and *incta*. *csplayrc* performs voice play and record functions, while *incta* performs call control for incoming calls.

Note: When two applications are using a local server, an application-defined context name (specified with **ctaCreateContextEx**) can be used with **ctaAttachContext** as a substitute for an object descriptor. The object descriptor consists only of the host name and a context name.

csplayrc works with *incta* as described in the following table:

Step	Action
1	When you start *csplayrc*, it creates and names a context (by default named commn-cxt) that is used for sharing events with *incta*. *csplayrc* uses **ctaWaitEvent** to listen for events on the event queue.
2	When you start *incta* (specifying the -C option for server mode), it creates and names a context, and opens the Natural Call Control (NCC) service and the Voice Message (VCE) service on the context.
3	*incta* uses **ctaAttachContext** to attach to the context created by *csplayrc* (commn-cxt), and uses **ctaQueueEvent** to send APPEVN_CONTEXT_NAME to the context's event queue. The buffer field in APPEVN_CONTEXT_NAME points to the context name used to create the context in step 2.
4	*csplayrc* receives APPEVN_CONTEXT_NAME and extracts the context name from the buffer field of the event. *csplayrc* uses **ctaAttachContext** to attach to the context.
5	When *incta* receives calls, it uses **ctaQueueEvent** to send APPEVN_RECORD_MESSAGE or APPEVN_PLAY_MESSAGE to the communication context event queue according to the input the user supplies.
6	Depending on the type of events it receives on the event queue, *csplayrc* uses appropriate VCE service functions to play or record voice messages for the specified calls.

Procedure

Complete the following steps to run *csplayrc* with *incta.* This procedure assumes you are testing on an AG 2000 board that has a 2500-type telephone connected to one of the lines.

Note: After Step 2, this procedure is identical to that used for running *incta*.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Start csplayrc by entering the following command at the prompt:
<code>csplayrc</code></td></tr>
<tr><td>2</td><td>Start incta by entering the following command at the prompt:
<code>incta -C -b n -s [stream:]slot -p wnk0</code></td></tr>
<tr><td>3</td><td>Specify the board and timeslot for the handset you are using.</td></tr>
<tr><td>4</td><td>Take the handset off-hook. The wink-start is acknowledged.</td></tr>
<tr><td>5</td><td>Enter three DTMF digits as the address.
<table>
<tr><th>If the first digit is...</th><th>Then the demonstration releases the call...</th></tr>
<tr><td>8</td><td>By playing a reorder tone.</td></tr>
<tr><td>9</td><td>With a busy tone.</td></tr>
<tr><td>0</td><td>By playing a special information tone.</td></tr>
</table>
Otherwise, the call connects and the demonstration proceeds into the conversation state. You are prompted to record voice, play back your recorded sample, or hang up.</td></tr>
<tr><td>6</td><td>Select the record voice option.
You hear the record beep prompt.</td></tr>
</table>

Step	Action
7	Begin speaking after the prompt. After you stop speaking, the record operation terminates, and you return to the options menu.
8	Select the playback option. The sample you just recorded plays back, and you return to the options menu.
9	Select the hang-up option.

Interactive test: ctatest

Demonstrates Natural Access functions, executing in asynchronous mode. Use *ctatest* to:

- Verify proper installation and operation of the NCC service and the ADI service.
- Experiment with Natural Access.
- Expose working examples of Natural Access and NCC service or ADI service functions.

Usage

```
ctatest [options]
```

where ***options*** are:

Option	Use this option to...
-A ***xxxmgr***	Specify the Natural Access service manager. Default: ADIMGR
-b ***n***	Specify the board number ***n***. Default: 0
-C	Notify the application through an event whenever Natural Access changes its list of internal wait objects. Default: 0
-f ***filename***	Specify the file name to retrieve service names. Default: None
-F ***filename***	Specify the name of the Natural Access configuration file. Default: *cta.cfg*
-i ***filename***	Specify the name of the input file. This option allows you to use a file listing Natural Access test commands and data line by line, instead of entering the commands interactively.
-l	Show low-level events.
-p ***protocol***	Specify the protocol to run. Default: nocc
-s [***stream:***]***slot***	Specify the port (DSP) address. Default: 0:0
-t d/r	Set the tracing option. Default: r d = disable tracing r = enable tracing
-v	Set the Natural Access compatibility level to 0.
-w	Wait before exit.
-?	Access Help.

Featured functions

Natural Access system functions, Voice Message service functions, and many ADI service functions are featured.

Description

ctatest is a menu-driven interactive program. Enter one- and two-letter commands to execute Natural Access and NCC service, ADI service, or VCE service commands. Some commands prompt for additional information, such as frequencies and amplitudes for tone generators. For more information about the service commands, refer to the service-specific reference manuals.

The following table describes the available commands:

Function	Command	Description
Help	H	Displays a table of available commands.
Quit	Q	Exits from the *ctatest* program.
Repeat command	!	Repeats the previous command.
Abort dial	AD	Calls **adiStopDial**.
Abort timer	AT	Calls **adiStopTimer**.
Accept call	CC	Calls **nccAcceptCall**.
Answer call	AC	Calls **nccAnswerCall**.
Assert signal	AS	Calls **adiAssertSignal**.
Attach context	AX	Calls **ctaAttachContext**.
Attach object	AO	Calls **ctaAttachObject**.
Block calls	BC	Calls **nccBlockCalls**.
Call progress begin	CB	Calls **adiStartCallProgress**.
Call progress stop	CS	Calls **adiStopCallProgress**.
Call status	C?	Calls **nccGetCallStatus**.
Change context	CH	Switches between different contexts created by *ctatest* with the CX or PX commands.
Close services (new)	CV	Calls **ctaCloseServices**.
Close services (old)	CP	Calls **ctaDestroyContext**.
Collect digits	CD	Calls **adiCollectDigits**.
Collect stop	SC	Calls **adiStopCollection**.
Get context information	CN	Calls **ctaGetContextInfoEx**.
Continuous event fetch	CE	Enables fetching of events.
Create context	CX	Calls **ctaCreateContextEx**.
Create persistent context	PX	Calls **ctaCreateContextEx** with the CTA_CONTEXT_PERSISTENT flag set.

Function	Command	Description
Create void context	VX	Calls **ctaCreateContext** with ***ctaqueuehd*** set to NULL_CTAQUEUEHD.
Destroy context	DX	Calls **ctaDestroyContext**.
Digit flush	DF	Calls **adiFlushDigitQueue**.
Digit get	DG	Calls **adiGetDigit**.
Digit peek	DP	Calls **adiPeekDigit**.
Disable bit detector	DB	Calls **adiStopSignalDetector**.
Disable DTMF detector	DD	Calls **adiStopDTMFDetector**.
Disable energy detector	DE	Calls **adiStopEnergyDetector**.
Disable MF detector	DM	Calls **adiStopMFDetector**.
Disable tone detector	DT	Calls **adiStopToneDetector**.
Disconnect call	DC	Calls **nccDisconnectCall**.
Detach object	DO	Calls **ctaDetachObject**.
Enable bit detector	EB	Calls **adiStartSignalDetector**.
Encoding information	EI	Calls **vceGetEncodingInfo**.
Enable DTMF detector	ED	Calls **adiStartDTMF Detector**.
Enable energy detector	EE	Calls **adiStartEnergyDetector**.
Enable MF detector	EM	Calls **adiStartMFDetector**.
Enable tone detector	ET	Calls **adiStartToneDetector**.
Enumerate contexts	EC	Lists contexts created by *ctatest* with the CX, PX, or VX commands.
Find file	FF	Calls **ctaFindFile**.
Format event	FE	Calls **ctaFormatEventEx**.
FSK abort receive	FA	Calls **adiStopReceivingFSK**.
FSK receive	FR	Calls **adiStartReceivingFSK**.
FSK send	FS	Calls **adiStartSendingFSK**.
Generate DTMFs	GD	Calls **adiStartDTMF**.
Generate net tone	GN	Calls **adiStartTones**.
Generate wink	GW	Calls **adiStartPulse**.
Generation stop	GS	Calls **adiStopTones**.
Generate user tone	GT	Calls **adiStartTones.**
Get call status	L?	Calls **nccGetLineStatus**.

Function	Command	Description
Get context list	CL	Calls **ctaQueryServerContexts**.
Get event source	GM	Calls **ctaGetEventSources**.
Get object descriptor	DH	Calls **ctaGetObjDescriptor**.
Get one event	GE	Enables fetching of one event.
Get parameter ID	PI	Calls **ctaGetParmID**.
Get server version	RV	Calls **ctaGetVersionEx**.
Get service version	SV	Calls **ctaGetServiceVersionEx**.
Hold call	OC	Calls **nccHoldCall**.
Loopback event	LE	Calls **ctaQueueEvent**.
Modify play gain	MG	Calls **vceSetPlayGain**.
Modify play speed	MS	Calls **vceSetPlaySpeed**.
Open services (new)	OV	Calls **ctaOpenServices**.
Open services (old)	OP	Calls **ctaCreateContext** and **ctaOpenServices**.
Place call	PC	Calls **nccPlaceCall**.
Play file	PF	Calls **vceOpenFile** and **vcePlayMessage**.
Play memory	PM	Calls **vcePlayMessage**.
Play/receive status	P?	Call **vceGetContextInfo**.
Play/receive stop	PS	Calls **vceStop**.
Play/record status	R?	Calls **vceGetContextInfo**.
Play/record stop	RS	Calls **vceStop**.
Query capability	CQ	Calls **nccQueryCapability**.
Query services	ES	Calls **ctaQueryServices**.
Record file	RF	Calls **vceCreateFile** and **vceRecordMessage**.
Record memory	RM	Calls **vceCreateMemory**, **vceEraseMessage**, and **vceRecordMessage**.
Refresh parameters	RP	Calls **ctaRefreshParms**.
Reject call	JC	Calls **nccRejectCall**.
Release call	RC	Calls **nccReleaseCall**.
Retrieve call	RR	Calls **nccRetrieveCall**.
Send digits	ND	Calls **nccSendDigits**.
Server shutdown	SH	Calls **ctaShutdown**.
Set call handle	IH	Specifies active and held call handles.
Set default server	SS	Calls **ctaSetDefaultServer**.
Set event handler	EH	Calls **ctaSetErrorHandler**.

Function	Command	Description
Set event source	SM	Calls **ctaSetEventSources**.
Set parameters	PD	Calls **ctaGetParmByName** and **ctaGetParmInfo**.
Set server trace	TR	Calls **ctaSetGlobalTraceMask**.
Set time slot	TS	Specifies an MVIP timeslot to open the NCC service.
Show state	X	Calls **adiGetContextInfo** and **nccGetLineStatus**.
Signal bit query	SQ	Calls **adiQuerySignalState**.
Start dial	SD	Calls **adiStartDial.**
Start protocol	SP	Calls **nccStartProtocol** or **adiStartProtocol**.
Start timer	ST	Calls **adiStartTimer**.
Stop event fetch	SE	Disables continuous fetch of events.
Stop protocol	UP	Calls **nccStopProtocol** or **adiStopProtocol**.
Supervised transfer	VT	Calls **nccTransferCall**.
Toggle pattern	TP	Calls **adiAssertSignal**.
Trace	W	Calls **ctaSetTraceLevel**.
Unblock calls	UC	Calls **nccUnblockCalls**.
View parameters	VD	Calls **ctaGetParmID** and **ctaGetParmInfoEx**.

You can enter commands while asynchronous functions execute, allowing you to execute multiple asynchronous functions concurrently or to stop functions. For example, you can run a tone detector (ET) and record voice (RF) simultaneously. Abort any of these functions by entering the respective stop command (DT and RS for tone and record).

To experiment with output and input functions simultaneously, execute two instances of *ctatest*. For example, to interactively experiment with tone generation and detection, start a tone detector in the first *ctatest* instance and a tone generator in the second *ctatest* instance. Use the *swish* MakeConnection command to make the QUAD connections to connect the two ADI service ports back-to-back. Refer to the *Switching Service Developer's Reference Manual* for more information about *swish*.

Natural Access parameters can be viewed or modified (VD, PD). If the *ctdaemon* is running, shared system global defaults are copied to the context. Otherwise, the local compiled defaults are copied. The commands prompt you to view or modify context parameters or global defaults.

ctatest writes trace messages to *ctdaemon* if *ctdaemon* is running. Use *ctdaemon* or **ctaSetTraceLevel** (W command) to set the global or local trace mask. To loop back an application event, use **ctaQueueEvent** (LE command).

Procedure

The following procedure assumes you are testing on an AG 2000 board that has a 2500-type telephone connected to one of the lines.

Complete the following steps to run *ctatest*:

Step	Action
1	Start *ctatest* by entering the following command at the prompt: `ctatest -b n -s [stream:]slot [-p protocol] [-l]` where ***n*** is the board number, ***[stream:]slot*** is the DSP stream and timeslot, and ***protocol*** is the name of the protocol appropriate for the functions that you want to test. A menu of one- and two-letter commands displays.
2	Enter CX to create a context. Specify a context name and service sharing mode.
3	Enter OV to open the service. Select a service from the list of available services. CTAEVN_OPEN_SERVICES_DONE displays on your screen.
4	Enter SP to start a protocol. Select a protocol appropriate for the functions you want to test. NCCEVN_STARTPROTOCOL_DONE displays on your screen. The port is now ready to accept functions.
5	Enter the commands you want to experiment with. The stop event fetch (SE), get one event (GE), and continue event fetch (CE) commands allow you to single step when issuing commands and retrieving events. These commands can be used to answer questions about state and event combinations. To select a new protocol, stop the active protocol (UP command), and enter the SP command again. All functions that are active when you stop the protocol terminate. NCCEVN_function_DONE or ADIEVN_function_DONE displays for each active function.
6	Enter DX to destroy the context. *ctatest* includes functions for retrieving commands from the keyboard and executing them. These functions include:

Function	Description
PerformFunc	Executes the keyboard commands and initiates Natural Access, NCC service, or ADI service functions.
Main	Processes options, creates Natural Access event queue, and goes into **ctaWaitEvent** loop.
MyPlayAccess	Voice play callback.
MyRecAccess	Voice record callback.
MyEventHandler	Displays the event and performs post-event processing (for example, closes files, prints extra information).

Inbound call control: incta

Demonstrates:

- Call control answer and release. Refer to the *Natural Call Control Service Developer's Reference Manual* for more information about call control.
- Synchronous programming model for one port in one process.
- Play and record using the Voice Message service. Refer to the *Voice Message Service Developer's Reference Manual* for more information.
- Context sharing and object descriptor hand off (when used with *csplayrc*). For more information, refer to *Service sharing on a local server: csplayrc* on page 195.

Usage

```
incta [options]
```

where ***options*** include:

Option	Use this option to...
-A ***xxxmgr***	Specify the Natural Access service manager. Default: ADIMGR
-b ***n***	Specify the board number ***n***. Default: 0
-F ***filename***	Specify the name of the Natural Access configuration file. Default: *cta.cfg*
-i ***i***	Specify the number of iterations (***i***) before exit. **Note:** You can also stop the program by entering **Ctrl+C** under Windows and Solaris.
-l	Show low-level events.
-p ***protocol***	Specify the protocol to run. Default: lps0
-s [***stream:***]***slot***	Specify the port (DSP) address. Default: 0:0

When running *csplayrc*, options include:

Option	Use this option to...
-C	Run *incta* in server mode. Specify this option only if you are using *incta* with *csplayrc*.
-m ***contextname***	Specify the context name for the communication context. Default: commn-cxt
-x ***contextname***	Specify the context name for call control/voice play and record processing. Default: incta-demo

For more information about using these options, refer to *Service sharing on a local server: csplayrc* on page 195.

Featured functions

adiCollectDigits, **nccAnswerCall**, **nccGetCallStatus, nccQueryCapability**, **nccRejectCall**, **nccReleaseCall**, **nccDisconnectCall**, **vceCreateFile**, **vceOpenFile, vcePlayList**, **vceRecordMessage**

Also, (when used with *csplayrc*):

ctaAttachContext, **ctaCreateContextEx**, **ctaCreateQueue**

Description

This description applies to the wink-start protocol wnk0.

When you take the handset off-hook, the TCP acknowledges the line seizure by generating a wink.

After the wink, enter three DTMF digits as the address. If the call connects, the demonstration proceeds into the conversation state and you are prompted to record voice, play back your recorded sample, or hang up.

Procedure

The following procedure assumes you are testing on an AG 2000 board that has a 2500-type telephone connected to one of the lines.

Complete the following steps to run *incta*:

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Start incta by entering the following command at the prompt:
<code>incta -b n -s [stream:]slot -p wnk0 -i i</code></td></tr>
<tr><td>2</td><td>Specify the board and timeslot for the handset you are using.</td></tr>
<tr><td>3</td><td>Take the handset off-hook. The wink-start is acknowledged.</td></tr>
<tr><td>4</td><td>Enter three DTMF digits as the address.
<table>
<tr><th>If the first digit is...</th><th>Then the demonstration releases the call...</th></tr>
<tr><td>8</td><td>By playing a reorder tone.</td></tr>
<tr><td>9</td><td>With a busy tone.</td></tr>
<tr><td>0</td><td>By playing a special information tone.</td></tr>
</table>
If the first digit is not 8, 9, or 0, the call connects and the demonstration proceeds into the conversation state. You are prompted to record voice, play back your recorded sample, or hang up.</td></tr>
<tr><td>5</td><td>Select the record voice option.
You hear the record beep prompt.</td></tr>
<tr><td>6</td><td>Begin speaking after the prompt.
After you stop speaking, the record operation terminates and you return to the options menu.</td></tr>
<tr><td>7</td><td>Select the playback option.
The sample you just recorded plays back. You return to the options menu.</td></tr>
<tr><td>8</td><td>Select the hang-up option.</td></tr>
</table>

To stop the program, enter **Ctrl+C** for Windows and UNIX.

incta demonstrates synchronous programming on a single port. A start command is initiated and the demonstration continuously retrieves events from the board until the expected event returns, simplifying the overall program flow.

This demonstration can be used with various protocols or configurations. If the address is not available to it, skip the first two parts (selective rejection of calls and recitation of the called number), and go to the play/record loop.

Placing and receiving calls: inoutcta

Use *inoutcta* to:

- Demonstrate the operation of all NMS CAS call control TCPs, either on a live trunk (placing and receiving calls while connected to the PSTN) or through the MVIP bus (for testing purposes on digital trunks only).
- Provide an example of C code implementation of a two-way trunk application (that can both place and receive calls) using Natural Access.

Usage

```
inoutcta [options]
```

General Options

Use this option...	To...
-?	Display a Help screen and terminate *inoutcta*.
-b ***board_#***	Specify the board number. Default: board 0
-p ***protocol***	Specify the name of the TCP. Refer to the *NMS CAS for Natural Call Control Developer's Manual* for a list of available protocols.
-s ***slot***	Specify the MVIP timeslot. Default is 0. The MVIP stream is automatically selected by the program, according to the type of board it is running on.
-t ***number***	Exploit the multi-threaded Natural Access programming model. A number of threads are created that either place or receive calls. The first thread starts on the timeslot specified with the -s option. The remaining threads use the subsequent timeslots.
-v ***level***	Specify the verbosity level of messages printed on screen. ***level*** can be any of the following values: **Value** / **Description** 0 – Display error messages only. 1 – Display errors and unexpected high-level events. 2 – Display errors and all high-level events (default). 3 – Display errors and all events.

Inbound options

<table>
<tr><th>Use this option...</th><th>To...</th></tr>
<tr><td>-a mode</td><td>Accept the inbound calls before answering or rejecting. Valid values for mode include:
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>1</td><td>Accept and play ring back.</td></tr>
<tr><td>2</td><td>Accept and remain silent until further command.</td></tr>
<tr><td>3</td><td>Accept with user audio and play a voice file.</td></tr>
<tr><td>4</td><td>Accept with user audio and detect DTMFs.</td></tr>
</table>
Note: Modes 2, 3, and 4 are not supported for analog trunks.</td></tr>
<tr><td>-B rate</td><td>Specify the rate of billing. Applies to digital trunks only. Valid values for rate include:
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>0</td><td>Free call.</td></tr>
<tr><td>N</td><td>Normally billed call.</td></tr>
</table></td></tr>
<tr><td>-g</td><td>Emulate a gateway application. The user controls the timing of the incoming call that is accepted or rejected. Applies to digital trunks only.</td></tr>
<tr><td>-j mode</td><td>Reject the inbound calls. Valid values for mode include
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>1</td><td>Reject by playing reorder tone or equivalent protocol signal.</td></tr>
<tr><td>2</td><td>Reject by playing busy tone or equivalent protocol signal.</td></tr>
<tr><td>3</td><td>Reject by playing ring tone.</td></tr>
<tr><td>4</td><td>Reject by playing user audio.</td></tr>
</table></td></tr>
<tr><td>-i</td><td>Enable interactive dialing (valid for outbound or trunk protocols only).</td></tr>
<tr><td>-r rings</td><td>Specify the number of ring tones to play before answering a call. Default: 2.</td></tr>
</table>

Outbound options

Use this option...	For...
-n ***number_to_dial***	Outbound or two-way protocols only. Default: 123 ***number_to_dial*** must be formatted appropriately for the TCP.

Hardware requirements

inoutcta can work with any NMS board with a processor and DSPs.

Note: Some TCPs supported by *inoutcta* do not work with all boards.

Software requirements

To use *inoutcta*, you need Natural Access and a protocol. Refer to the *NMS CAS for Natural Call Control Developer's Manual* for a description of the protocols.

Description

To place and receive calls, *inoutcta* uses CAS protocols such as the MFC-R2 TCP (*mfc0.tcp*). The MFC-R2 TCP can handle inbound calls, receive a dial command, and seize the line for an outbound call. Optionally, the demonstration program can act as either an inbound application or an outbound application, for use on one-way lines.

inoutcta performs the following actions:

Step	Action
1	It loads the user option parameters that the TCP will use (the parameter files provided with NMS CAS). The parameter files are • adi**xxx**.par • nccx**xxx**.par where ***xxx*** represents the first three characters of the protocol. *inoutcta* also loads the following parameter files: • ncc**x**cas.par • ncc**x**adi.par These files are located in the *nms**ctaccess**cfg* directory.
2	It starts the TCP specified on its command line, configuring it using the parameters loaded in Step 1.
3	It either places a call or waits for one, depending on the -i command line argument.
4	If commanded to dial out (outbound behavior), it dials either automatically or interactively, depending on the -i command line argument. If the demonstration program is commanded to wait for a call (inbound behavior): • It waits for an incoming call. • When a call is received, the program prints out the received call information (such as caller ID information on analog trunks, if enabled). If instructed by the user, it accepts the call through one of the standard call acceptance procedures. • The program answers or rejects the call according to the -j ***mode*** command line option. • If the call is answered, the program plays back the digits received. • The program plays another prompt asking the user for a DTMF specifying the action to take next: playing a voice file, recording a voice file, or hanging up. • The program starts the DTMF detector and waits for a tone. If the tone does not come, it hangs up. • If the tone arrives, the demonstration program performs the action that the tone specifies, then hangs up.

Step	Action
5	If the demonstration program seizes the line first (outbound behavior), the program takes the following actions: • It dials the number specified by the user. • When the call is answered, it starts recording. • When silence is detected, it stops recording and plays the DTMF tone meaning "record" for its inbound counterpart (for back-to-back operations). **Note:** To run the *inoutcta* application in back-to-back mode on analog lines, make connections through the switch or the PBX to obtain the loop start signaling necessary for the loop start TCP. • The program starts playing back the newly recorded voice file. • When playing is completed, it hangs up. *inoutcta* hangs up if the caller hangs up at any time. *inoutcta* displays the Natural Access messages it receives on the screen according to the verbosity level (-v command line option).

The following table lists the interactions between two *inoutcta* applications in which one acts as an inbound application, and the other as an outbound application:

Outbound		Inbound
Place call.	→	Wait for call.
	←	Optionally, may accept the call.
Get connected.	←	Answer call.
Record.	←	Speak prompt on both analog and digital trunks and digits on a digital trunk.
Send DTMF.	→	Detect DTMF.
Play back recorded file.	→	Record voice.
Wait for hang up, or hang up.	↔	Wait for hang up, or hang up.

The demonstration program illustrates both the one-context-per-process Natural Access programming model and the multi-threaded model with one context per thread. The functionality does not change in either case.

Program structure and coding features

inoutcta is based on the asynchronous programming model implemented by Natural Access. Some function calls in the program are found in the *ctademo* library supplied with Natural Access to provide users with examples of how to use the APIs.

ctademo functions have names starting with **Demo**. For example, **DemoHangNCCup** uses **nccDisconnectCall** and then waits for the result of the call (either SUCCESS or DISCONNECT) before returning. Most *ctademo* functions are wrappers that enclose the corresponding NCC service function or ADI service function and wait for an event to terminate the function, either by signaling that the function was successful or by signaling a failure. In this way, they transform an asynchronous wait for an event into a synchronous function call.

main function

The **main** function of *inoutcta* first defines the list of Natural Access service managers that the application needs. Then it parses its command line arguments, and assigns the corresponding values to its variables. The function then checks that none of the user options are inconsistent with each other, and warns the user if it finds a problem. Then it registers an error handler with Natural Access and starts Natural Access.

main then queries the board to learn the MVIP stream DSPs are on, and launches the demonstration loop in one of two ways, depending on whether -t (the use threads option) is specified on the command line.

If the -t option is not specified, the demonstration program calls the **RunDemo** function. Otherwise, the demonstration program launches **RunDemo** a number of times, as many as the argument to the -t option. All threads share the same parameters.

Opening the driver, context, and protocol

In *inoutcta*, **RunDemo** defines the list of Natural Access services needed by the thread. It then calls **DemoOpenPort**, a function from the *ctademo* library. **DemoOpenPort** opens the Natural Access application queue, attaching all defined service managers. It then creates a context and opens the defined services on the new queue.

RunDemo then calls **ctaLoadParameterFile**, a function that loads a parameter file containing parameters to configure the TCP for its specific implementation.

Parameter files are found in the following locations:

Operating system	Directory
Windows	*\nms\ctaccess\cfg*
UNIX	*/opt/nms/ctaccess/cfg/*

If the parameter file is not found, **RunDemo** uses the TCP's default values.

Next, **RunDemo** modifies one or more parameters in the standard NCC service or ADI service start parameter structures. To do so, it uses **ctaGetParms** to retrieve the current defaults, and then explicitly sets the parameter presented in the following table. It then starts the TCP on the opened context. To do so, it calls **nccStartProtocol** with

- A NULL pointer for the TCP-specific parameter structure argument, so that the TCP uses the TCP-specific parameters just loaded into the process parameter space.
- A pointer to the changed NCC_ADI.START structure to load these parameters.

The following Natural Access parameters change:

Parameters	Description
NCC.X.ADI_START.mediamask	Tells the TCP what functions are to be started and ready for the application to use in the conversation state. Functions that can be started by the TCP include DTMF detection, energy detection, silence detection, and echo cancellation.
NCC.START.eventmask	Programs the TCP to send to the application or retrieve a number of informational events. These events can be useful to determine the state of calls.

inoutcta enables all informational events. The informational events are:

Event	Description
NCCEVN_CALL_PROCESSING	For outbound calls. Reports when all the digits have been delivered to the telephone network.
NCCEVN_REMOTE_ALERTING	For outbound calls. Reports when the distant terminal is ringing.
NCCEVN_REMOTE_ANSWERED	For outbound calls. Reports when the remote terminal has answered.
NCCEVN_BILLING_INDICATION	For outbound calls. Reports billing pulses. These are signaling pulses that can be received from a PSTN switch while the call is connected to signal that a unit of cost has been billed to the call. (Applies to digital trunks only.)
NCCEVN_CALL_STATUSUPDATE	Reports asynchronous changes on digital trunks in the NCC_CALL_STATUS structure. This happens because of network events that the application may need, but does not modify the state of the call. (Applies to digital trunks only.)

MyReceiveCall and MyPlaceCall

Once the protocol is started on the board, depending on the type of TCP, **RunDemo** is ready to accept incoming calls or dial outgoing calls. **MyReceiveCall** is invoked repeatedly. This function performs all call control and hangs up the line when the call is completed.

If the TCP is only outbound (only places outgoing calls), the loop does the following:

- If the -i option (for interactive dialing) is specified on the command line, the loop prompts the user to press a key, and starts waiting for the keyboard event. When the user presses a key, the loop calls the function **MyPlaceCall** to dial out a call.
- If the -i option is not specified, the loop immediately calls the function **MyPlaceCall** to dial out a call.

If the TCP is bidirectional, the call comes in while the loop is waiting for the user to press a key. **MyReceiveCall** is called since this is a legal event for a bidirectional trunk.

MyReceiveCall

MyReceiveCall performs the following actions:

1. Checks to see why it has been invoked.

   If **MyReceiveCall** was invoked because an incoming call was detected while the program was waiting to place an outgoing call, there is no need to wait further for a call. **MyReceiveCall** queries Natural Access to get information about the incoming call, by invoking **nccGetCallStatus**.

   The information includes the following:

Information	Description
called number	Called number digits.
calling party number	ANI (automatic number identification) digits.
calling name	Calling party name.
user category	For MFC-R2, MFS, EAM, and R1.5 TCPs, shows the user category of the calling party (for example, normal subscriber, operator, test equipment).
toll category	Usually the same as the user category, but, in some countries, it can have a different meaning (toll category); for MFC-R2 and EAM TCPs only.
calling number presentation indicator	ANI digits might not be available because of interworking of different protocols with different features in the call path or ANI presentation might be restricted. This value is used even when no ANIs are present, when the TCP does not support ANI reception, and when the TCP parameters specify that ANIs are not requested. Possible values include: 0 = Calling number presentation allowed (default) 1 = Calling number presentation restricted 2 = Calling number not available

   These attributes are stored in the NCC_CALL_STATUS data structure. The fields that need to be completed will vary depending on the protocol and the country. For more information, refer to the *NMS CAS for Natural Call Control Developer's Manual*.

   If a purely inbound TCP is active and **MyReceiveCall** is called, or if the TCP signals to the application that the line has been seized and that call setup has begun, the application must wait for the incoming call setup to be completed. The function does so by waiting for the appropriate event (NCCEVN_INCOMING_CALL). Once this event is detected, the function calls **nccGetCallStatus** to get the digit information.

   On digital trunks, while waiting for NCCEVN_INCOMING_CALL, **MyReceiveCall** may receive the NCCEVN_RECEIVED_DIGIT event which indicates that a digit has arrived in the queue. **MyReceiveCall** displays a message to the screen that indicates the digit.

   MyReceiveCall may also receive NCCEVN_CALL_DISCONNECTED, indicating that the calling party has hung up. **MyReceiveCall** jumps to the label hangup_in to hang up the call.

2. On digital trunks, **MyReceiveCall** can optionally set the billing rate of the incoming call. This is not supported by all CAS protocols. For more information, refer to the *NMS CAS for Natural Call Control Developer's Manual*. The TCP replies with one of the following events:

Error	Description
NCCEVN_BILLING_SET	The billing rate has been set. Refer to the value field to check if the network accepted the requested billing rate.
NCCEVN_PROTOCOL_ERROR	The protocol does not support the set billing operation.
NCCEVN_SEQUENCE_ERROR	The protocol supports the set billing operation, but the operation was requested at a time when it was no longer allowed by the protocol, for instance after the call was accepted (with the default billing indication).

3. **MyReceiveCall** can optionally accept the call before answering or rejecting. The three modes of accept are:

Mode	Description
NCC_ACCEPT_PLAY_RING	Accepts the call and plays ring indefinitely. This may be important for applications that intend to answer the call, but need time to process the information associated with the incoming call. This is the only available option for analog loop start TCPs.
NCC_ACCEPT_USER_AUDIO	Accepts the call and plays a voice message on the line. This may be necessary for IVR applications that want to explain features of the service offered, without billing the caller. An option of this accept mode is also demonstrated, that of detecting DTMF tones at this stage. This might not be allowed by the network, as typically at accept time the voice path is connected only in one direction, from the NMS board to the caller, and not in the other direction. This option is not supported for analog protocols.
NCC_ACCEPT_PLAY_SILENT	Accepts the call and remains silent. Use this option for trunk-to-trunk switching applications. This option is not supported for analog protocols.

MyReceiveCall answers the call or rejects it depending on the -j command line option.

-j Argument	Action
1 = NCC_REJ_PLAY_REORDER	Rejects the call using **DemoRejectCall**, directing the TCP to signal that the number called is not allocated.
2 = NCC_REJ_PLAY_BUSY	Rejects the call using **DemoRejectCall**, which calls **nccRejectCall** directing it to signal that the line is busy. The MFC-R2, MFS, and EAM TCPs do this by sending a special MF tone in the compelled sequence. (Other TCPs use methods that are protocol-specific.) **DemoRejectCall** waits for the event signifying that the caller has hung up (NCCEVN_CALL_DISCONNECTED).
3 = NCC_REJ_PLAY_RINGTONE	Rejects the call by playing ring until the other side hangs up.
4 = NCC_REJ_USER_AUDIO	Rejects the call using **CustomRejectCall**, which plays a special information tone (SIT). **CustomRejectCall** defines the SIT and calls **nccRejectCall** with the argument NCC_REJ_USER_AUDIO. This call tells the TCP to reject the call and open the DSP resource to the application. The custom audio can then be played on the line. **CustomRejectCall** plays the defined SIT and monitors the event queue for NCCEVN_TONES_DONE with reason NCC_REASON_RELEASED, and NCCEVN_CALL_DISCONNECTED. When both events are received, the function terminates.
No -j option is specified	Answers the call. It calls **nccAnswerCall** with an argument that makes the TCP play three rings before answering automatically. **MyReceiveCall** waits for a user's keystroke. If the keystroke is detected, **MyReceiveCall** calls **nccAnswerCall** again. The TCP answers the call immediately.

4. If **MyReceiveCall** answers the call, it allows the demonstration program to talk either with a human being on a telephone or with a corresponding outbound application. **MyReceiveCall** performs the following operations:
    - Plays voice files announcing the called number and the calling number, if available. Applies to digital trunks only.
    - Plays another voice file with a menu, prompting the user to enter a DTMF tone from the keypad to do one of the following: record a voice file, play a previously recorded voice file, or hang up.
5. To play voice files, it calls the **CTADEMO** function **DemoPlayFile**, which opens a voice file and calls **adiStartPlaying** to play it. **MyReceiveCall** does the following:
    - Waits for a DTMF tone to be detected. To do so, it calls **DemoPromptDigit**, which sets three-second timeouts between digits, and calls **adiCollectDigits** twice in an attempt to get the digits. If a DTMF is not detected, **MyReceiveCall** hangs up.
    - If a DTMF digit is detected, **MyReceiveCall** does what the DTMF tones direct it to do, and then hangs up. (If the command was to hang up, it hangs up immediately.) If no voice file exists, **MyReceiveCall** plays a default voice file (that is, there is nothing to play).

MyPlaceCall

MyPlaceCall performs the following actions:

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Tries to place a call to the number that the user specified in the command line.
To place the call, MyPlaceCall invokes nccPlaceCall with the digits to call, and waits for events. Possible events include:
<table>
<tr><th>Event</th><th>Description</th></tr>
<tr><td>NCCEVN_PLACING_CALL</td><td>Placing a call.</td></tr>
<tr><td>NCCEVN_CALL_PROCEEDING</td><td>All digits delivered to the network.</td></tr>
<tr><td>NCCEVN_REMOTE_ALERTING</td><td>Outbound ring detected.</td></tr>
<tr><td>NCCEVN_STATUSINFO_UPDATE</td><td>Call status message received. Applies to digital trunks only.</td></tr>
<tr><td>NCCEVN_REMOTE_ANSWERED</td><td>Outbound answer detected.</td></tr>
<tr><td>NCCEVN_CALL_DISCONNECTED</td><td>Call control is in disconnected state.</td></tr>
<tr><td>NCCEVN_CALL_CONNECTED</td><td>Call control is in connected state.</td></tr>
</table></td></tr>
<tr><td>2</td><td>When the call is connected, MyPlaceCall starts recording whatever is on the line, waiting for a period of silence to stop recording. This operation is designed both to interact with a human being ("Hello") and to interact with the MyReceiveCall function, which speaks a little longer.
To record the voice on the line, MyPlaceCall calls vceRecordMessage, which opens a voice file to receive the data and then invokes adiStartRecording.</td></tr>
<tr><td>3</td><td>If silence is detected, MyPlaceCall first sends a DTMF tone, by invoking adiStartDTMF. This is strictly to interact with MyReceiveCall. The DTMF tone is the one that tells it to start recording.</td></tr>
<tr><td>4</td><td>MyPlaceCall invokes MyPlayMessage to play back what was just recorded. When playback is finished, MyPlaceCall calls DemoHangUp to hang up.</td></tr>
<tr><td>5</td><td>On digital trunks, if billing pulses are detected, MyPlaceCall displays a message for each pulse, with a pulse count (if billing pulse detection is enabled).</td></tr>
</table>

Using WaitForAnyEvent

WaitForAnyEvent waits for an event to arrive from an event queue in use by the application. The function is operating system specific, with #ifdefs used to differentiate among different operating systems. Although Natural Access provides an operating system independent function to collect events, the demonstration application must still collect the user's keystrokes. Collecting user keystrokes must be performed differently on different operating systems.

Outbound call control: outcta

Demonstrates:

- Placing an outbound call. Refer to the *Natural Call Control Developer's Reference Manual* for more information about placing an outbound call.
- Simple glare detection.
- Synchronous programming model for one port in one process.
- Play and record using the Voice Message service. Refer to the *Voice Message Service Developer's Reference Manual* for more information.

Usage

```
outcta [options]
```

where ***options*** are:

Option	Use this option to...
-A ***xxxmgr***	Specify the Natural Access service manager. Default: ADIMGR
-b ***n***	Specify the board number ***n***. Default: 0
-d ***digits***	Specify the number to dial. Default: 123
-F ***filename***	Specify the name of the Natural Access configuration file. Default: *cta.cfg*
-l	Print low-level call control events.
-s [***stream:***]***slot***	Specify the port (DSP) address. Default: 0:0.
-p ***protocol***	Specify the protocol to run. Default: lps0
-r ***tries***	Specify the maximum number of dial attempts or retries on failure.

Featured functions

adiCollectDigits, **vceCreateFile**, **vceOpenFile**, **vcePlayList**, **vceRecordMessage**

Description

The demonstration program seizes the line and places a call. For the wink-start protocol, the program must receive a wink (momentary off-hook) before it dials the digits.

When the demonstration detects an answer (off-hook indication), it plays a greeting message, then prompts the called party to perform one of the following actions:

- Record a message
- Play back the message
- Hang up

Procedure

The following procedure assumes you are testing on an AG 2000 board that has a 2500-type telephone connected to one of the lines.

Complete the following steps to run *outcta*:

Step	Action
1	Change to the directory containing the *outcta* program and voice files.
2	Start *outcta* by entering the following command at the prompt: `outcta -b n -s [stream:]slot -p wnk0 [-d digits -r tries -l]` Use the -l option if you want to see all call control events. *outcta* displays the message: "Placing a call to 'xxx' ..."
3	Lift the handset and replace it to simulate a wink from the network. NCCEVN_PLACING_CALL displays as the NCC service dials the digits. If you specified the -l command line option, NCCEVN_CALL_PROCEEDING displays when dialing is complete.
4	Lift the handset to simulate network answer. You hear the greeting message and the menu message.
5	Dial 1 to record a message.
6	Begin speaking after the beep. Recording ceases shortly after you stop speaking. You hear the menu message again.
7	Dial 2 to play back what you recorded.
8	Dial 3 to quit. The demonstration hangs up the line and *outcta* exits.

Note: *outcta* demonstrates synchronous programming of a single port. A start command is initiated and the demonstration continuously retrieves events from the board until the expected event returns, simplifying the overall program flow.

Controlling the takeover program: takecmd

Controls the service object sharing demonstration program, *takeover*. Use *takecmd* to:

- Stop instances and modify the runtime mode of the *takeover* demonstration program.
- Start and stop the *takeover* DTM functions.
- Start and stop the *takeover* OAM function.
- Start and stop the *takeover* call control and voice functions.

Usage

```
takecmd [options]
```

where ***options*** are the following values:

Option	Use this option to...
-c ***host_name***	Specify the name or address of the host on which the Natural Access Server (*ctdaemon*) is running. Default: localhost
-f ***filename***	Specify the name of a file containing the command input.

Featured functions

ctaCreateContext, **ctaQueueEvent**

Description

Refer to *Sharing service objects: takeover* on page 219 for information on how *takecmd* interacts with the *takeover* demonstration program.

Sharing service objects: takeover

Demonstrates a Natural Access client application that passes control of a service object to a peer application. Use *takeover* to

- Operate with a second instance of *takeover* so that when one instance exits, the second instance continues to perform operations.
- Interact with service managers running on remote Natural Access Servers (*ctdaemon*).

Usage

```
takeover [options]
```

where ***options*** are the following values:

Option...	Use this option to...
-@ ***host_name***	Specify the name or IP address of the host on which the Natural Access Server (*ctdaemon)* service managers are running. Default: localhost
-1	Specify to run *takeover* in primary mode. If this option is not specified, the instance with the lowest process identifier (PID) runs in primarymode.
-b ***board_number***	Specify the OAM board number. Default: 0
-c ***host_name***	Specify the name or IP address of the host on which to create the command and checkpoint contexts. Default: localhost
-d ***name***	Specify the name of the board switching driver. Default: agsw
-e ***encoding_number***	Specify a voice encoding selector contained in *adidef.h*, for example, ADI_ENCODE_NMS_32. Default: 3
-f ***name***	Specify the name of the voice file used for playing messages. Default: *american.vox*
-p ***protocol***	Specify the protocol to run. Default: wnk0
-r [***n:***]***m***	Specify the called DSP resource [stream and] timeslot. Only required if the -s timeslot is not the same. Default: 16:0 or 4:0. (Board dependent)
-s [***n:***]***m***	Specify the caller line [stream and] timeslot. Default: 0:0
-v	Display verbose event output.

Featured functions

ctaAttachObject, **ctaDetachObject**, **ctaCreateContext**, **ctaGetObjDescriptor**, **dtmAttachTrunkMonitor**

Description

By using the shared context and the shared object handle capabilities provided in the Natural Access development environment, independently operating applications can share responsibility for overseeing and managing service resources, such as active telephone calls. In such an environment, if one of these applications fails, its cooperating peer can take over responsibility for the control and management of the service resource.

This demonstration program illustrates how applications can share responsibility for the control of Natural Access service objects. This program also illustrates the ability to run contexts on remote Natural Access servers, whereby the application can run

on one host and the service managers that are being used by the application can be running on another host.

Takeover application

This program consists of two applications, *takeover* and *takecmd*. *takeover* is the main demonstration program that opens Natural Access services and exchanges checkpoint information with another running instance of *takeover*. Each instance of *takeover* is started independently.

Takecmd application

Use the second application, *takecmd*, to control the behavior of *takeover*. For example, from the *takecmd* interface, you can tell *takeover* to start playing a file on a connected call or tell the two instances of *takeover* to switch roles (the backup instance becomes the primary instance, and the primary instance becomes the backup).

Note: The *takecmd* interface is similar in style to the *ctatest* interface. Refer to *Interactive test: ctatest* on page 198 for more information about *ctatest.*

Primary and backup mode

takeover operates either in primary mode or in backup mode. At startup, set the -1 option to indicate which instance of *takeover* you want to run in primary mode. If the option is not set, an internal algorithm dynamically determines which instance runs in primary mode.

When *takeover* is run in backup mode, it can request the current state of the primary instance and continue operating from this derived state. That is, the second instance of *takeover* does not need to be started immediately after starting the primary instance. It is possible to start the primary instance, get it to the state of performing some function such as playing a file, and then start the backup instance of *takeover*. If the primary instance of *takeover* then exits, the backup instance continues to play the file, even though it had not been running at the time you invoked the initial command to play the file.

The backup instance receives periodic events or heartbeats from the primary instance. If the backup instance doesn't receive them, it takes over as the primary instance.

takeover prevents more than one instance from running in primary mode, but enables more than one instance to run in backup mode. If the instance of *takeover* running in primary mode exits, for example, when **Ctrl+C** is entered, the remaining instances dynamically determine which of the remaining instances will take over the role as the primary instance and make the necessary adjustments.

Checkpointing

For independent applications to share the management of service objects, runtime information must be exchanged between the applications at certain times. This exchange of data is referred to as checkpointing. Checkpointing is necessary when a new instance of a service object is created, for example, if you make a call, or if the state of an operation changes.

Cooperating applications can exchange checkpoint information through standard file input/output, shared memory, or a private IPC channel. In this demonstration program, Natural Access application events are used for the exchange of checkpoint information. The use of these events allows for flexibility in the location of the

running applications. For example, if a shared memory method is chosen, the running applications must be located on the same server.

The checkpoint information exchanged by an application consists of object descriptors and general state data. The format and content of the information exchanged is specific to the thread performing the exchange.

General architecture

takeover consists of five threads, each of which operates on a unique, shared context. The following table describes each of the threads and the context on which they operate:

Thread	Context	Description
ccVoice	TakeOverCcV	Performs call control and voice operations and sets up switch connections using the NCC, ADI, and SWI services.
checkPoint	Exchange	Manages the exchange of checkpoint data with its running peer. This thread is also responsible for generating or monitoring heartbeat events.
dispatchCommand	TakeCmd	Hands off events from the *takecmd* process to the appropriate thread.
manageBoards	TakeOverOam	Stops and starts boards using the OAM service. It is also responsible for periodically performing OAM get keyword operations.
monitorTrunks	TakeOverDtm	Uses the DTM service to show trunk status information.

Procedure

To run the *takeover* demonstration program, open three windows: one for the primary instance of *takeover*, one for the backup instance of *takeover*, and one for the *takecmd* control program.

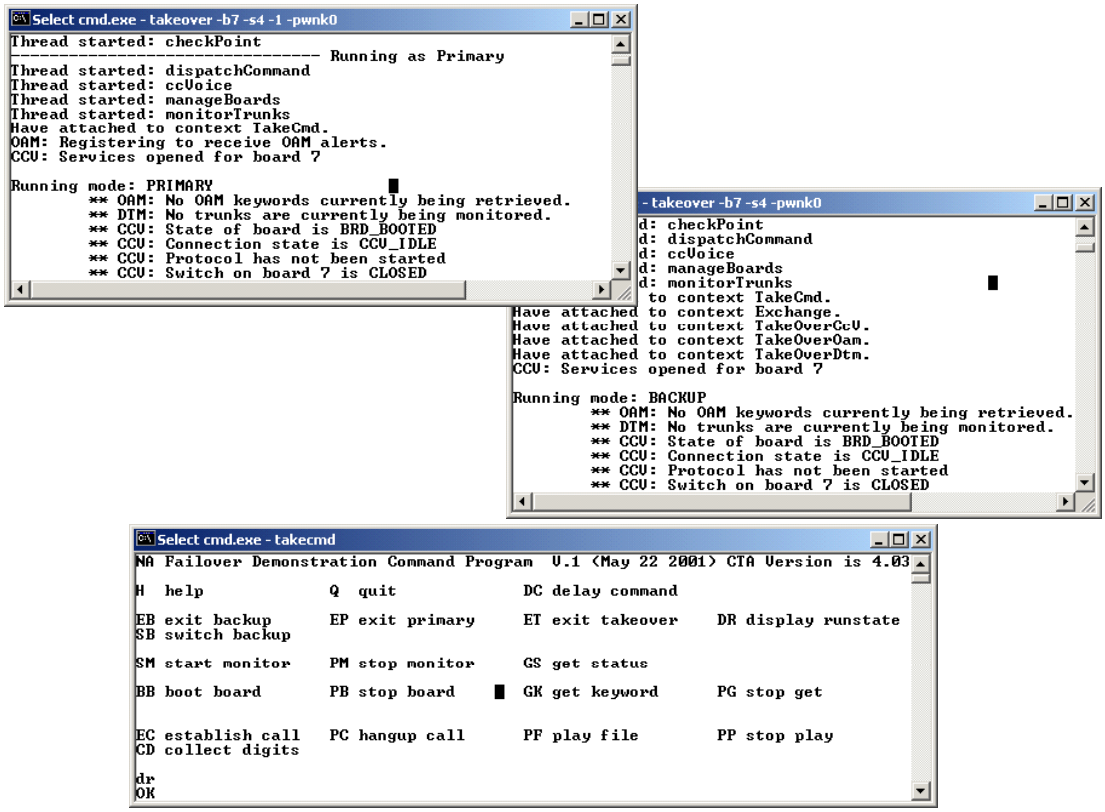
By using the command line options for *takeover*, you can run this demonstration program on two separate Natural Access Servers. One server (-c ***server_name***) can control the TakeCmd and Exchange contexts. These contexts are overhead contexts. When you enter commands at the *takecmd* prompt, *takecmd* uses the shared context TakeCmd to transfer command information to the *takeover* program. The information is transferred through the application events. Similarly, the checkPoint thread uses the Exchange context to exchange data between the primary and backup instances of *takeover*.

The second server (-@ ***server_name***), indicates where the Natural Access Server (*ctdaemon*) that controls the service contexts (for example, DTM, OAM, NCC) is running. This server can be a separate computer.

The following procedure assumes that you are testing on a chassis with a CG 6000 board (board 0) and an AG 2000 board (board 7). The AG 2000 has a DID hybrid on port 4 with a 2500-type telephone connected to it.

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Open the first window and enter the following command at the prompt:
<code>takecmd</code>
This command brings up the application that controls the behavior of takeover.</td></tr>
<tr><td>2</td><td>Open the second window and enter the following command:
<code>takeover -1 -b 7 -s 4 -p wnk0</code>
This command runs the takeover application as the primary instance where:
<table><tr><th>Command</th><th>Description</th></tr><tr><td>-l</td><td>Specifies the primary instance.</td></tr><tr><td>-b 7</td><td>Specifies an NMS OAM board number of 7.</td></tr><tr><td>-s 4</td><td>Specifies a caller line timeslot of 4.</td></tr><tr><td>-p wnk0</td><td>Specifies the wink-start protocol.</td></tr></table></td></tr>
<tr><td>3</td><td>Open the third window and enter the following command:
<code>takeover -b 7 -s 4 -p wnk0</code>
This command runs the takeover application as the backup instance where:
<table><tr><th>Command</th><th>Description</th></tr><tr><td>-b 7</td><td>Specifies an NMS OAM board number of 7.</td></tr><tr><td>-s 4</td><td>Specifies a caller line timeslot of 4.</td></tr><tr><td>-p wnk0</td><td>Specifies the wink-start protocol.</td></tr></table></td></tr>
</table>

The following illustrations provide an example of running primary and backup applications:



Note: Each instance of *takeover* reports the threads as they are created. If you enter the command DR, each instance of *takeover* reports its current runtime state.

Configurations

You can run this demonstration program in a number of configurations. The following tables provide sample configurations.

Option 1: Four separate hosts named East, West, North, and South.

Host	Start up command	Description
East	takecmd	Run the application that modifies the behavior of *takeover.* This host is currently running Natural Access Server (*ctdaemon*).
West	takeover -l -c east -@ south	Run the primary instance of *takeover.* The service managers are running on South, and *takecmd* is creating the command context on East.
North	takeover -c east -@ south	Run the backup instance of *takover*. The service managers are running on South, and *takecmd* is creating the command context on East.
South		This host is currently running Natural Access Server (*ctdaemon*).

Option 2: Three separate hosts named West, North, and South.

Host	Start up command	Description
West	takecmd takeover -l -@ south	Run the application that modifies the behavior of *takeover.* Run the primary instance of *takeover. takecmd* is creating the command context on localhost (West). This host is currently running Natural Access Server (*ctdaemon*).
North	takeover -c west -@ south	Run the backup instance of *takeover*. The service managers are running on South, and *takecmd* is creating the command context on West.
South		This host is currently running Natural Access Server (*ctdaemon*).

Option 3: Three separate hosts named East, North, and South.

Host	Start up command	Description
East	takecmd	Run the application that modifies the behavior of *takeover.* This host is currently running Natural Access Server (*ctdaemon*).
North	takeover -l -c east -@ south takeover -c east -@ south	Run the primary instance of *takeover.* The service managers are running on South, and *takecmd* is creating the command context on East. Run the backup instance of *takeover*. The service managers are running on South, and *takecmd* is creating the command context on East.
South		This host is currently running Natural Access Server (*ctdaemon*).

Option 4: One host named West.

Host	Start up command	Description
West	takecmd takeover -l takeover	Run the application that modifies the behavior of *takeover.* Run the primary instance of *takeover. takecmd* creates the command context. Run the backup instance of *takeover*. This host is currently running Natural Access Server (*ctdaemon*).

General operating commands

Use the following *takecmd* commands to operate the *takecmd* and *takeover* demonstration programs.

Function	Command	Description
Help	H	Displays table of available commands within the *takecmd* interface.
Quit	Q	Exits from the *takecmd* program.
Exit primary	EP	Exits from the *primary takeover* instance.
Exit backup	EB	Exits from the backup *takeover* instance.
Exit takeover	ET	Exits from both instances of *takeover,* as well as *takecmd*.
Delay command	DC Sleep seconds	Pauses a specified number of seconds before executing the next command. Use this command when entering commands from a file, and a delay is needed to allow certain *takeover* states to be reached.
Display runtime	DR	Displays the current runtime state of each running instance.
Switch backup	SB	Transitions the primary instance to the backup role. The backup instance becomes the primary instance.

Note: To exit from the *takeover* instance or to switch run modes, you can use the *takecmd* commands such as EP, EB, or SB. These commands are useful in scripted input files. In addition, you can exit from a running instance of *takeover* by closing a window or by entering **Ctrl+C**.

Main worker threads

The following sections describe the three main worker threads in *takeover* and how to use *takecmd* to operate them:

- DTM worker thread
- OAM worker thread
- Call control and voice worker thread

These worker threads operate independently. You can issue commands that cause functions to run concurrently, for example, you can request trunk status at the same time that a file is being played.

DTM worker thread

This thread uses the DTM service to monitor trunks. It maintains a linked list of board and trunk numbers to monitor for *takecmd.* After receiving DTMEVN_TRUNK_STATUS, the *takeover* primary instance displays the trunk status information. The backup instance ignores the event. The thread also sets the DTM_START_PARMS parameter field reportmask to 0x7F that reports any change in trunk status at one second intervals.

This thread shows how an application attached to a shared context can be automatically attached to a service object, in this case, a monitored trunk, after receiving a service event. The availability of this automatic attachment is dependent upon the interface provided by the service. In the DTM service, DTMEVN_MONITOR_STARTED provides all applications attached to the context with a unique handle to the monitored trunk object. This handle is contained in the event ***objHd*** field.

The backup instance can immediately use the object handle from this event to perform **dtmGetTrunkStatus**. From this call, it learns the board and trunk number associated with the handle passed in DTMEVN_MONITOR_STARTED. The backup instance then adds an entry to its own list of trunks being monitored.

Whenever the primary instance fails or transitions to the backup role, the backup instance then takes over the role as the primary instance and reports status information for the list of trunks on its list.

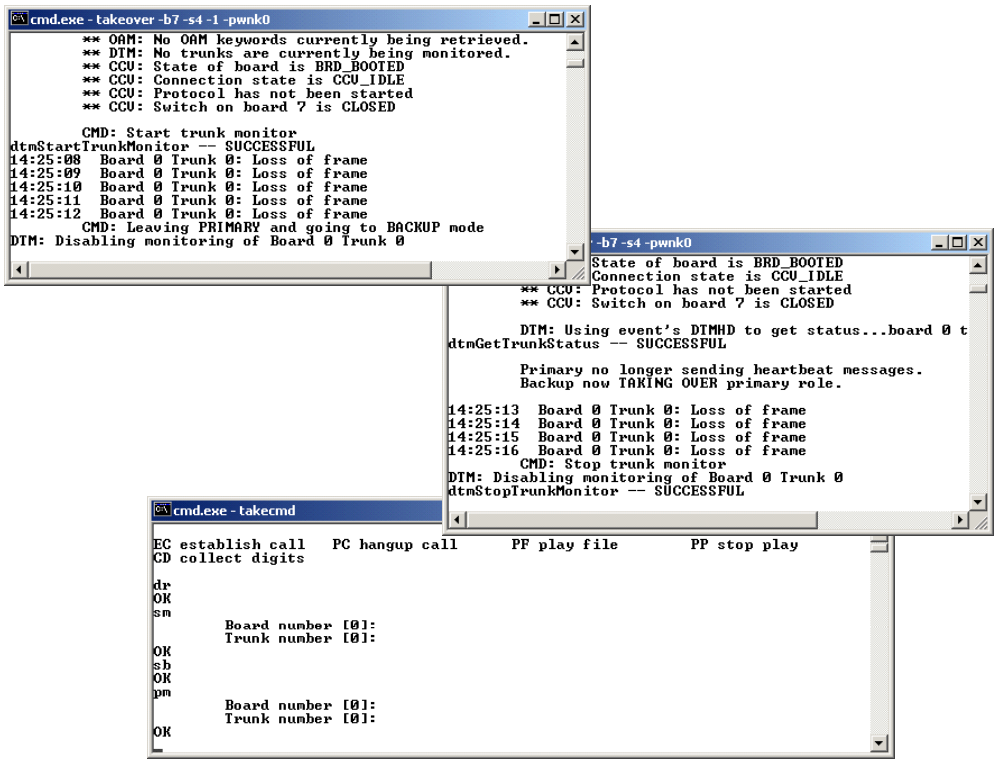
Whenever a backup instance is started, it requests current state information from the primary instance. The DTM thread sends all board-trunk values its list of trunks being monitored. The backup instance then performs **dtmAttachTrunkMonitor** for each entry that it receives and creates its own list entry.

In the following table, the command column lists the two-letter commands to enter at the *takecmd* prompt. Some commands will prompt for additional information, for example, a board number or trunk number.

Use the following *takecmd* commands to operate this thread:

Function	Command	Action
Start monitor	SM Board number Trunk number	The primary instance calls **dtmStartTrunkMonitor**. The backup receives DTMEVN_MONITOR_STARTED and extracts the service object handle from the event ***objhd*** field. It then calls **dtmGetTrunkStatus** to determine which board and trunk are being monitored. It adds the new trunk to the list of monitored trunks.
Stop monitor	PM Board number Trunk number	The primary instance calls **dtmStopTrunkMonitor**.
Get status	GS Board number Trunk number	Both the primary and backup instances call **dtmGetTrunkStatus**. It displays the current trunk status.

The following illustration provides an example of using the DTM service:



OAM worker thread

This thread uses the OAM service to start and stop boards, as well as to periodically retrieve and display OAM keywords. Currently, the OAM service does not support shared service objects. However, the OAM service can be opened on a shared context facilitating the shared responsibility of managing boards within the system.

This thread illustrates how the OAM service aids in systematically shutting down and starting up a board. It also shows how the clock manager event, CLKEVN_ CONFIGURED, indicates when applications can use a board once the clocks are successfully configured.

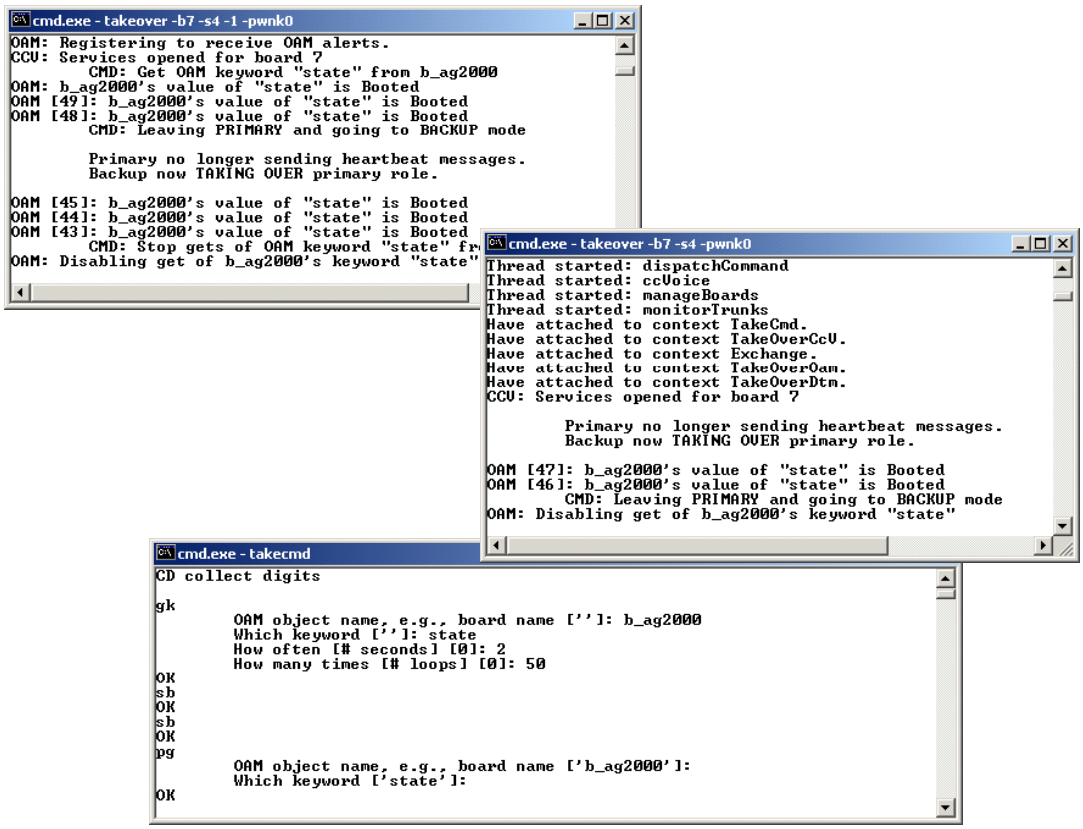
In addition, the thread implements logic to periodically retrieve an OAM object keyword. This implementation demonstrates an application's specific need to exchange checkpoint date independent of the Natural Access services being used. In this case, the primary instance informs the backup instance that it will execute **oamGetKeyword *n*** times and allow ***m*** seconds to elapse between calls. Each time **oamGetKeyword** is called, the primary instance informs the backup instance. If the backup instance needs to take over the periodic keyword retrievals, it has sufficient information to complete the task.

In the following table, the Command column lists the two-letter commands to enter at the *takecmd* prompt. Some commands prompt for additional information, for example, a board number or trunk number.

Use the following *takecmd* commands to operate this thread:

Function	Command	Action
Boot boards	BB Board name	The primary instance calls **oamStartBoard**. The backup instance receives OAMEVN_STARTBOARD_DONE and displays the event information.
Stop boards	PB Board name	The primary instance calls **oamStopBoard**. Before making the call, the OAM thread alerts the other worker threads that the board is about to be stopped and gives them time to make appropriate preparations. The backup instance receives OAMEVN_STOPBOARD_DONE and display the event information.
Get keyword	GK OAM object name Which Keyword How Often How many times	The primary instance establishes a loop to read an OAM object keyword based on the command prompts for an object name, keyword name, how often to read the keyword, and how many times to read it. The primary instance notifies the backup instance that the loop is started, and the backup instance sets up an internal record. Each time the primary instance decrements the loop count, the backup instance is notified to make a similar change. When the loop count reaches 0, the entry for the OAM service keyword is removed from both the primary and backup instances.
Stop get	PG OAM object name Which Keyword	The primary instance stops the loop that is reading OAM keywords and notifies the backup instance that the loop has been stopped.

The following illustration provides an example of using the OAM service:



Call control and voice worker thread

This thread uses the Natural Call Control (NCC), ADI, and Switching (SWI) services to illustrate how to use object descriptors to attach to existing service objects. This thread uses object descriptors to attach to a call that is in the connected state and to attach to an open switch handle (SWIHD).

This thread shows how a second application can hand off and control a connected call. In addition, it shows that if a file is being played on the call connection, a second application can continue the play functionality if the primary application exits. To set up calls, *takeover* uses logic similar to the logic the *incta* program uses. Refer to *Inbound call control: incta* on page 204 for more information.

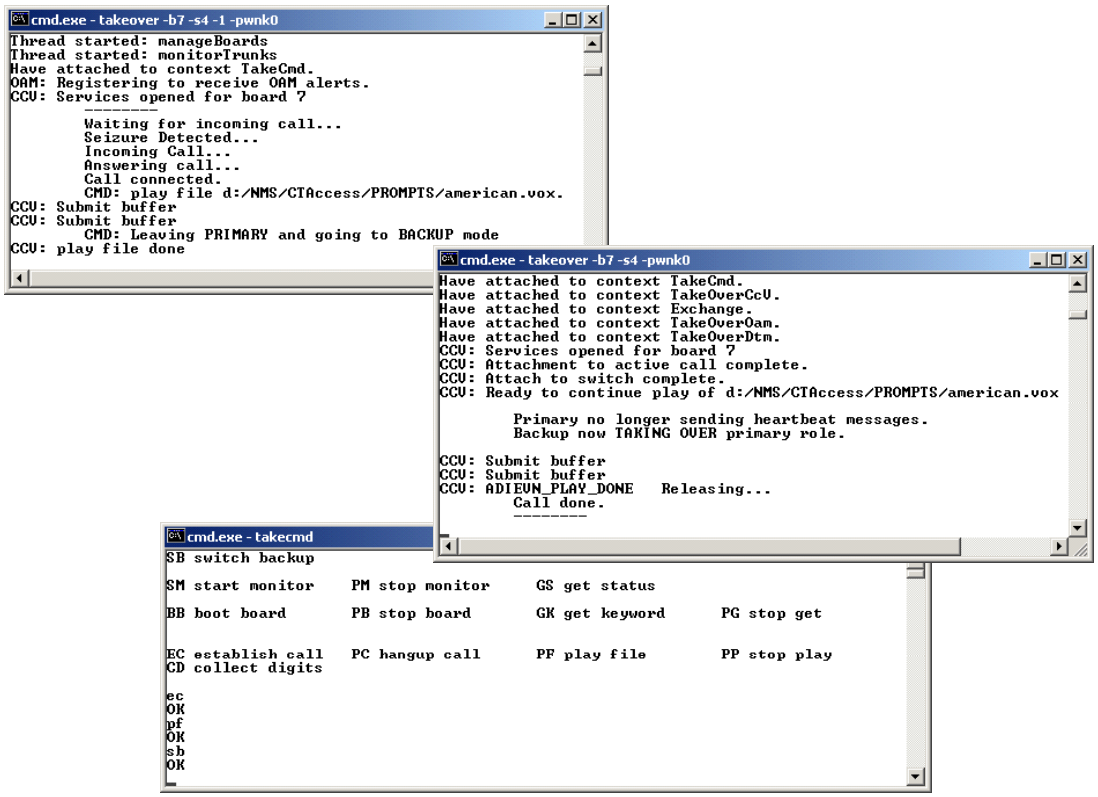
Whenever a backup instance is started, it requests current state information from the primary instance. The primary instance sends object descriptors for the active call and switch handle. In addition, if a file is playing, current position information is transferred to the backup instance. The position information enables the backup instance to continue play operations even though it was inactive at the time the play operation was started.

In the following table, the Command column lists the two-letter commands to enter at the *takecmd* prompt. Some commands will prompt for additional information, for example, number of digits.

Use the following *takecmd* commands to operate this thread:

Function	Command	Action
Establish call	EC	The primary instance establishes a call with switch connections based on the -s and -r options. It then waits for an incoming call. When a call is placed on the connection and the call reaches the connected state, the object descriptors for both the call handle and the switch handle are sent to the backup instance. The backup instance uses **ctaAttachObject** to attach to both the call handle and the switch handle. Other internal data is updated to reflect the connected call.
Hangup call	PC	The primary instance hangs up a call. Once the call is disconnected, the switch outputs are disabled and the switch handle is closed. When the backup instance receives the disconnected event, it releases the call handle and detaches from the switch. If a play file is active, all of the associated resources are released.
Play file	PF	The primary instance plays a voice file on the call connection. *takeover* finds and opens the file, reads in a portion of the file based on encoding information, and uses **adiPlayAsync** to submit and play the first buffer. Subsequent buffers are submitted with **adiSubmitPlayBuffer** when ADIEVN_PLAY_BUFFER_REQ is received. The primary instance notifies the backup instance that a play file has begun. The backup instance immediately opens the file and continues to maintain the current position in the file based on checkpoint updates from the primary instance. The backup instance is always ready to continue the play if the primary instance fails.
Stop play	PP	The primary instance stops playing the file. The primary instance notifies the backup instance to release the resources associated with the aborted play file.
Collect digits	CD Number of digits	The primary instance collects digits in a buffer. The primary instance notifies the backup instance that digit collection has begun. The backup instance prepares to collect the digits. When ADIEVN_COLLECTION_DONE is received, both the primary and backup instances display the buffer of digits collected and release resources.

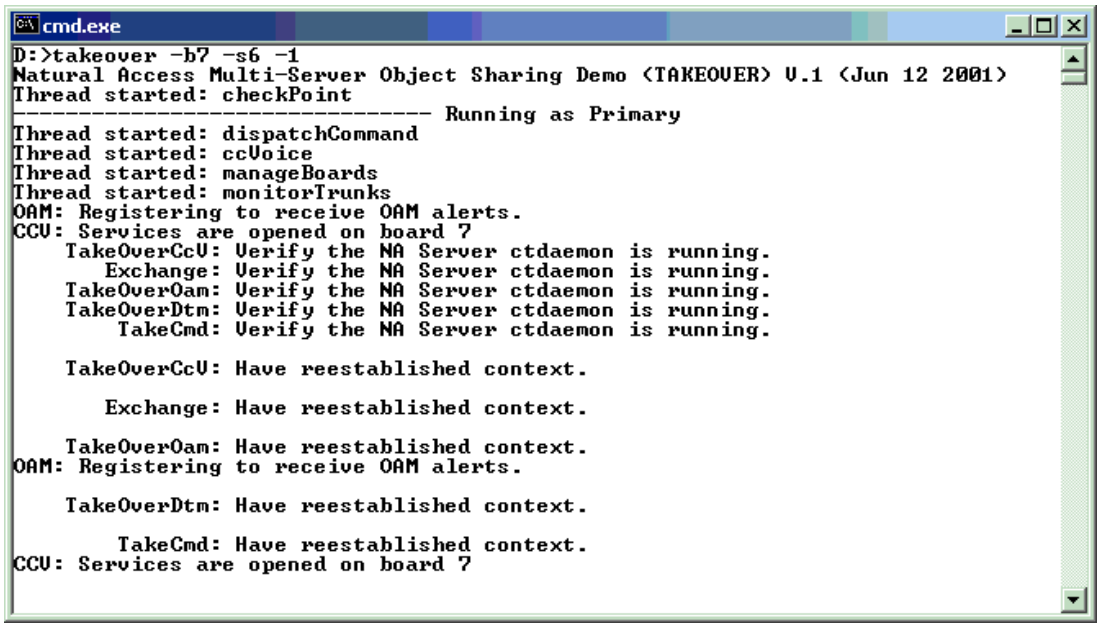
The following illustration shows using the Natural Call Control and Voice services:



Server recovery

If a Natural Access Server (*ctdaemon*) stops running, *takeover* does not fail. *takeover* periodically checks to see if the server is running again. When it determines that Natural Access Server (*ctdaemon*) is active, *takeover* reestablishes the contexts associated with the server.

The following illustration shows recovering from a server loss:



Natural Access Server (*ctdaemon*) does not have to be active at the time *takeover* is started. *takeover* automatically determines its presence and establishes the contexts.

If the server associated with the *takecmd* context is not running, *takeover* is unable to receive commands from *takecmd*.

Specifying server type: testdm

Demonstrates a Natural Access application that specifies the server type on which a context is created and processed. Applications can specify that context processing occur within the application (inproc), on the local server (localhost), or on a remote server (server name or IP address). Applications using the OAM service must use a context created on a local or remote server.

Usage

```
testdm [options]
```

Where call control ***options*** are:

Call control option...	Use this option to...
-?	Display the Help screen and terminate *testdm*.
-@ ***host_name***	Specify the name or IP address of the host on which to create the context. Default: localhost
-b ***board_#***	Specify the board number. Default: board 0
-p ***protocol***	Specify the name of the TCP. Refer to the *NMS CAS for Natural Call Control Developer's Manual* for a list of available protocols.
-d ***digits***	Specify the number to dial. Default: 555
-m ***inslot***,***outslot***	Specify the port (DSP) addresses for incoming and outgoing calls respectively. Default: 0,2

Where OAM ***options*** are:

OAM option...	Use this option to...
-n ***name***	Specify the name of the managed object.
-k ***keyword***	Specify the keyword name. If unspecified, the program displays all keywords.
-f ***field***	Specify the desired field within keyword name ***keyword***.
-i ***index***	Specify an array index of keywords. Default: all
-q	Display the keyword qualifiers.
-s ***text***	Display all keyword names containing the specified ***text***.

Hardware requirements

testdm works with any NMS board that has a processor and DSPs.

Note: Some TCPs supported by *testdm* do not work with some boards.

Software requirements

To use *testdm*:

- Natural Access, NMS OAM, and a protocol must be installed on your system. Refer to the *NMS CAS for Natural Call Control Developer's Manual* for a list of available protocols.
- Natural Access Server (*ctdaemon*) must be running either on a local or remote server. Use the -@ option to specify the name or IP address of a remote server. The test program uses the local server as the default.

Description

testdm displays and sets keyword information for NMS OAM managed objects. The program starts threads that act as inbound and outbound parties. The program does the following:

- Places a call.
- Answers the call.
- Plays a voice message.
- Hangs up the line.

NMS OAM functions are performed on the local or remote server. Call control functions are performed within the processing space of the application (inproc).

Procedure

The following procedure assumes you are testing on an AG 2000 board that has two lines connected through a central office simulator.

Complete the following steps to run *testdm*:

Step	Action
1	Navigate to the directory containing *testdm*.
2	Start *ctdaemon*.
3	Start *testdm* by entering the following command at the prompt: `testdm -b n -p lps0 -d 555 -m slot1,slot2`
4	If the test is successful, *testdm* prints Test passed.

10 Utilities

Summary of the utilities

Natural Access provides the following utilities:

Utility	Description
ctaparm	Displays system parameter defaults if using the Natural Access Server.
ctavers	Verifies Natural Access installation and displays the version information of Natural Access and its components.
ctdaemon	Modifies system global default parameters, sets the global trace mask, performs remote tracing with a Java applet, and optionally logs trace messages to a file. Creates and manages processing contexts on behalf of the applications and enables applications to share and hand off server resources.
nmsabout	Displays names and versions of NMS products installed in the NMS root directory.
nmssearch	Displays names and versions of NMS products installed in a user-specified root directory.
pfencode	Encodes Parameter Management service text parameter files into binary (encoded) parameter files.
rtrace	Monitors trace messages from either a local or a remote Natural Access Server (*ctdaemon*).
setctatrace	Sets the global trace mask of *ctdaemon* while it is active.

Natural Access parameter display: ctaparm

Use this utility to view selected parameter definitions and values stored in shared memory. Refer to *Managing parameters* on page 51 for more information.

Usage

```
ctaparm [options]
```

where ***options*** are:

Option	Use this option to...
-e	Display the names of all parameter structures.
-s ***structnames***	Display all parameters in the listed structures.
-f ***fieldnames***	Display all parameters matching the listed field names.
-g	Generate a parameter file from the current values.
-l ***cfgfile***	Specify the configuration file defining services to initialize.

Description

If *ctdaemon* is running, *ctaparm* reads parameters stored in system shared memory and displays them on the screen. If *ctdaemon* is not running, *ctaparm* displays the compile-time defaults. Use the command line to specify the parameters to retrieve.

Procedure

To run *ctaparm,* enter the following command at the prompt:

```
ctaparm [-e -s structlist -f fieldlist -g]
```

The lists generated by the -s and -f options are delimited by the + (plus) character. For example:

```
ctaparm -s adi.play+adi.record
ctaparm -f gain+dtmfabort
```

Note: The error CTAERR_ALREADY_INITIALIZED indicates that *ctaparm* is using a different configuration file than *ctdaemon*. The file used by *ctaparm* lists services in the [ctasys] section that are not loaded by *ctdaemon*. Both *ctaparm* and *ctdaemon* load *\nms\ctaccess\cfg\cta.cfg* by default. (For UNIX, the path is */opt/nms/ctaccess/cfg/cta.cfg*.)

Service version confirmation: ctavers

Use this utility to view version information for services and service managers that *ctdaemon* uses and to confirm that you are running the correct versions. Refer to *Verifying compatibility* on page 57 for more information.

Usage

```
ctavers [options]
```

where ***options*** are:

Option	Use this option to...
-s	Report the versions of services and service managers *ctdaemon* uses. If *ctdaemon* is not running, *ctavers* runs in library mode and reports the versions of the services and service managers contained within the default Natural Access configuration file.
-x ***xxx***,***xxxmgr***	Report the version of the specified service (***xxx***) and service manager (***xxxmgr***)
-l ***cfgfile***	Report the versions of services and service managers contained within the Natural Access configuration file (***cfgfile***).

Description

ctavers calls **ctaInitialize** using the services and service managers specified in *cta.cfg*. It retrieves and displays the version information for all services and service managers registered during Natural Access initialization. The version information for the Natural Access dispatcher (*cta.dll*) also displays.

Procedure

Ensure that CTA_DPATH includes the directory where *cta.cfg* is located. (The default *cta.cfg* is located in the *\nms\ctaccess\cfg* directory.)

To run *ctavers,* enter the following command at the prompt:

```
ctavers
```

Natural Access Server: ctdaemon

This topic describes:

- Using *ctdaemon*
- Using system global default parameters
- Using tracing
- *ctdaemon* output

Using ctdaemon

Use *ctdaemon* to:

- Modify system global default parameters, set the global trace mask, perform remote tracing with a Java applet, and optionally log trace messages to a file.
- Create and manage processing contexts on behalf of the applications and enable applications to share and hand off server resources.

Usage

```
ctdaemon [option]
```

Where Windows ***options*** are:

Windows option	Use this option to...
-f ***filename***	Specify an ASCII file containing statements for defining services and service managers and overriding default parameters. Usually this file is *cta.cfg*.
-i	Run *ctdaemon* as an interactive console application.
-? or -h	Access help.

Where UNIX ***options*** are:

UNIX option	Use this option to...
-f ***filename***	Specify the ASCII configuration file containing statements for defining services and service managers and overriding default parameters. Usually this file is *cta.cfg*.
-i	Run *ctdaemon* as an interactive console application.
-q	Run *ctdaemon* in quiet mode (as a non-interactive console application). This mode is the default if no option is specified.
-s	Stop the currently running, non-interactive *ctdaemon* session.
-k	Deallocate global objects created by previous executions of *ctdaemon*.
-? or -h	Access help.

Description

ctdaemon performs the following actions:

Step	Action
1	Reads the configuration file specified on the command line or finds *cta.cfg* in CTA_DPATH (defined during installation).
2	Passes the services and service managers specified in the [ctasys] section of the *cta.cfg* to the call to **ctaInitialize**.
3	Places the compile-time parameter values in shared memory to be modified after service initialization.
4	Reads the [ctapar] section of the configuration file and writes new parameter values into shared memory.

Procedure

Under Windows, when you reboot your system following the Natural Access installation, the Natural Access Server (*ctdaemon*) starts automatically as a Windows service. If the service is not running, you can start it in one of the following ways:

- Access a command prompt and enter:

```
net start ctdaemon
```

- In the Windows Control Panel, double-click on **Services** and start the NMS CT daemon service.

Under UNIX, invoke ctdaemon -i from the command prompt. This method allows full console interaction with *ctdaemon*.

Using system global default parameters

Applications that use system global default parameters retrieve default parameter values from shared memory instead of retrieving compile-time process global defaults from the process data area.

To use system global default parameters, an application must:

- Specify in *cta.cfg* all Natural Access services used by the application. For example, *cta.cfg* must be modified as shown to allow a fax application to use the system global default parameters:

```
Service = nfx, nfxmgr
```

- Set the parmflags field of the CTA_INIT_PARMS structure passed to **ctaInitialize** to CTA_PARM_MGMT_SHARED. If this bit is chosen and *ctdaemon* is not running, CTAERR_SHAREMEM_ACCESS is returned. Since an error is not returned if *ctdaemon* crashes, parmflags cannot be used to check *ctdaemon.*

  If an application wants global default parameters to be stored in shared system memory but does not want an error to be returned if *ctdaemon* is not running, set parmflags to CTA_PARM_MGMT_SHARED_IF_PRESENT. If this setting is chosen and *ctdaemon* is not running, the default parameter management is local to the process.

When you start *ctdaemon* in a command window, you can use it to retrieve and modify global default parameter values. For example:

- Enter s ***parmname*** = ***value*** or g ***parmname*** to modify (set) or retrieve (get) the current default value.
- Enter ? at the interactive command prompt to display the available commands.

To generate a list of parameters that can be used in the [ctapar] section of the configuration file, use the *ctaparm* utility. Refer to *Natural Access parameter display: ctaparm* on page 236. Edit the parameters that have different default values and place them in *cta.cfg*.

Using tracing

ctdaemon holds a shared memory segment for tracing. To enable tracing from applications, *ctdaemon* must be running and the traceflags field of the CTA_INIT_PARMS structure passed to **ctaInitialize** must be set to CTA_TRACE_ENABLE.

Trace messages display only if the trace mask corresponding to the specific type of message is enabled. To instruct *ctdaemon* to set the global trace mask, use the m ***tracemask*** command from the *ctdaemon* command line. This command sets the trace mask for all processes that enabled tracing with **ctaInitialize**. To be more selective in tracing, use **ctaSetTraceLevel** from the application itself to set the trace mask for a specific context in a specific process.

Specify the trace mask as a sequence of strings delimited by + signs (with no spaces). Each string corresponds to a trace mask value that you can specify in **ctaSetTraceLevel**.

Note: On UNIX systems, Natural Access Server (*ctdaemon*) must be running to provide board resources for Natural Access applications. If *ctdaemon* stops running, stop any Natural Access applications that are performing tracing or using global default parameters before restarting *ctdaemon.*

Enter d at the interactive command prompt to see recognized names for setting the trace mask:

String	Trace mask value
all	CTA_TRACEMASK_ALL
allcmd	CTA_TRACEMASK_ALL_COMMANDS
allevt	CTA_TRACEMASK_ALL_EVENTS
drvcmd	CTA_TRACEMASK_DRIVER_COMMANDS
drvevt	CTA_TRACEMASK_DRIVER_EVENTS
dispcmd	CTA_TRACEMASK_DISP_COMMANDS
dispevt	CTA_TRACEMASK_DISP_EVENTS
apicmd	CTA_TRACEMASK_API_COMMANDS
apievt	CTA_TRACEMASK_API_EVENTS
apierr	CTA_TRACEMASK_API_ERRORS
svcerr	CTA_TRACEMASK_SVC_ERRORS
dbgbit0	CTA_TRACEMASK_DEBUG_BIT0
dbgbit1	CTA_TRACEMASK_DEBUG_BIT1
dbgbit2	CTA_TRACEMASK_DEBUG_BIT2
dbgbit3	CTA_TRACEMASK_DEBUG_BIT3
cltcmd	CTA_TRACEMASK_CLNT_COMMANDS
cltevt	CTA_TRACEMASK_CLNT_EVENTS
svrcmd	CTA_TRACEMASK_SRVR_COMMANDS
svrevt	CTA_TRACEMASK_SRVR_EVENTS
none	0

To view the trace messages that are made available by *ctdaemon*:

- Open the trace file. The TraceFile keyword in *cta.cfg* specifies the name of the trace file. Trace messages are appended to the trace file each time *ctdaemon* is started. You must delete or archive this file as necessary.
- Use the local console.

  Note: When running the Natural Access Server, view trace messages from the trace file.
- From a remote location, use the remote trace utility and a Java 1.1-capable web browser. Refer to *Remote trace utility: rtrace* on page 251 for details about using the remote trace utility.

  Refer to *Managing parameters* on page 51 and *Handling Natural Access errors* on page 55 for more information.

ctdaemon output

ctdaemon generates the following output:

- Trace timestamp.
- Generic trace information (valid for all trace messages).
- Type-specific trace information.

Note: All values are in hexadecimal format unless otherwise noted.

The timestamp displays as follows:

```
Wed Dec 24 21:40:01 2003
```

The timestamp indicates when the event was logged.

Generic trace information

The generic trace information displays as follows:

```
     | pid=ce tid=ba ctahd=80000001 (CTATEST) uid=0 tag=4005 sev=0
```

Generic trace information includes:

Field	Description
pid	Identifier of the process that logged trace messages.
tid	Identifier of the thread in the process that logged the trace message. This information is not valid on non-threaded operating systems.
ctahd	Identifier of the context for which the trace message was logged. (The context ID is an arbitrary string passed to **ctaCreateContext**.)
text in parentheses	Name of the context for which the trace message was logged.
uid	User ID specified when creating the context for which the trace message was logged.
tag	Numeric value of trace type tag. This value determines how information pertaining to this trace message is formatted. The standard trace types are explained later in this section.
sev	Severity of the error. Refer to CTA_TRACE_SEVERITY in *ctadef.h*. Valid values include:

Value	Description
0	CTA_TRACE_SEVERITY_INFO
1	CTA_TRACE_SEVERITY_WARNING
2	CTA_TRACE_SEVERITY_ERROR

Type-specific trace information

Type-specific trace information pertains to the following types of objects that are being traced:

- Commands
- Events
- Errors
- Strings
- DWORDs
- 32-bit integers (INT32)

Command information

Command information displays as follows:

```
|
↓ ADICMD_START_PROTOCOL (=13001) src=2000 dst=1 s1=60 d1=12f8c8 s2=e0
d2=0 s3=78002907 d3=0
```

Command information includes:

Field	Description
↓	Displays under Windows only.
Name	Name of the command sent to the dispatcher.
Number in parentheses	Numeric value of the command.
src	Source service identifier. Service identifier values are defined in their respective service header files (for example, *adidef.h* or *ctadef.h*).
dst	Destination service identifier. In this example, the destination of 1 corresponds to ADI_SVCID. Service identifier values are defined in their respective service header files (for example, *adidef.h* or *ctadef.h*).
s1	Size of the first buffer associated with this command.
d1	Address of the first buffer associated with this command.
s2 and d2	Size and address of the second buffer associated with the command.
s3 and d3	Size and address of the third buffer associated with the command.

Event information

You can retrieve information about events generated by the dispatcher and events generated by the application. The event information displays as follows:

```
|
↑ CTAEVN_DISP_OPEN_SERVICE_DONE (10002111) Finished  (val=1001) src=1
dst=1000 time=df8ee5ce uid=0 size=7c buf=421a90
```

Event information includes:

Field	Description
↑	Displays under Windows only.
Name	Name of the dispatcher event.
Number in parentheses	Event ID value.
String	Text associated with the reason code for this event.
val	Value of the dispatcher event.
src	Source identifier of the service that caused this event to be sent to the dispatcher.
dst	Destination service identifier to which this event was sent.
time	Timestamp that the dispatcher associates with the event. Compare timestamps with other events to determine the order in which events were generated.
uid	User identifier (0) of the context that caused this event to be generated (**ctaCreateContext**).
size	Size of the event-specific buffer associated with this event.
buf	Address of the event-specific buffer associated with this event.

Error information

You can retrieve error information from *ctdaemon*. Errors can appear as:

```
| ERR: CTAERR_INVALID_STATE (=c) d1=0 d2=0 txt=  fn=adiStartProtocol ln=0 file=
```

Error information includes:

Field	Description
Name	Name of the error.
Code in parentheses	Value of the error code.
d1 and d2	Error-specific buffers associated with the error.
fn	Name of the function that generated this error.
ln	Line number in the source file for the function that generated the error. If ln=0, then the function that generated the error did not specify a line.
file	Name of the source file for the function that generated the error. If file= is blank, the function that generated the error did not identify its source file.

Strings

If you explicitly call **ctaLogTrace** with a tracetag value of CTA_TRACETAG_STRING, the string displays as:

```
| debug_string
```

DWORD

If you explicitly call **ctaLogTrace** with a tracetag value of CTA_TRACETAG_DWORD, the value displays as:

```
| DWORD: 10 (0xa)
```

The first number is the decimal value of the DWORD. The value in parentheses is the hexadecimal value of the DWORD.

32-bit integers (INT32)

If you explicitly call **ctaLogTrace** with tracetag value of CTA_TRACETAG_INT32, the value displays as:

```
| INT32: 10 (0xa)
```

The first number is the decimal value of the 32-bit integer. The value in parentheses is the hexadecimal value of the DWORD.

Installed product and version display: nmsabout

Use this utility to view names and version information for NMS products installed in the NMS root directory, and for Windows installations, in the ***system_root****system32* directory.

Usage

```
nmsabout [options]
```

where ***options*** are:

Option	Use this option to...
-h	Access help.
product_name	Display the NMS product name and base version number followed by version information for each installed product file. If a file was not installed during the original installation, the utility displays the origin of the file (either patch or service pack).
-f ***filename***	Display the NMS product name and version number of each file that matches the file name specified. For the utility to locate the file, the file must be in the same directory in which it was originally installed. **Note:** When entering a file name, enter just the name of the file, not a complete path.
-x ***product***	Display the NMS product name and base version number followed by version information on all existing product files that were not installed during the original installation. If a file is associated with a patch or service pack that you installed, the utility displays the patch or service pack number. It also displays the number of files that have changed since the original installation, as well as the total number of installed files.

Description

Each Natural Access product contains a signature file (*.sgn*) that lists each file associated with the product. A signature of checksum information and a full path uniquely identify each associated file.

nmsabout uses the signature file to display the name and version information of product files that are on your system. For example, *ctaccess.sgn* lists 100 files. To access information for the files on your system, enter the following command at the prompt:

```
nmsabout ctaccess
```

nmsabout displays information for the 75 files on your system.

If you request information on a product file that does not match any file listed in the signature file, you receive one of the following error messages:

Error message	Meaning
unknown	The file for which you requested information is located on your system. Although the path of the file matches the path in the signature file, the checksum information does not match.
not NMS file	The file for which you requested information exists on your system, but it is not listed in any signature file. Ensure that the file is in its original installation directory.
file not found	The file for which you requested information does not exist on your system.

Procedure

To run *nmsabout*, enter the following command at the prompt:

```
nmsabout [option]
```

For example, if you enter the following command:

```
nmsabout  -x  ctaccess
```

nmsabout lists the ctaccess product name and base version number followed by version information on all existing ctaccess files that were not installed during the original installation. If a file is associated with a patch or service pack that you installed, the utility displays the patch or service pack number. It also displays the number of files that have changed since the original installation, as well as the total number of installed files.

Note: For UNIX systems, run this utility as super user.

User-specified product directory search: nmssearch

Use this utility to search a user-specified root directory and display a list of installed NMS products.

Usage

```
nmssearch [options]
```

where ***options*** are:

Option	Use this option to...
-h	Access Help.
directory	Display the NMS product name and version number for each associated file in the specified root directory.
-f ***filename directory***	Display the NMS product name and version number for each file that matches the specified file name and root directory. For the utility to locate the file, the file must be in the same directory in which it was originally installed. **Note:** When entering a file name, enter just the name of the file, not a complete path.
-p ***product directory***	Display the NMS product name and base version number followed by version information on all existing product files installed in the specified root directory. To exclude non-NMS files from the display, enter the following command line: `nmssearch [options] | grep -v "not NMS file"`

Description

Each Natural Access product contains a signature file (*.sgn*) that lists each file associated with the product. A signature of checksum information and a full path uniquely identify each associated file.

nmssearch uses the signature file to display the name and version information of product files that are on your system. For example, *ctaccess.sgn* lists 100 files. To access information for the files on your system, enter the following command at the prompt:

```
nmssearch -p ctaccess \nms\bin
```

nmssearch displays information for the 75 files on your system in the Windows *\nms\bin* root directory.

If you request information on a product file that does not match any file listed in the signature file, you receive one of the following error messages:

Error message	Meaning
unknown	The file for which you requested information is located on your system. Although the path of the file matches the path in the signature file, the checksum information does not match.
not NMS file	The file for which you requested information exists on your system, but it is not listed in any signature file. Ensure that the file is in its original installation directory.
file not found	The file for which you requested information does not exist on your system.

Procedure

To run *nmssearch*, enter the following command at the prompt:

```
nmssearch [option]
```

For example, if you enter the following command with a UNIX root directory name of */opt/nms/bin*:

```
nmssearch /opt/nms/bin
```

nmssearch lists the NMS product names and version numbers for each of the associated files in the */opt/nms/bin* root directory.

Note: For UNIX systems, run this utility as super user.

PRM service text parameter file encoder: pfencode

Use this utility to encode a Parameter Management service text parameter file to a binary parameter file. The PRM service can load either the text parameter file or an encoded binary parameter file. Encoded binary parameter files provide an extra level of security to discourage end user modification. Use the encoded files to comply with international regulations.

Usage

```
pfencode [option] filename
```

where ***option*** is:

Option	Use this option to...
-o ***filename***	Specify the output (encoded) file name. The default is the specified text parameter file name with a *.pf* extension. If the output (encoded) name conflicts with supplied text parameter file name, the original text parameter file extension is changed to *.tpf*.

Description

pfencode encodes your parameter file to discourage modification by end users. Both text and binary (encoded) parameter files can have a *.pf* extension. Refer to *Defining and using application-specific parameters* on page 265 for more information.

Procedure

The following two examples demonstrate using *pfencode* on a text parameter file *prmMyFile.pf*.

Parameter files (text or encoded binary) loaded by the PRM service must contain the *prm* prefix and the *.pf* extension.

Note: Save the original text parameter file, since *pfencode* cannot decode an encoded parameter file to reproduce the original text parameter file.

Example 1

Enter the following command at the prompt:

```
pfencode prmMyFile.pf
```

pfencode renames the original text parameter file *prmMyFile.tpf* and generates an encoded binary file *prmMyFile.pf*.

Example 2

Enter the following command at the prompt:

```
pfencode -o prmData.pf prmMyFile.pf
```

pfencode leaves the original text parameter file unchanged, and generates an encoded binary file *prmData.pf* containing the identical parameter information specified in *prmMyFile.pf*.

Remote trace utility: rtrace

Use this utility to monitor trace messages from local or remote instances of Natural Access Server (*ctdaemon*), set a global trace mask, and log trace messages to a file.

Note: The rtrace utility is available under Linux and Solaris only.

Usage

```
rtrace [options]
```

where ***options*** are the following values:

Option	Use this option to...
-? or -h	Access Help and terminate *rtrace*.
-@ ***host_name***	Specify the name or IP address of the server to monitor. If unspecified, the default is localhost.
-f ***filename***	Specify the name of a file to maintain a log of trace messages.

Description

When *rtrace* starts up, it connects to the specified server and waits for trace messages. All of the received messages are displayed and optionally logged to the specified file. Press **Q** or **Esc** to exit from *rtrace*.

To run *rtrace*, Natural Access Server (*ctdaemon*) must be running. If Natural Access Server (*ctdaemon*) disconnects from *rtrace* because of a system restart or failure, *rtrace* waits for Natural Access Server (*ctdaemon*) to restore the connection.

Procedure

To run *rtrace*:

Enter the following command at the prompt:

```
rtrace [-@ host_name] [-f filename]
```

For example, if you enter:

```
rtrace -@ localhost -f trace.log
```

rtrace connects to localhost and begins logging messages to *trace.log*.

Set global trace mask of ctdaemon utility: setctatrace

Use this utility to set the global trace mask of *ctdaemon* while it is active.

Usage

```
setctatrace [options]
```

where ***options*** are the following values:

Option	Use this option to...
-? or -h	Access Help and terminate *setctatrace*.
-s ***host_name***	Specify the name or IP address of the server to monitor. If unspecified, the default is localhost.
-m ***mask***	Specify the global trace mask for *ctadaemon*.

where ***mask*** values are:

String	Constant	Hex value
all	CTA_TRACEMASK_ALL	0xff7f
allcmd	CTA_TRACEMASK_ALL_COMMANDS	0x5015
allevt	CTA_TRACEMASK_ALL_EVENTS	0xa02a
drvcmd	CTA_TRACEMASK_DRIVER_COMMANDS	0x0001
drvevt	CTA_TRACEMASK_DRIVER_EVENTS	0x0002
dispcmd	CTA_TRACEMASK_DISP_COMMANDS	0x0004
dispevt	CTA_TRACEMASK_DISP_EVENTS	0x0008
apicmd	CTA_TRACEMASK_API_COMMANDS	0x0010
apievt	CTA_TRACEMASK_API_EVENTS	0x0020
apierr	CTA_TRACEMASK_API_ERRORS	0x0040
svcerr	CTA_TRACEMASK_SVC_ERRORS	0x0080
dbgbit0	CTA_TRACEMASK_DEBUG_BIT0	0x0100
dbgbit1	CTA_TRACEMASK_DEBUG_BIT1	0x0200
dbgbit2	CTA_TRACEMASK_DEBUG_BIT2	0x0400
dbgbit3	CTA_TRACEMASK_DEBUG_BIT3	0x0800
cltcmd	CTA_TRACEMASK_CLNT_COMMANDS	0x1000
cltevt	CTA_TRACEMASK_CLNT_EVENTS	0x2000
svrcmd	CTA_TRACEMASK_SRVR_COMMANDS	0x4000
svrevt	CTA_TRACEMASK_SRVR_EVENTS	0x8000
none	CTA_TRACEMASK_NONE	0x0000

Description

When you use *setctatrace* to set the global trace mask of *ctadaemon*, you do not have to stop *ctdaemon*. *setctatrace* sets the global trace mask of *ctdaemon* dynamically, while *ctdaemon* is active.

Procedure

To run *setctatrace*, enter the following command at the prompt:

```
setctatrace [-s host_name] [-m mask] [-h]
```

Specify the trace mask as a sequence of strings, constants, decimal values, or hex values delimited by + signs (with no spaces). To specify a hex value, prefix the value with 0x. Enter -? or -h at the interactive command prompt to see recognized names for setting the trace mask.

Trace messages display only if the trace mask corresponding to the specific type of message is enabled. *setctatrace* sets the trace mask for all processes that enabled tracing with **ctaInitialize**. To be more selective in tracing, use **ctaSetTraceLevel** from the application itself to set the trace mask for a service on a specific context.

For example, if you enter:

```
setctatrace -m apievt+CTA_TRACEMASK_API_ERRORS+0x0001
```

setctatrace sets the trace mask for API events, API errors, and driver commands.

Call *setctatrace* -m 0 to disable the global trace mask.

11 Errors and events

Alphabetical error summary

The following table lists the Natural Access errors in alphabetical order. All errors are 32 bit.

Error name	Hex	Decimal	Description
CTAERR_ALREADY_DEFINED	0x10000009	268435465	*Problem*: The service or service manager was already defined. *Solution*: Ensure that there is no redundancy in the services or service manager list specified in **ctaInitialize** and **ctaCreateQueue**.
CTAERR_ALREADY_INITIALIZED	0x1E	30	The Natural Access library or a Natural Access service was already initialized.
CTAERR_BAD_ARGUMENT	0x7	7	*Problem*: A function argument value was invalid or a required pointer argument was NULL. *Solution*: Check all arguments for valid types and ranges.
CTAERR_BAD_NAME	0x42	66	The ***descriptor*** specified with **ctaCreateContext** or **ctaCreateContextEx** is already in use on the Natural Access Server.
CTAERR_BAD_SIZE	0xB	11	*Problem*: A ***size*** argument was too small to receive a data structure, or a play or record buffer was not a multiple of the frame size for the specified encoding. *Solution*: Check all arguments for valid sizes.
CTAERR_BOARD_ERROR	0x3	3	An unexpected error occurred on the board. In most cases, a CTAEVN_BOARD_ERROR event contains the board error code.
CTAERR_CREATE_EVENT_FAILED	0X2A	42	An operating-system-specific event handler returned an error.
CTAERR_CREATE_MUTEX_FAILED	0x1C	28	Failure to create a thread lock.
CTAERR_DISK_FULL	0x29	41	*Problem*: There is not enough room on the disk to complete the write operation. No data was written. *Solution*: Make some room on the disk.

Error name	Hex	Decimal	Description
CTAERR_DRIVER_ERROR	0x18	24	*Problem*: A device driver returned an error code. *Solution*: Check that device drivers are loaded and running.
CTAERR_DRIVER_OPEN_FAILED	0x14	20	The driver open failed because either the driver was not installed or the maximum number of opens was exceeded.
CTAERR_DRIVER_RECEIVE_FAILED	0x16	22	An error occurred when retrieving an event from a driver.
CTAERR_DRIVER_SEND_FAILED	0x17	23	A driver returned an error while sending a message. This error occurs on an AG board if the board is reset.
CTAERR_DRIVER_VERSION	0x15	21	The driver does not support the requested function.
CTAERR_DUPLICATE_CTAWAITOBJ	0x2E	46	An already registered Natural Access wait object was passed as an argument to **ctaRegisterWaitObject**.
CTAERR_DUPLICATE_EXTPARMS	0x10000003	268435458	A duplicate parameter name exists in two services of the same name implemented by different service managers.
CTAERR_ENUM_END	0x43	67	End of list reached and no element was found with the specified index.
CTAERR_FATAL	0x2	2	An internal error occurred in Natural Access.
CTAERR_FILE_ACCESS_DENIED	0x23	35	*Problem*: Tried to open one of the following files: • A file for writing that another application process or thread already opened for writing • A read-only file for writing • A file that has inadequate permissions • A file using a directory name *Solution*: Open the file for read-only, change the file attributes or permissions, or specify a different file name.
CTAERR_FILE_CREATE_FAILED	0x26	38	*Problem*: File creation failed because of a system error. *Solution:* Verify that the path is valid and that the file does not already exist.

Error name	Hex	Decimal	Description
CTAERR_FILE_EXISTS	0x24	36	*Problem*: You attempted to create a file that already exists. *Solution*: Remove or rename the existing file, or specify a different file name.
CTAERR_FILE_NOT_FOUND	0x21	34	*Problem*: The specified file does not exist. *Solution*: Create the file or specify a different file name.
CTAERR_FILE_OPEN_FAILED	0x25	37	*Problem*: File open failed because of a system error. *Solution*: Verify that the file exists.
CTAERR_FILE_READ_FAILED	0x27	39	*Problem*: The file is not open, is locked, or the expected amount of data could not be read. *Solution*: Make sure that the file type is the expected type. Verify that an incorrect handle was not used to close another file.
CTAERR_FILE_WRITE_FAILED	0x28	40	*Problem*: The file is not open, is locked, or the expected amount of data could not be written. *Solution*: Verify that an incorrect handle was not used to close another file.
CTAERR_FUNCTION_ACTIVE	0xF	15	One of the following occurred: • Tried to start an asynchronous function that is already started. • Tried to get a digit or flush the digit queue while collecting digits.
CTAERR_FUNCTION_NOT_ACTIVE	0xE	14	Tried to stop or modify a function that is not running.
CTAERR_FUNCTION_NOT_AVAIL	0xD	13	One of the following applies: • The requested operation is not available with the current protocol. • The necessary *.dsp* or *.tcp* file was not downloaded to the board. • The requested function required a service that is reserved for use by the ADI service.

Error name	Hex	Decimal	Description
CTAERR_INCOMPATIBLE_PARMS	0x10000001	268435457	*Problem*: A discrepancy was detected in the definition of standard parameters of two identical services implemented by different service managers. *Solution*: Do not require both implementations of the same service in your application.
CTAERR_INCOMPATIBLE_REVISION	0x11	17	*Problem*: A service or service manager is incompatible with the Natural Access version being used. *Solution*: Ensure that all versions are compatible.
CTAERR_INCOMPATIBLE_SERVICES	0x00000047	71	On the server on which the context is created, at least one of the available services conflicts with the same service open on another server. An application can use only one instance of a service at a time.
CTAERR_INTERNAL_ERROR	0x2D	45	A Natural Access internal error occurred.
CTAERR_INVALID_ADDRESS	0x10000005	268435461	*Problem*: Natural Access dispatcher detected an invalid command or event address. *Solution*: Ensure that all required services are initialized and open.
CTAERR_INVALID_BOARD	0x1A	26	*Problem*: The specified board number was not successfully configured. *Solution*: Reconfigure the board.
CTAERR_INVALID_CTAHD	0x5	5	An invalid context handle was passed as an argument to a function, or the context was destroyed by another thread.
CTAERR_INVALID_CTAQUEUEHD	0x4	4	An invalid queue handle was passed as an argument to a function, or the queue was destroyed by another thread.
CTAERR_INVALID_HANDLE	0x20	32	An invalid handle was passed as an argument to this function.
CTAERR_INVALID_LIB	0x45	69	When using **ctaInitialize,** the setting for the ctaflags parameter specifies a library that is not supported by the linked Natural Access library. For more information, refer to *Initializing Natural Access applications* on page 35.

Error name	Hex	Decimal	Description
CTAERR_INVALID_SEQUENCE	0x13	19	One of the following occurred: • An attempt to open or close services was made while another request was pending. • An attempt was made to stop a function that is already being stopped. • A call control function was called before receiving a response to the previous call control command. • A play or record buffer was submitted when it was not expected.
CTAERR_INVALID_STATE	0xC	12	This function is not valid in the current state.
CTAERR_LICENSE_EXPIRED	0x30	48	The service license expired.
CTAERR_LOAD_LIB	0x40		An error occurred when loading the Natural Access core library.
CTAERR_LOCK_TIMEOUT	0x1D	29	A thread lock timed out.
CTAERR_NO_LICENSE	0x2F	47	No valid license for the specified service exists, or all available license units are in use.
CTAERR_NOT_FOUND	0xA	10	One of the following occurred: • The specified parameter does not exist. • The service or service manager was not initialized, not attached to the queue, or not opened on the context. • The application requested a service on a local or remote server that is not specified in the server configuration file.
CTAERR_NOT_IMPLEMENTED	0x9	9	The function was not implemented.
CTAERR_NOT_INITIALIZED	0x1F	31	*Problem*: Natural Access was not yet initialized. *Solution*: Call **ctaInitialize**.
CTAERR_OS_INTERNAL_ERROR	0x2C	44	An operating-system-specific error occurred.
CTAERR_OUT_OF_MEMORY	0x6	6	*Problem*: Unable to allocate memory for queue, for driver or context, for play or record buffers, or for temporary storage. *Solution*: When this error occurs on a DONE event, it can mean that there was insufficient memory on the board.

Error name	Hex	Decimal	Description
CTAERR_OUT_OF_RESOURCES	0x8	8	*Problem*: Unable to find free space in internal tables. *Solution*: Verify that handles are closed when no longer used.
CTAERR_OUTPUT_ACTIVE	0x1B	27	*Problem*: Open services failed because the stream and timeslot are already open on another context, or an output function such as play failed because there is another active output function. *Solution*: Check another stream:timeslot or stop the running function.
CTAERR_PATH_NOT_FOUND	0x22	33	*Problem*: The directory portion of a file name is invalid. *Solution*: Create the directory or specify a different path.
CTAERR_RESOURCE_CONFLICT	0x12	18	*Problem*: A needed resource is already being used. *Solution*: Release the resource and try again.
CTAERR_SERVICE_IN_USE	0x10000008	268435464	Attempted to open a service (with **ctaOpenServices**) that is already open.
CTAERR_SERVICE_NOT_AVAILABLE	0x10000006	268435462	*Problem*: Attempted to call a service function on a context that does not have the service opened. *Solution*: Open the service before calling the service function.
CTAERR_SHAREMEM_ACCESS	0x10	16	*Problem*: Failed to access shared memory. *Solution*: Check that *ctdaemon* is running.
CTAERR_SVR_COMM	0x41	65	The server did not respond to the client application request within the required timeout period (by default, three seconds).
CTAERR_TOO_MANY_OPEN_FILES	0x19	25	*Problem*: No more available file handles in the current process. *Solution*: Close some files or increase the file handle limit.
CTAERR_TRACE_NOT_ENABLED	0x1000000A	268435466	*Problem*: CTA_TRACE_ENABLE was not set during **ctaInitialize**. *Solution*: Ensure that CTA_TRACE_ENABLE is set.
CTAERR_WAIT_FAILED	0x10000004	268435460	An operating-system-specific wait function returned an error.
CTAERR_WAIT_PENDING	0x10000003	268435459	**ctaWaitEvent** was already called on the same queue handle.

Error name	Hex	Decimal	Description
CTAERR_WRONG_COMPAT_LEVEL	0x34	52	*Problem*: The compatibility level passed to **ctaInitialize** is different from the current level of Natural Access. *Solution*: Recompile the application.

Numerical error summary

The following table lists the Natural Access errors in numerical order:

Hex	Decimal	Error name
0x1	1	CTAERR_RESERVED
0x2	2	CTAERR_FATAL
0x3	3	CTAERR_BOARD_ERROR
0x4	4	CTAERR_INVALID_CTAQUEUEHD
0x5	5	CTAERR_INVALID_CTAHD
0x6	6	CTAERR_OUT_OF_MEMORY
0x7	7	CTAERR_BAD_ARGUMENT
0x8	8	CTAERR_OUT_OF_RESOURCES
0x9	9	CTAERR_NOT_IMPLEMENTED
0xA	10	CTAERR_NOT_FOUND
0xB	11	CTAERR_BAD_SIZE
0xC	12	CTAERR_INVALID_STATE
0xD	13	CTAERR_FUNCTION_NOT_AVAIL
0xE	14	CTAERR_FUNCTION_NOT_ACTIVE
0xF	15	CTAERR_FUNCTION_ACTIVE
0x10	16	CTAERR_SHAREMEM_ACCESS
0x11	17	CTAERR_INCOMPATIBLE_REVISION
0x12	18	CTAERR_RESOURCE_CONFLICT
0x13	19	CTAERR_INVALID_SEQUENCE
0x14	20	CTAERR_DRIVER_OPEN_FAILED
0x15	21	CTAERR_DRIVER_VERSION
0x16	22	CTAERR_DRIVER_RECEIVE_FAILED
0x17	23	CTAERR_DRIVER_SEND_FAILED
0x18	24	CTAERR_DRIVER_ERROR
0x19	25	CTAERR_TOO_MANY_OPEN_FILES
0x1A	26	CTAERR_INVALID_BOARD

Hex	Decimal	Error name
0x1B	27	CTAERR_OUTPUT_ACTIVE
0x1C	28	CTAERR_CREATE_MUTEX_FAILED
0x1D	29	CTAERR_LOCK_TIMEOUT
0x1E	30	CTAERR_ALREADY_INITIALIZED
0x1F	31	CTAERR_NOT_INITIALIZED
0x20	32	CTAERR_INVALID_HANDLE
0x21	34	CTAERR_FILE_NOT_FOUND
0x22	33	CTAERR_PATH_NOT_FOUND
0x23	35	CTAERR_FILE_ACCESS_DENIED
0x24	36	CTAERR_FILE_EXISTS
0x25	37	CTAERR_FILE_OPEN_FAILED
0x26	38	CTAERR_FILE_CREATE_FAILED
0x27	39	CTAERR_FILE_READ_FAILED
0x28	40	CTAERR_FILE_WRITE_FAILED
0x29	41	CTAERR_DISK_FULL
0x2A	42	CTAERR_CREATE_EVENT_FAILED
0x2B	43	CTAERR_EVENT_TIMEOUT
0x2C	44	CTAERR_OS_INTERNAL_ERROR
0x2D	45	CTAERR_INTERNAL_ERROR
0x2E	46	CTAERR_DUPLICATE_CTAWAITOBJ
0x2F	47	CTAERR_NO_LICENSE
0x30	48	CTAERR_LICENSE_EXPIRED
0x34	52	CTAERR_WRONG_COMPAT_LEVEL
0x40	64	CTAERR_LOAD_LIB
0x41	65	CTAERR_SVR_COMM
0x42	66	CTAERR_BAD_NAME
0x43	67	CTAERR_ENUM_END
0x44	68	CTAERR_EVNT_BUF
0x45	69	CTAERR_INVALID_LIB
0x47	71	CTAERR_INCOMPATIBLE_SERVICES
0x10000001	268435457	CTAERR_INCOMPATIBLE_PARMS
0x10000002	268435458	CTAERR_DUPLICATE_EXTPARMS
0x10000003	268435459	CTAERR_WAIT_PENDING

Hex	Decimal	Error name
0x10000004	268435460	CTAERR_WAIT_FAILED
0x10000005	268435461	CTAERR_INVALID_ADDRESS
0x10000006	268435462	CTAERR_SERVICE_NOT_AVAILABLE
0x10000007	268435463	CTAERR_SERVICE_ERROR
0x10000008	268435464	CTAERR_SERVICE_IN_USE
0x10000009	268435465	CTAERR_ALREADY_DEFINED
0x1000000A	268435466	CTAERR_TRACE_NOT_ENABLED

Event summary

Natural Access events are defined in the *ctadef.h* include file. Events are prefixed with CTAEVN_. The following table lists the Natural Access events in alphabetical order:

Event name	Hex	Decimal	Description
CTAEVN_CLOSE_SERVICES_DONE	0x10002103	268443907	The specified services were closed on the context.
CTAEVN_DESTROY_CONTEXT_DONE	0x10002101	268443905	The context was destroyed.
CTAEVN_NULL_EVENT	0x10002100	268443904	**ctaWaitEvent** returned an error.
CTAEVN_OPEN_SERVICES_DONE	0x10002102	268443906	The specified services were opened on the context. Refer to **ctaOpenServices**.
CTAEVN_TRACE_MESSAGE	0X10002104	268443908	The buffer field of the CTA_EVENT structure points to a ***char*** array containing trace messages the server sent to the application.
CTAEVN_UPDATE_WAITOBJS	0x10002005	268443653	The internal array of wait objects managed by Natural Access changed. Refer to **ctaQueryWaitObjects**.
CTAEVN_WAIT_TIMEOUT	0x10002004	268443652	**ctaWaitEvent** timed out.

12 Application-specific parameters

Defining and using application-specific parameters

The Parameter Management (PRM) service enables you to manipulate application-defined parameters in the same manner as other Natural Access service parameters. After you initialize user-specified parameters through the PRM service, the application can retrieve and modify defined parameters in shared memory, process global memory, or on a context basis.

Define parameters in an ASCII text file and load them using the PRM service. The PRM service can load the ASCII definition files or encoded binary versions of the ASCII files. Encode your parameter files if you want to discourage modification by end users.

Complete the following steps to define and use application-specific parameters under Natural Access:

Step	Action
1	Create an ASCII text parameter file (*prm**xxx**.pf*, where *prm* is the prefix, ***xxx*** identifies the file, and *.pf* is the file extension) using any ASCII text editor (such as Microsoft Notepad, vi, emacs).
2	Encode the ASCII file into a binary parameter file using the *pfencode* conversion utility (optional).
3	Ensure that the resulting *.pf* file is in a directory that is contained in the list of directories specified by the CTA_DPATH environment variable so that the Parameter Management service automatically loads it at application startup. **Note:** The file name of the text parameter or encoded binary parameter file intended for PRM service must have the *prm* prefix as part of the parameter file name. Without this prefix, the PRM service will not load your parameter file.
4	Run *ctaparm* from the command line to ensure that Service = prm, prmmgr is specified in *cta.cfg*. Refer to the defined parameters listed in the output.
5	In the application, specify the PRM service in addition to all other desired Natural Access services in the list passed to **ctaInitialize**. A CTA_SERVICE_NAME structure should be initialized to `{ "PRM","PRMMGR" }` and included in the array of CTA_SERVICE_NAME structures passed to **ctaInitialize**.
6	Use the supported Natural Access parameter functions to access your parameters. An application parameter can be accessed by name or by ID, just like any other Natural Access parameter. The naming convention is similar to the Natural Access parameter naming convention, except that the service name must be set to PRM. Use the following syntax to access application-specific parameters: • PRM.category.fieldName • PRM.category.subStructure.fieldName

Creating a text parameter file

To declare application-specific parameters, use any ASCII text editor to create a text parameter file with a *.pf* file extension. Ensure that your text editor or word processor does not embed formatting information into the text parameter file.

This topic presents:

- Text parameter file syntax
- Sample .pf file

Text parameter file syntax

Specify the following information in a *.pf* file:

Keyword	Description
SvcName	Name of the service to contain the application parameters. Always set to PRM for application-specific parameters.
SvcID	Service ID of the service to contain the application parameters. Always set to 0xE for application-specific parameters.

Category	Named list that can contain substructure blocks, field names, or both. From a hierarchical perspective, a category provides the first level of name scoping under the service name. For example, RECORD is a category under the ADI Service, and the hierarchical name is ADI.RECORD.
	Terminate a category block with the keyword End. For example:
	`Category category name`
	`field descriptor or substructure`
	`End`

SubStructure	Named list that can contain field names and other substructure blocks. From a hierarchical perspective, a substructure provides a second level of name scoping (under a category). For example, callctl is a substructure under the START category of the ADI service and its hierarchical name is ADI.START.callctl.
	Substructure blocks are optional. Terminate a substructure block with the keyword End. For example:
	`SubStructure substructure name`
	`field descriptor`
	`End`

<table>
<tr><th>Keyword</th><th>Description</th></tr>
<tr><td>field descriptor</td><td>
The actual specification of a parameter. A field descriptor can be declared only within a Category or SubStructure block. A field descriptor specification begins with the name of the field itself and has the following form:
<pre>name, type, unit, [size,] default value</pre>
For example:
<pre>qualamp, INT32, dBm, -28</pre>
<table>
<tr><th>Parameter</th><th>Valid values</th></tr>
<tr><td>name</td><td>A field name must be unique within its scope. (There can be only one busycount field within the CALLPROG category. However, there could also be a busycount field in the COLLECT category.)</td></tr>
<tr><td>type</td><td>Supported data types are:
WORD (16-bit unsigned integer)
DWORD (32-bit unsigned integer)
INT16 (16-bit signed integer)
INT32 (32-bit signed integer)
BYTE_ARRAY
STRING
When the field data type is a string or byte-array, declare a maximum string size (in bytes) that is sufficient to store the default string. Refer to the sample .pf file on page 268.</td></tr>
<tr><td>unit</td><td>Supported units are:
INTEGER
COUNT
MASK
HZ
MS
DB
DBM
IDU (internal data unit)
PERCENT
STRING
The STRING units specification must be used when the field data type is a string.</td></tr>
<tr><td>size</td><td>When declaring a STRING, you must also declare a maximum string size (in bytes) sufficient to store the largest expected string and the null terminator.</td></tr>
<tr><td>default</td><td>Value with which to initialize the parameter. String values must be surrounded by double quotes. Integer values can be entered in either hexadecimal or decimal format.</td></tr>
</table>
</td></tr>
</table>

Use the following syntax for a text parameter file:

Item	Description
Comments	A number sign (#) is a comment character and can be used anywhere within a text parameter file. All characters following the # are ignored on the current line. Functionality is similar to # in UNIX shell scripts or the rem command in DOS batch files. A blank line in the text parameter file is also ignored.
Special characters	Use a backslash (\) to escape reserved characters. For example, use \# to enter a literal # character. Use \\ to enter a literal \ character.
Case	The PRM service and utilities are not case sensitive. For example, keywords and service names can be in uppercase, lowercase, or any combination.
Naming rules	Categories, subcategories, and fields are named entities. In order to name them, the following rules apply: • The legal alphabet includes all alphanumeric characters (a-z, 0-9) and underscore (_). • The first character of a name cannot be an underscore or a numeric character. • White space is not allowed in a name. • Name length is not restricted (however, typical names do not exceed 20 characters in length).
Delimiters	Separate keywords and declarations with a space or a comma. Separate declarations containing multiple pieces of information with a space or commas.

Sample .pf file

The following example declares a subset of the ADI service parameters as separate, autonomous application parameters.

The parameter list is:

- PRM.TONEDETECT.qualampl
- PRM.TONEDETECT.qualtime
- PRM.START.CALLCTL.blockmode
- PRM.START.CALLCTL.eventmask
- PRM.START.DIAL.interpulse
- PRM.START.DIAL.method
- PRM.START.t_str1
- PRM.START.testbytearray

Sample file:

```
#############################################################################
# Start of text parameter file
#############################################################################
#############################################################################
# PRM Service Name and ID
#############################################################################
SvcName         PRM     # do not modify this line for PRM Service
SvcID           0xE     # do not modify this line for PRM Service
#############################################################################
# Category TONEDETECT
#############################################################################
Category        TONEDETECT
        qualamp1, INT32, dBm, -28       # min. detectable signal level
        qualtime, DWORD, ms,   500      # min. duration of detected tone
End
#############################################################################
# Category START
#############################################################################
Category        START
        Substructure    callcontrol
               blockmode,  DWORD, Integer, 0           # blocking method
               eventmask,  DWORD, Mask,    0x00000000  # event reporting
        End
        Substructure    dial
               interpulse, DWORD, Ms,      700         # interdigit pulse time
               method,     DWORD, Mask,    0X00000000  # signaling type
        End
        t_str1, STRING, STRING, 50, "sample string"    # Just an example
                                                       # of a string
        testbytearray, BYTE_ARRAY, IDU, 5, 0x1, 0x2, 0x3, 0x4, 0x5
End
#############################################################################
# End of text parameter file
#############################################################################
```

Index