Dialogic
Making Innovation Thrive

Dialogic® NaturalAccess™ Service Writer's Manual

October 2009 64-0496-01

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
9000-6405-10	June 1998	JLC (updated from MEL, SRG)
9000-6405-12	October 1998	ERS, CT Access 2.1
9000-6405-13	July 2000	SJC, template update
9000-6405-14	March 2001	SRR, NACD 2000-2
9000-6405-15	November 2001	CYF, NACD 2002-1 beta
9000-6405-16	May 2002	CYF, NACD 2002-1
9000-6405-17	April 2003	MVH, NA 2003-1
9000-6405-18	April 2004	MCM, NA 2004-1
64-0496-01	October 2009	LBG, NaturalAccess R9.0
Last modified: September 3, 2009		

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

Table Of Contents

Chapter 1: Introduction 11

Chapter 2: Terminology 13

Chapter 3: Natural Access architecture 15
Review of distributed computing concepts 15
Basic distributed computing concepts 15
Remote procedure call 16
Summary 17
Natural Access architecture overview 18
Managed resource 18
Service implementation 19
Service API 19
Service SPI 19
Service manager 20
Dispatcher 20
Dispatcher functionality 21
Communications 21
Built-in dispatcher services 21

Chapter 4: Natural Access service overview 25
Service components 25
Service name 26
Service ID 26
Software structure 26
Binary files 26
Source files 27
tik service examples in this manual 29
Template source files 30

Chapter 5: Service writing overview 31
Writing a service 31
Defining a service 31
Implementing a service 32
Coding conventions 32
Function names 33
Function arguments 33
Return values 34
Event and reason names 34
Error, reason, event, command, and trace tag codes 35
Manifest constants and enumerations 36
Structures 36
Parameter structures 36

Chapter 6: Defining a service 37
Understanding the managed resource 37
Getting a service ID 40
Defining the API 40
Defining events and reasons 41
Defining errors 43

Defining parameters 44
Defining tracing 45
Service-specific trace record categories 45
Defining trace tags 46

Chapter 7: Implementing the API 49
API implementation overview 49
Creating the header files 49
Defining the API function prototypes 50
Defining the API function command codes 50
Defining error codes 51
Defining event and reason codes 51
Defining the trace tags 52
Writing the API function code 52
Application API command flow 53
Calling the SPI function 53
Error handling macros 53
Example API function 54
Declaring parameters 55
Creating the parameter definition file 55
Generating the parameter files 57

Chapter 8: Implementing the SPI 59
SPI implementation overview 59
Declaring SPI function prototypes 59
Implementing the SPI functions 60
Service SPI command flow 60
DISP_COMMAND structure 61
DONE events associated with asynchronous API functions 64
Example SPI function 65
Implementing client-side binding functions 65

Chapter 9: Implementing the service manager 67
Service manager overview 67
Binding functions 67
Life cycle binding functions 69
Life cycle binding functions: registration 72
Life cycle binding functions: event handling 73
Life cycle binding functions: service startup 74
Life cycle binding functions: service shutdown 75
Registering services 76
xxxInitializeManager 77
xxxDefineService 83
Initializing managed resource event handling 85
Starting services 89
xxxOpenServiceManager 91
xxxOpenService 92
Shutting down services 95
xxxCloseService 95
xxxCloseServiceManager 96
Shutting down event handling 96
Runtime binding functions 98
Command processing 101

Error handling 102
Trace handling 105
Event processing 108
Additional binding functions 108

Chapter 10: Implementing the service 109
Service implementation overview 109
Implementing service functions 110
Arguments to service implementation functions 111
Handling service parameters as function arguments 111
Processing events from the managed resource 114
DISP_EVENT structure 117
Generating trace records 119

Chapter 11: Service object handles 121
Service objects 121
Service handles 123
Server-side object handles 123
Server-side context handles 124
Client-side context handles 125
Client-side object handles 125
Mapping server handles to client handles 126
Service object sharing 129
Peer-to-peer sharing 129
Query and attach 133
Unsolicited event 133

Chapter 12: Natural Access runtime control 135
Runtime control overview 135
Conditions and actions 135
RTC function summary 137
Program flow 137
Using aliases 142
Overview of using aliases 142
Program flow 144
Registering for RTC notification 147
Canceling RTC notification 148
Reading the RTC database 149
Sample code 150
Natural Access base services RTC aliases 152

Chapter 13: Service manager functions 153
Service manager function summary 153
Life cycle binding functions 153
Runtime binding functions 154
Using the service manager function reference 155
xxxAddRTC 156
xxxAttachServiceManager 158
xxxCloseService 159
xxxCloseServiceManager 161
xxxDefineService 162
xxxDetachServiceManager 163
xxxFormatMessage 164

xxxFormatTraceBuffer 165
xxxGetText 166
xxxInitializeManager 167
xxxOpenService 168
xxxOpenServiceManager 170
xxxProcessCommand 171
xxxProcessEvent 173
xxxRemoveRTC 174
xxxSetTraceLevel 175

Chapter 14: Dispatcher service functions 177
Dispatcher service function summary 177
Service manager and service registration 177
Event queue processing control 177
Parameter modification 178
Service command queuing and sending 178
Handle management 178
Wait object 179
Error and tracing log 179
Service information retrieval 179
Buffer management 179
Runtime control 180
Application initialization information 180
Server command 180
Using the dispatcher service function reference 181
dispAddRTC 182
dispAllocBuffer 185
dispApiError 186
dispAttachHandle 187
dispCheckParmByName 188
dispCreateHandle 189
dispDestroyHandle 191
dispError 192
dispFindHandle 194
dispFindHandleByName 195
dispFindWaitObject 196
dispFreeBuffer 197
dispGetActiveRTC 198
dispGetAppCtl 199
dispGetHandleValue 201
dispGetHandleValueEx 202
dispGetNumRTC 203
dispGetNumRTCAliases 204
dispGetParmByName 205
dispGetParms 206
dispGetQueueHandle 207
dispGetRTCAliases 208
dispGetText 209
dispGetTracePointer 210
dispLogTrace 211
dispMakeAndQueueEvent 213
dispQueueCommand 214
dispQueueEvent 215

dispRegisterRTCAlias 217
dispRegisterService 218
dispRegisterServiceManager 220
dispRegisterWaitObject 223
dispRemoveRTC 225
dispSendCommand 227
dispSetAsyncIn 229
dispSetAsyncOut 230
dispSetHandleValue 231
dispSetHandleValueEx 232
dispSetParmByName 233
dispShutdown 234
dispUnregisterRTCAlias 235
dispUnregisterWaitObject 236

Chapter 15: Demonstration programs and utilities 237
Summary of the demonstration programs and utilities 237
tik service test program: tiktst 238
EVT service test program: evttest 240
Parameter file generation utility - pf2src 244

Chapter 16: Errors and events 245
Error codes 245
Alphabetical error summary 245
Numerical error summary 251
Events 254

Chapter 17: Buffer management summary 255
Buffer passing rules 255
DISP_COMMAND buffer flags 255
DISP_EVENT buffer flags 256

1 Introduction

The *Dialogic® NaturalAccess™ Service Writer's Manual* provides the information necessary to write a service that fits into the NaturalAccess framework.

This manual requires that the reader be familiar with the NaturalAccess concepts and terminology discussed in the *Dialogic® NaturalAccess™ Software Developer's Manual*. Previous experience with writing an application with NaturalAccess is helpful.

The *Dialogic® NaturalAccess™ Service Writer's Manual* is part of the Service Writer's Kit. Full sample code including a sample NaturalAccess service and test application can be found in the Service Writer's Kit associated with this manual.

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 Natural Access architecture

Review of distributed computing concepts

The Natural Access internal architecture is based on the client/server model of distributed computing. This topic briefly reviews distributed processing concepts and implementation strategies.

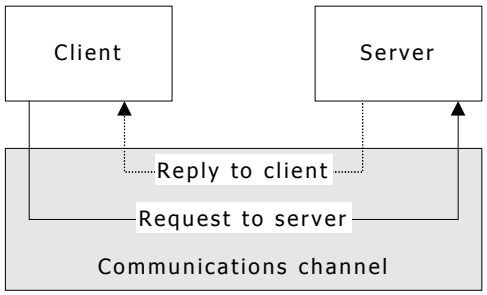
In this topic:

- Basic distributed computing concepts
- Remote procedure call (RPC)
- Summary

Basic distributed computing concepts

Distributed computing refers to two or more processes (which can be located on different computers) that interact with each other over a communications channel in order to accomplish some computing task. The client/server model is a popular model for implementing distributed computing. In this model, two different sets of code, known as the client and the server, are produced. These parts are typically located on different computers within a network, although they may exist within different processes on the same computer, or even within different libraries or threads within the same process.

The client makes a request over a communications channel and waits for a reply. The server waits for incoming requests, processes the request, and returns a response. In this model of computing (shown in the following illustration), the server is responsible for coordinating and providing access to a resource on behalf of one or more clients. The resources controlled by a server can be any computing resource such as a database, a physical device, or even another server process.



Client server model

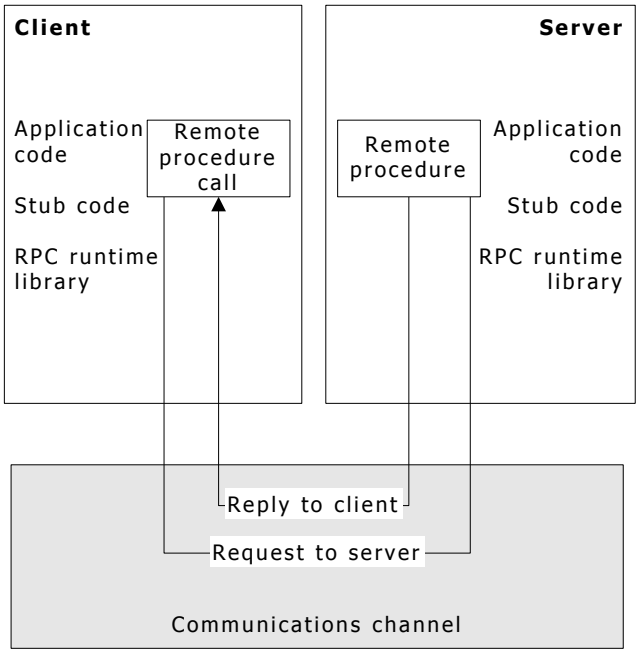
Writing an application that works in the raw client/server environment requires detailed knowledge of inter-process and network communications mechanisms. Implementing applications in this environment deals with communications details rather than with server implementation details.

Remote procedure call

A remote procedure call (RPC) is a way of implementing the communications between the client and server in the client/server model. The main strength of RPC is that the details of using inter-process and network communications mechanisms are hidden.

Using RPC, the client makes a procedure call that looks like a local call from the client's perspective. The procedure call is translated into a message and sent to the server using the RPC specific communications protocol. Once the server receives the request, it is translated back to a procedure call and the appropriate server procedure is executed. Responses are returned in a similar fashion.

RPCs are implemented using stubs and an RPC runtime library as shown in the following illustration. Stubs contain functions which map simple local procedure calls into a series of RPC function calls. RPC function calls are the communication interface implementing the RPC protocol. They also specify how the messages are constructed and exchanged.



Remote procedure call (RPC)

The client and server both communicate using stubs, one for the client and one for the server. The client stub calls RPC library procedures to:

1. Package (marshal) the arguments and ID of the call into a message.
2. Find the server.
3. Transmit the message to the server using the RPC runtime library.
4. Wait for the response from the server.
5. Return the response to the caller.

The server stub calls RPC library procedures to:

1. Receive the incoming message from the RPC runtime library.
2. Unpack (unmarshal) the arguments and procedure id.
3. Invoke (perform an upcall to) the appropriate server procedure.
4. Marshal the results.
5. Send a response back to the client.

In the client/server and RPC environments, the term service describes the remote procedure call (service interface) and remote procedure implementation (service implementation) components of the server. The service interface component is typically packaged as a library to be linked into a client application. Similarly, the service implementation component is packaged as a separate library to be linked into a server process (a daemon). Both components (both libraries) can be linked into the same application, effectively making one process operate as both client and server.

As described earlier, the internal architecture of Natural Access is based on the client/server model. The internal implementation strategy of Natural Access is loosely based on RPC. The outcome is a highly efficient, distributable infrastructure for telephony services.

Summary

The Natural Access architectural model is based on the RPC implementation strategy of the client/server model. Rather than using RPC directly, Natural Access was developed to address the following additional requirements beyond what is specified by RPC:

- Event handling

  RPC is a request/reply paradigm. The service can only respond to requests from a client. Natural Access also requires the ability for a service to generate unsolicited events and send them to clients.

- Built-in services

  RPC does not provide an infrastructure of system-level services such as parameter management. Natural Access requires a rich infrastructure of built-in services.

- Efficiency

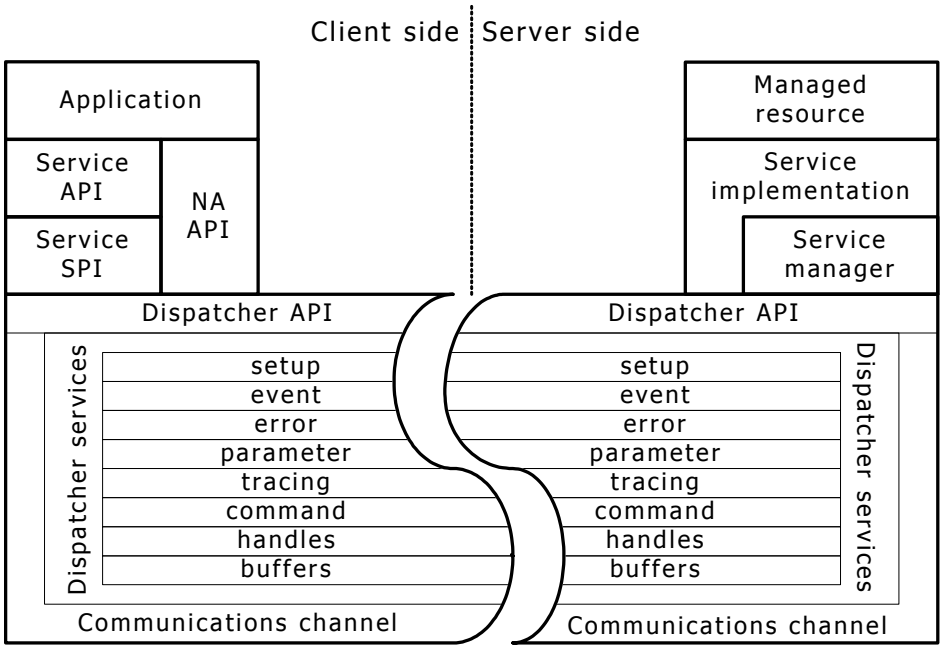
  Access to Natural Access services must be very efficient with very little overhead in processing a message from the service interface to the service implementation.

Natural Access architecture overview

The major components in the Natural Access architecture are:

- Service implementation
- Service API
- Service SPI
- Service manager
- Dispatcher

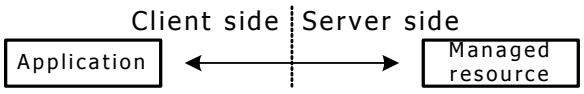
The following illustration shows the components in the Natural Access architecture:



Natural Access architectural model

Managed resource

Natural Access provides an environment for an application to interact with a managed resource such as a telephony board.

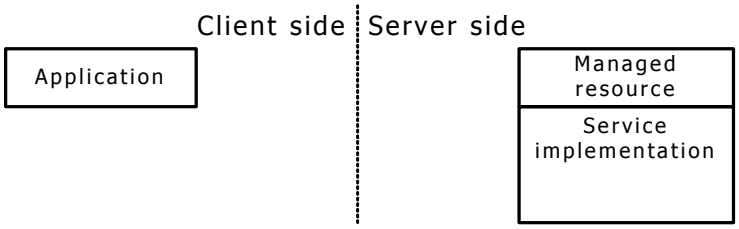


Managed resource

The Natural Access application corresponds to an RPC client. The managed resource corresponds to a managed resource in RPC. Client-side and server-side components can reside on the same computer, or on different computers.

Service implementation

The service implementation provides the underlying (detailed) logic to interact with the managed resource.



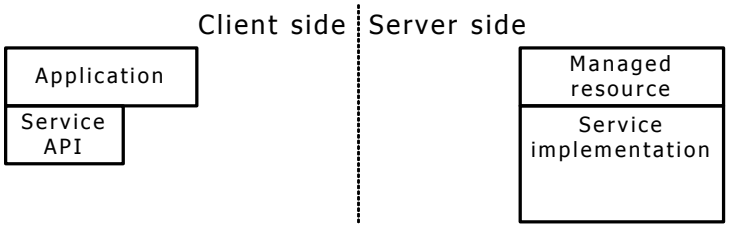
Service implementation

A Natural Access service implementation corresponds to an RPC service implementation.

Service API

In order to access the managed resource through the service implementation, a Natural Access application makes calls on a well-defined API. The service API is the set of function calls, including arguments and managed parameters, return values, and asynchronous events, that define the work that is performed by the service.

The service API acts as a contract specifying what a client can expect and what an implementation must provide.



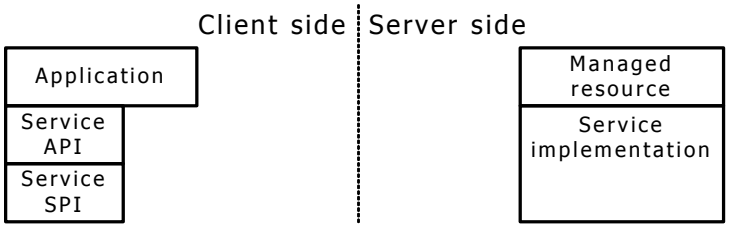
Service API

The service API corresponds to the list of remote procedure calls in RPC.

Service SPI

The service SPI (service provider interface) converts the API call into a message that is suitable for transmission to the service implementation.

Services communicate with other services by calling their SPI functions, not their high level API.



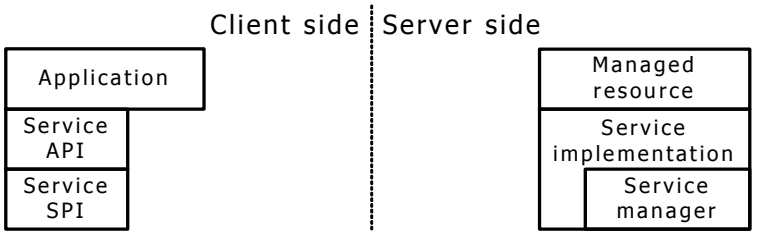
Service SPI

The service SPI corresponds to the RPC client side stub.

Service manager

The service manager is a set of well-known, Natural Access-specified functions (known as binding functions) to be implemented on a per-service basis.

Among other things, the service manager implements binding functions to start up and shut down the service implementations, and functions to perform upcalls to service implementation functions.



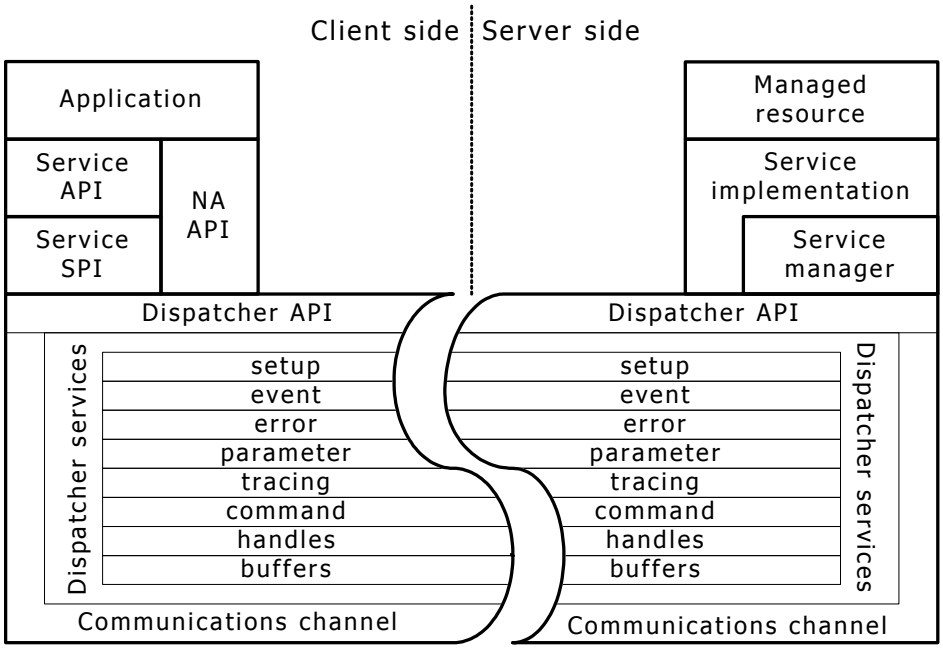
Service manager

The service manager corresponds to RPC server side stubs and a portion of the RPC runtime library having to do with upcall processing.

Dispatcher

The dispatcher provides the communications channel between the service interface and the service implementation components. If the client-side and server-side components reside on different computers, for example, the dispatcher provides the communications link between the two sides.

The dispatcher also provides a set of built-in services addressing common, system-level programming needs. The built-in dispatcher services are implemented across the client/server separation.



Dispatcher functionality

The dispatcher services are called using dispatcher functions (**disp** functions). The service manager binding functions interact with corresponding dispatcher built-in services.

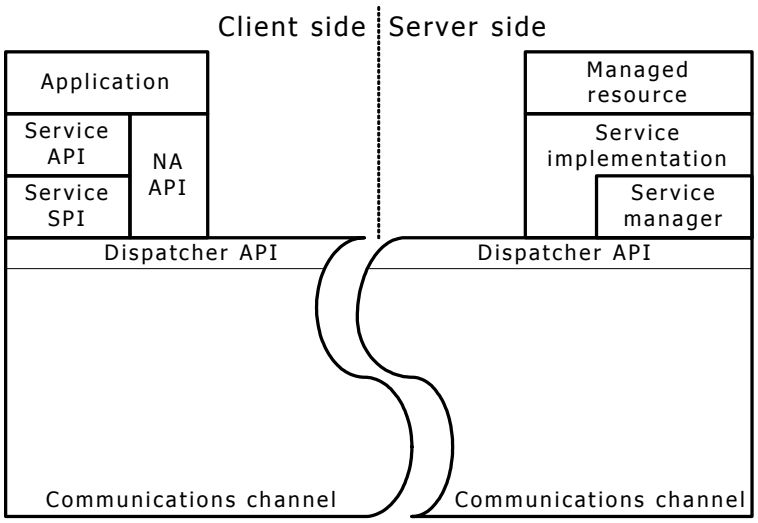
Dispatcher functionality

The dispatcher is responsible for providing functionality for:

- Communications
- Built-in services

Communications

The dispatcher provides and maintains a communications path between the SPI and the service manager, even between different computers. By using dispatcher functions, details of network programming are hidden from the service developer.



Dispatcher communications

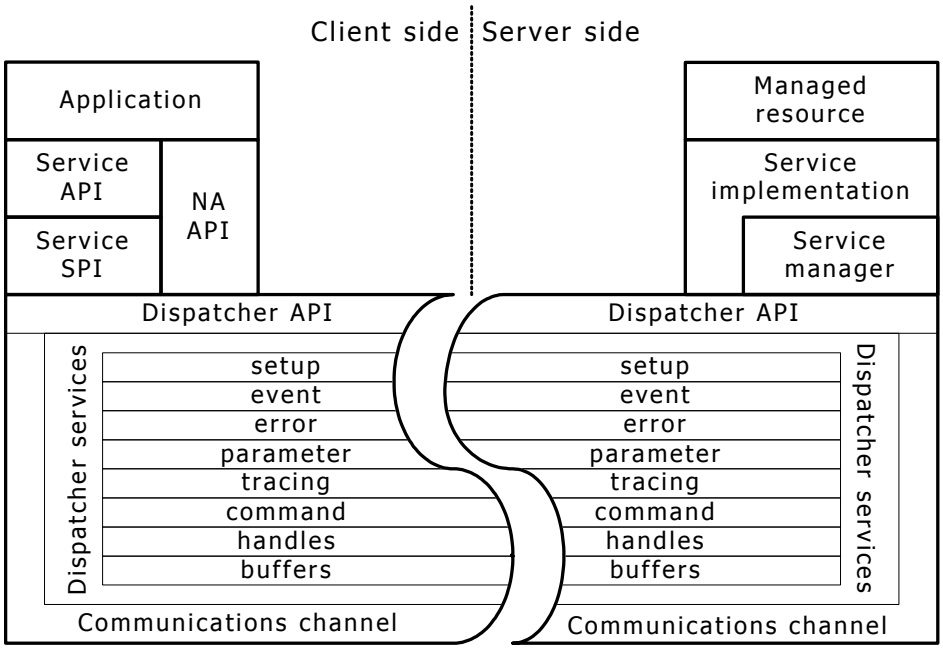
The Natural Access dispatcher corresponds to portions of the RPC runtime library having to do with low level IPC and network communications.

Built-in dispatcher services

The dispatcher built-in services include:

- Service setup
- Command processing
- Event handling
- Parameter management
- Tracing
- Service object handles
- Buffers

The following illustration shows the dispatcher built-in services:



Dispatcher built-in services

The built-in dispatcher services are implemented across the client/server separation. They are not exclusively owned by either side.

Service setup

When setting up a Natural Access application, services and service managers are registered and initialized by calling dispatcher functions. These dispatcher functions declare the version numbers, compatibility levels, and binding function addresses for the service.

Command processing

When an application calls a service API function, a command buffer is passed to the dispatcher. The dispatcher invokes the appropriate service manager binding function which, in turn, performs an upcall to the corresponding service implementation function.

Event handling

Conceptually, an event service is a mechanism that allows:

- Event producers to publish their events in a well-known location.
- Event consumers to register their interest in certain events and, when those events are published, to retrieve them.

An event service does not impose a restriction on the types of event information that can be published nor does it inform the consumer as to the contents of an event. The content of an event is completely determined and documented by the producer of the event.

However, an event service defines the transport structure to encapsulate the raw event published by the producer and retrieved by the consumer. By doing so, the mechanics of sending an event from the producer to the consumer are greatly simplified since the event service is trafficking in only one common data structure.

The dispatcher includes a built-in service to provide streamlined event service functionality. The built-in dispatcher event service:

- Defines a generic event structure for transport of events between Natural Access services and Natural Access applications. This generic event structure encapsulates raw event data produced by a service.
- Provides the ability for a Natural Access service to define and publish events to a specific consumer (either another Natural Access service or a Natural Access application).
- Provides the ability for a Natural Access service to process an incoming event.
- Provides the ability for a managed resource to publish raw (unencapsulated) event data to a specific Natural Access service.
- Provides the ability for a Natural Access service to process an incoming raw event from a managed resource using a specific callback function.
- Provides the ability for independently operating Natural Access services to be notified when particular service events are generated. This feature is called runtime control.

Error handling

Natural Access handles errors in API calls by calling an error handler. The default error handler returns the error as the function return value. An application can override error handling by registering its own error handler with Natural Access. Internal errors in services or service managers are sent to the trace log if tracing is enabled.

Parameter management

Each service has a defined set of parameters that allow dynamic modification of service behavior. These parameters are specified by each service, but are managed by the dispatcher. Each context maintains the parameter values for each service opened on the context. There are also Natural Access server defaults for service parameters that may be either shared between processes or held locally in process memory. Parameter default values are copied from the Natural Access server defaults to a context when it is created. Natural Access parameter functions can be used by an application to modify either the parameter values in a specific context or the Natural Access server default parameter values.

Service APIs support the passing of structures for parameter values. Parameters passed in API calls modify the behavior of the function but do not modify the default parameter values stored in the context.

Each service has a set of standard parameters that all conforming service implementations must expose. Natural Access also provides the capability for services to define extension parameters. Currently, Natural Access services do not define any extension parameters.

Tracing

To facilitate debugging of both services and applications, Natural Access provides a built-in dispatcher service to log warning, error, and informational messages to a central location. These messages are referred to as trace records. The trace records can then be viewed using the Natural Access Server (*ctdaemon*). Refer to the *Natural Access Developer's Reference Manual* for more information on viewing trace records this way.

There are a variety of trace record categories that are supported by the dispatcher built-in tracing service. For some of these categories, the Natural Access dispatcher automatically generates the trace records. The remaining trace record categories are service-specific. For service-specific trace records, each Natural Access service must associate corresponding trace records and explicitly generate them.

The dispatcher also provides a mechanism (a tracemask) to identify which trace record categories are active. If the trace record category is active, the associated trace records are generated and logged.

Service object handles

The service object is a functional unit within a service that represents a managed resource and is associated with a Natural Access context. Service object handles are used as the reference to real service objects that are managed by the service implementation. Two kinds of object handles are supported in Natural Access:

- A server-side object handle provides service implementation access to resources and resides on the server.
- A client-side object handle provides the client of the service access to objects on the server.

Mapping information between the client side and server side object handles must be maintained by all services interfaces. The Natural Access dispatcher includes general functions to handle mapping table logic.

Buffers

The Natural Access dispatcher includes a set of specialized functions for handling of data buffers. Due to the client/server architecture, there are some issues you must understand and account for when passing data buffers, either through an API function call or through a service generated event.

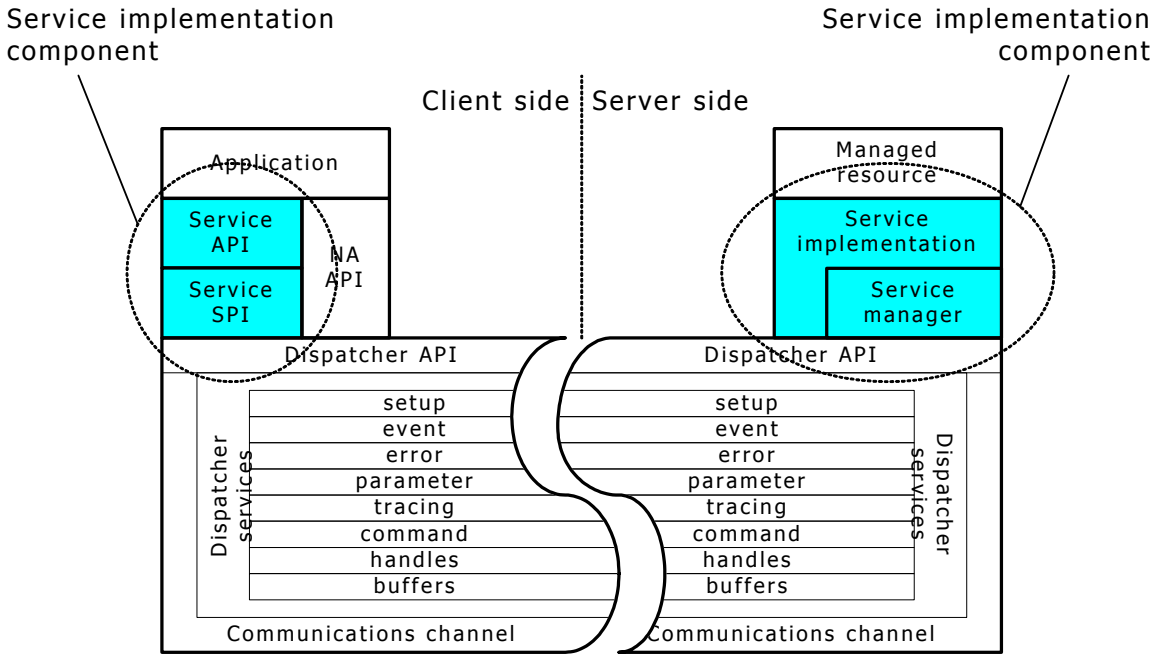
4 Natural Access service overview

Service components

The following components comprise a Natural Access service:

- Service API
- Service SPI
- Service implementation
- Service manager

The following illustration shows the Natural Access architectural model:



Natural Access service

The service interface component of a service includes the service API and the service SPI. The service implementation component includes the service manager and the service implementation.

Each service has a service name and service ID.

Service name

Each Natural Access service is assigned a unique three letter acronym. This acronym is an integral part of the naming conventions for the components of a service. For example, the AG Board Driver Interface is referred to as the ADI service.

In this manual, the letters ***xxx*** are used to represent the three letter acronym assigned to a service.

Service ID

Since Natural Access supports multiple services, any of which can be accessible to a client application, an internal mechanism is required to uniquely identify each service. This mechanism is the service ID.

The service ID is a 16 bit numeric value that is embedded in all command, error, and event messages. This allows the dispatcher to route messages between the service interface and the service implementation components.

Software structure

This topic describes the contents and naming conventions for binary files and source files of a Natural Access service.

Binary files

Two DLLs or libraries are created for a service: one for the service interface and one for the service implementation. Developers link with the interface library when compiling an application. The corresponding implementation library is dynamically loaded at runtime. The CTA_SERVICE_NAME structure passed to **ctaInitialize** controls which service interface the application uses and which implementation of that interface should be dynamically linked in. For example, using ADI, ADIMGR states that the application is making calls to the ADI service and that the implementation of those calls is performed by ADIMGR (for the AG hardware). Similarly, using ADI, QDIMGR, directs the application to make calls to the ADI service, and directs QDIMGR to perform the implementation of those calls. In both cases, the application links with *adiapi.lib*.

Having two libraries allows a service developer to leverage an existing service by simply re-implementing the service implementation. For example, the ADI interface can be used for another vendor's telephony board by creating a new service implementation component that communicates with the board. The service interface remains the same.

The naming conventions for service binary files are:

	Windows DLL	UNIX library
Service interface	*xxxapi.dll*	*libxxxapi.so*
Service implementation	*xxxmgr.dll*	*libxxxmgr.so*

The appropriate implementation library is dynamically loaded at runtime. The loading process is controlled by the CTA_SERVICE_NAME structure passed to **ctaInitialize**.

For example:

This service and service manager...	links with...	and loads...
ADI, ADIMGR	*adiapi.lib*	*adimgr.dll*
ADI, QDIMGR	*adiapi.lib*	*qdimgr.dll*

The service implementation can be written in either C or C++. However, the service interface must be written in C to prevent issues associated with name mangling. Ensure that function declarations within header files are declared as having C style linkage characteristics, so that all functions are callable by both C and C++ programs. For instance, header files should contain:

```
/* At the top of the header: */
#ifdef __cplusplus
extern "C" {
#endif

...

/* At bottom of header:      */
#ifdef __cplusplus
}
#endif
```

Source files

The service interface typically consists of the following source files:

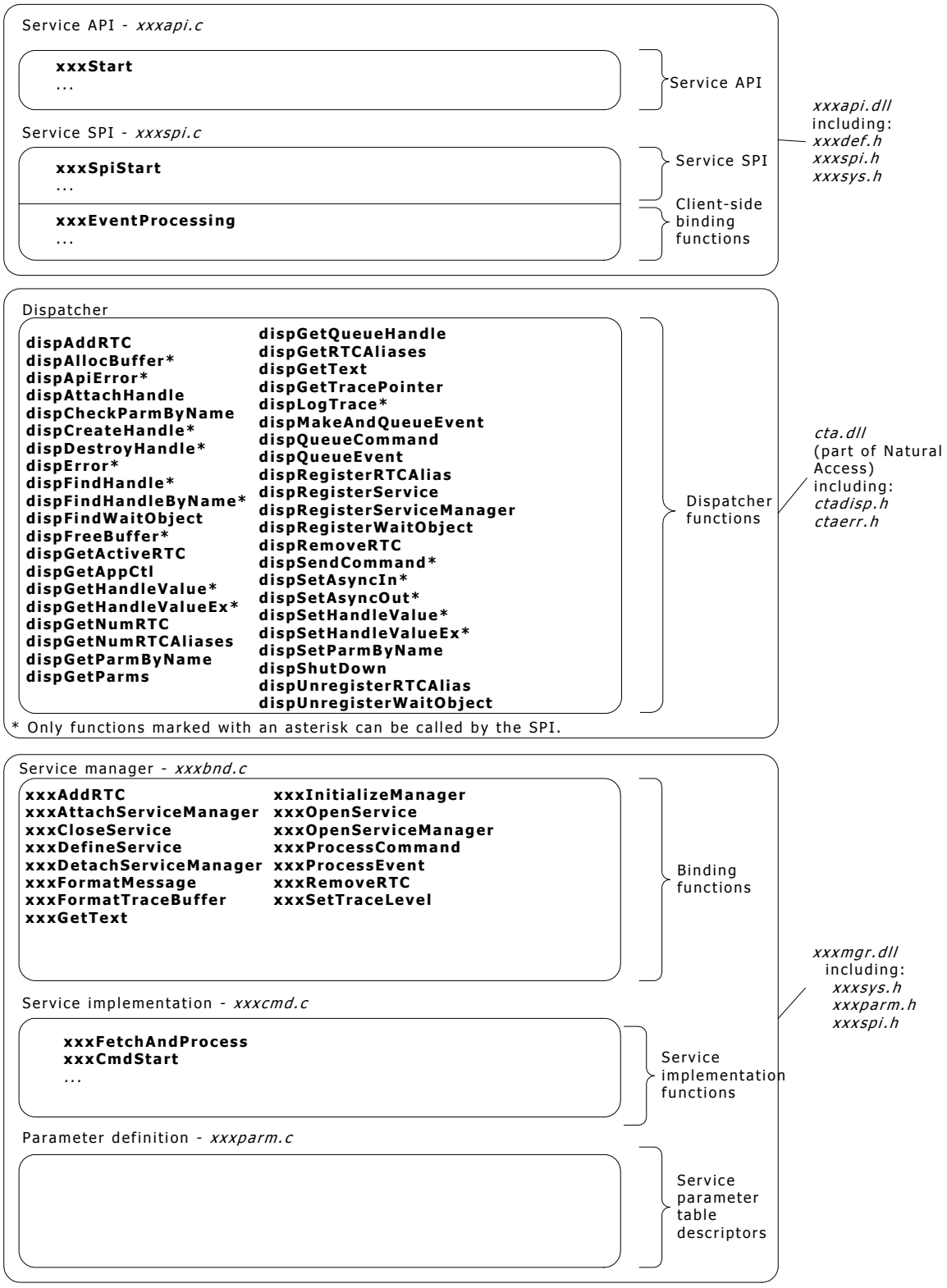
File	Description
xxxapi.c	Implements the service API by calling the corresponding SPI function and handling API errors.
xxxspi.c	Implements the service SPI functions and the client side binding functions.
xxxdef.h	Header file containing API data structures, function prototypes, and definitions.
xxxspi.h	Header file containing SPI function prototypes.
xxxparm.pf	ASCII file containing definition of service parameters.

The service implementation typically consists of the following source files:

File	Description
xxxbnd.c[*cpp*]	Implements the service manager (binding) functions.
xxxcmd.c[*cpp*]	Implements the service functions that interact with the managed resource.
xxxparm.c[*cpp*]	Parameter source file automatically generated from *xxxparm.pf*.
xxxparm.h	Parameter header file automatically generated from *xxxparm.pf*.
xxxsys.h	Header file containing internal data structures, function prototypes, and definitions.

Additional source files for a service can also exist as required. These files should also follow the Natural Access naming conventions (for example, *xxx**something**.c*). File names should follow the 8.3 format convention.

The following illustration shows the relationship between service source files, binaries, and the dispatcher functions available to service writers:



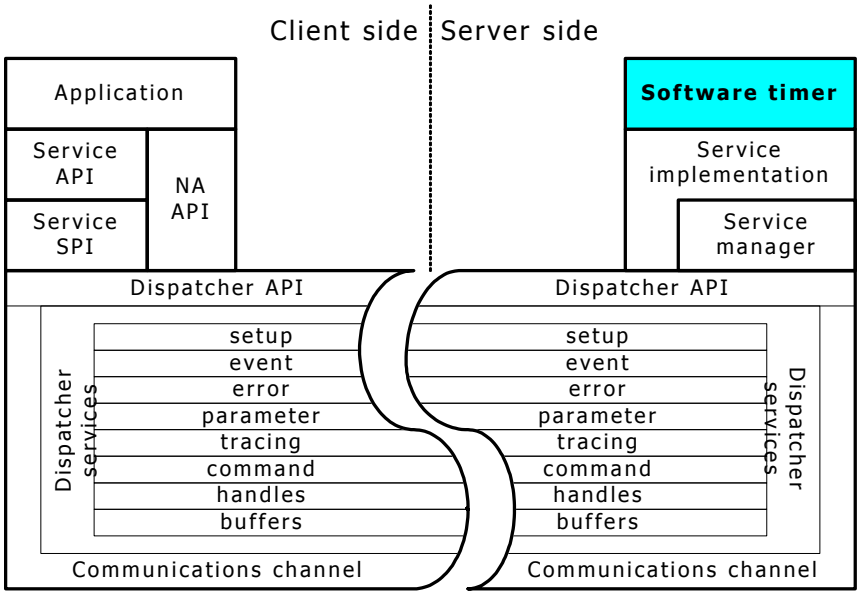
Natural Access software structure

tik service examples in this manual

The Natural Access software is accompanied by source code to the tik sample service. The goal of the tik service is to embed software timer functionality (a ticker) into Natural Access by:

- Encapsulating the details of interacting with the software timer into a Natural Access service.
- Presenting a simple API to Natural Access applications.

The implementation of the software timer is meant to model a physical device. The software timer sends a command to the device and receives asynchronous messages back. From an application's perspective, the software timer is simply another managed resource available under Natural Access, as shown in the following illustration:



Software timer service architecture

In subsequent topics of this manual, each phase of writing a Natural Access service is illustrated using code fragments from the tik service. All tik-specific information is displayed in a tik service section that looks similar to the following illustration:

tik service

```
tik example information
```

All source code for the tik service is provided as part of the Natural Access Developer's kit and can be found in the *nms**ctaccess**demos**tiksvc* directory. Refer to this source code as you read the information about creating a Natural Access service.

The tik service includes the following files:

File	Description
tikapi.c	Implements the service API by calling the corresponding SPI function and handling API errors.
tikspi.c	Implements the service SPI functions.
tikdef.h	Contains API data structures, function prototypes, and definitions.
tikspi.h	Contains SPI function prototypes.
tikbnd.c	Implements the service manager binding functions.
tikcmd.c	Implements the service functions which interface to the managed resource.
tikcomms.c	Contains communication logic to tik server (the driver interface).
tikutils.c	Contains miscellaneous functions.
tikparm.pf	ASCII parameter definition file (used to generate *tikparm.c* and *tikparm.h*).
tiksys.h	Contains internal data structures, function prototypes, and definitions.
tikapi.def	Windows export definition file for tik API.
tikmgr.def	Windows export definition file for tik service manager.
makefile	Makefile to build the tik service.

Template source files

To facilitate writing a service, a set of template source files (based on the tik service) is provided in *nms**ctaccess**template**ctatmplt*.

Template files exist for each of the files listed in the previous table. Use these skeleton template files to start constructing a new service.

5 Service writing overview

Writing a service

There are two phases in writing a service:

- The definition phase, when the requirements of the managed resource are defined.
- The implementation phase, when the actual software components for the service are developed and tested.

Defining a service

To define a service:

Step	Description	See...
1	Understand the managed resource and how to best interact with it.	*Understanding the managed resource* on page 37
2	Design an interface (a set of API function calls and their associated errors, events, and reasons).	• *Defining the API* on page 40 • *Defining events and reasons* on page 41 • *Defining errors* on page 43 • *Defining parameters* on page 44
3	Determine the tracing capability for debugging purposes.	*Defining tracing* on page 45

Implementing a service

To implement a service:

Step	Description	See...
1	Write the service API.	*API implementation overview* on page 49
2	Write the service SPI.	*SPI implementation overview* on page 59
3	Implement the service manager functions for initialization and shut down.	• *Registering services* on page 76 • *Initializing managed resource event handling* on page 85 • *Starting services* on page 89 • *Shutting down event handling* on page 96 • *Shutting down services* on page 95
4	Implement the service manager functions for command and event processing.	• Command processing • Event processing
5	Implement the service manager functions for tracing and error handling.	• Error handling • Trace handling
6	Implement the service functions that interact with the managed resource.	*Implementing service functions* on page 110

Coding conventions

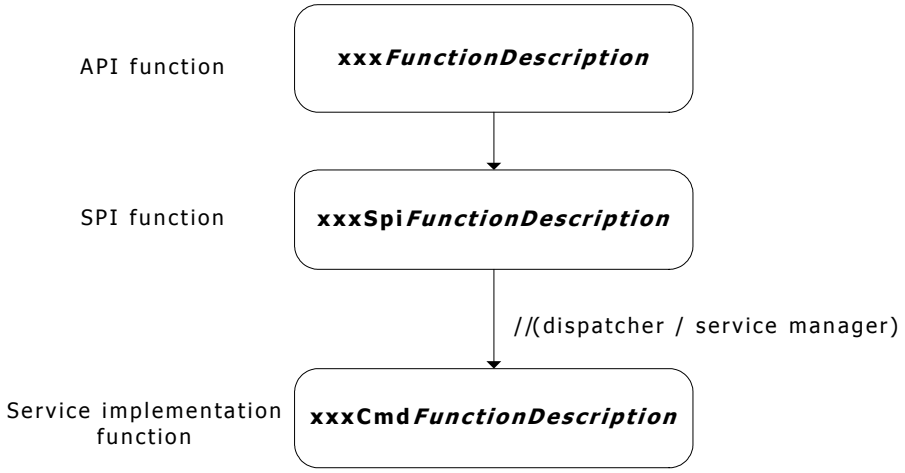
Use the conventions described in this topic when writing the service interface for a Natural Access service. Natural Access includes conventions for:

- Function names
- Function arguments
- Return values
- Error, reason, event, command, and trace tag codes
- Manifest constants and enumerations
- Structures
- Parameter structures

Function names

All API functions for the service begin with the service's three letter acronym (for example, **adiStartCallProgress**).

For each API function, the corresponding SPI function and implementation function must use the following naming convention:



API and SPI function names

The naming convention for the managed resource event handler is **xxxFetchAndProcess**. All other functions should follow the convention **xxxFunctionDescription**.

Function arguments

Arguments to functions can be one of the following types:

Argument type	Description
in	Argument provides an input value for the function.
out	Argument is a buffer to place output from the function.

A single argument cannot be used for both input and output (inout). This is not merely a convention, but is mandated by internal Natural Access processing of **dispSendCommand**.

In arguments should precede out arguments whenever possible.

Argument names should be lower case without underscores. Underscores can be used if the argument name is long and would be unreadable or misread without them. Arguments should be weakly typed (for example: int or unsigned) instead of strongly typed (for example: INT32 or DWORD).

Certain data types cannot be used as arguments to API or SPI functions. These are:

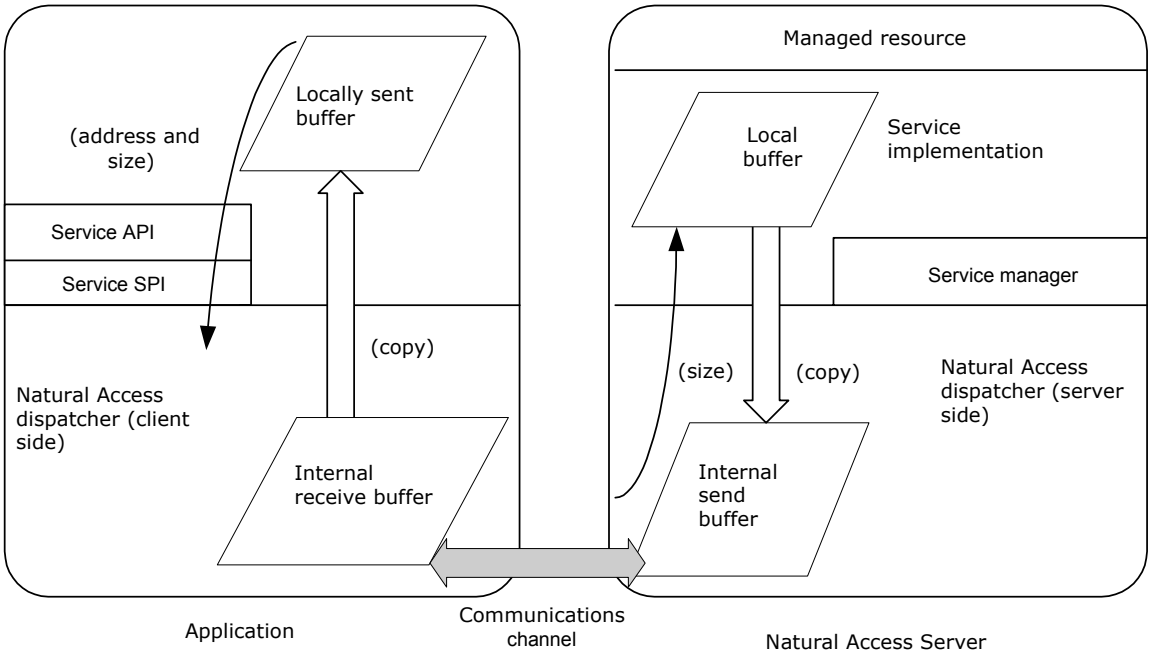
- Function addresses (such as callback function addresses)
- Any local (to the application process) handles such as a file handle
- Pointers to local memory, except in the case of an asynchronous buffer

Buffer arguments in API calls

Service API functions can use both synchronous and asynchronous buffers. All buffers must be allocated on the client side. Buffers cannot be allocated on the server side and directly returned to the client side.

Note: Buffers for events are handled differently. For details, refer to the Buffer management summary section.

Special flags are used within the command structure when data buffers are associated with an API command. These flags are discussed in DISP_COMMAND structure and DISP_EVENT structure. The following illustration provides an overview of buffer management:



Buffer management overview: API calls

Return values

All Natural Access functions return a status code that can be either SUCCESS, CTAERR_***something***, or XXXERR_***something***. Any other information returned from a function should be handled using out arguments.

A return value of SUCCESS for an asynchronous function means that the function request was initiated. The function subsequently may fail and return an error in an event. For a synchronous (short term blocking) function, no events are returned and SUCCESS means that the command completed with no errors.

When defining return codes, reuse existing Natural Access return and error codes as much as possible (for example, CTAERR_***something***).

Event and reason names

Events for a service must be named XXXEVN_***something.***

The event structure contains a value field which can further classify an event by providing additional information (for example, an error code).

To further classify a DONE event, the value field may contain a reason code. Reasons for a service must be named XXX_REASON_***something***. Reasons are typically used to indicate an application error (for example, the network is down). Error codes typically indicate a programming problem.

When defining the service events and reasons, one should attempt to reuse existing Natural Access event and reason codes as much as possible.

Error, reason, event, command, and trace tag codes

Error, reason, event, command, and trace tag codes are constructed using the service ID, a type code (to distinguish between errors, events, commands, or reasons), and a sequence number.

The following table lists the valid range of types as defined in *ctadef.h*:

Description	Sequence range	Type code
Error codes	0x0000 - 0x0FFF	CTA_CODE_ERROR (0x0000)
Reason codes	0x1000 - 0x1FFF	CTA_CODE_REASON (0x1000)
Event codes	0x2000 - 0x2FFF	CTA_CODE_EVENT (0x2000)
Command codes	0x3000 - 0x3FFF	CTA_CODE_COMMAND (0x3000)
Trace tag codes	0x4000 - 0x4FFF	CTA_CODE_TRACETAG (0x4000)

Note: DONE event codes must have the 0x100 bit set. All other event codes must not have this bit set.

Natural Access macros are used to assign an error/reason/event/command/tracetag code. A code is constructed by using the service ID, the type code, and a sequence number as shown:

((***service_id*** << 16) | ***type_code*** | ***sequence***)

For example, assigning an ADI service error code:

```
#define ADIERR_PLAYREC_ACCESS (( ADI_SVCID << 16 ) | CTA_CODE_ERROR | 1 )
```

To facilitate browsing, the defines should be assigned to the resultant hexadecimal number rather than constructed. That is, the *adidef.h* header actually contains the following definition:

```
#define ADIERR_PLAYREC_ACCESS        0x10001
```

where the ADI_SVCID value is 1.

Note: DONE events must have the 0x100 bit set. All other events must not have this bit set. For example, an event with code 0x10002103 is a DONE event. An event with code 0x10002004 is not a DONE event.

Manifest constants and enumerations

The naming convention for enumerations is the same as for manifest constants. Names are in all capital letters. The service acronym is separated from the manifest by an underscore. Underscores should be used freely to separate words to make the manifests and enumeration names more readable. For example:

```
#define CTA_UNITS_INTERNAL  0
#define CTA_UNITS_INTEGER   1

typedef enum {CTA_TRACE_SEVERITY_INFO=0,
              CTA_TRACE_SEVERITY_WARNING,
              CTA_TRACE_SEVERITY_ERROR} CTA_TRACE_SEVERITY;

#define VCE_FILETYPE_AUTO 0
#define VCE_FILETYPE_VOX  1
#define VCE_FILETYPE_WAVE 2
#define VCE_FILETYPE_FLAT 3
```

Structures

Structures should include a size field as the first field. The size of the structure should be passed as an extra argument if the structure is an out argument and the size field of the structure will be filled with the output size. Structure fields must be strongly typed (for example, INT32, DWORD) for compatibility between libraries and applications.

For backward compatibility, structure fields should never be removed, only added.

Each field within a structure must use 4 byte packing (each must begin on a four byte boundary). The total size of the structure must be a multiple of 4 bytes, in case an application uses 1 or 2 byte packing.

Structure names (typedefs) should be all upper case with underscores to improve readability.

Structure field names should be all lower case with no underscores unless needed for readability.

Parameter structures must have fields that can be classified into one of the valid parameter field types (see **ctaGetParmInfo** for a list of valid field descriptors).

Parameter structures

Parameter structures that are arguments to a service API function call must follow the xxx_PARMS naming convention. For example, the ADI function to start playing requires the ADI_PLAY_PARMS data structure.

If the value of the parameter passed with the API function call is NULL, the service implementation uses the context default values.

6 Defining a service

Understanding the managed resource

Before designing a Natural Access service, you must understand exactly what the managed resource is and how it operates. For example, if the managed resource is a telephony board, then you must first understand the low-level details of inter-operating with the telephony board such as how to:

- Initialize the board.
- Send commands to the board.
- Process messages from the board.

You can then define a simple, intuitive Natural Access API to the managed resource. Suggested action items include:

1. API analysis

   Create a list of managed resource interface calls and then determine if any of those calls can be combined into a smaller set of simpler, more abstract functions.

2. Event analysis
   - Itemize the list of all incoming messages (raw events) from the managed resource and determine which of these messages should be processed and forwarded to a client. By definition, the remaining messages (if any) are processed and consumed.
   - Determine whether the managed resource is multiplexed (that is, whether you can use one wait object for events from the managed resource or the managed resource requires multiple wait objects). If the messages sent to and from the managed resource contain addressing information describing the identity of the caller, the managed resource is multiplexed (that is, requires only one wait object).

3. Data abstraction analysis

   When appropriate, create data structures to model the managed resource in software as follows:
   - A data structure that contains information about the managed resource itself.
   - A data structure that contains information pertaining to a client of the managed resource.
   - Zero or more data structures containing information pertaining to entities that can be manipulated by a client on the managed resource.

There are at least two types of information that are useful during this phase:

- A documented interface to the managed resource. This documentation can be used as a baseline for defining the corresponding simpler, more abstract Natural Access service API. Without documentation, the interface must be discerned from the implementation of the managed resource.
- Documented internal behavior of the managed resource, such as state diagrams or interaction diagrams.

In Natural Access parlance, a managed resource is represented and referred to as a service object. The data structure that contains information pertaining to a client of the service object is referred to as an instance of a service object.

tik service

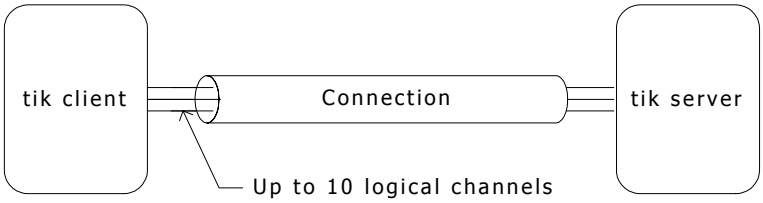
The managed resource for the tik service is a software timer known as the tik server. The tik server operates as a software timer. It waits for a certain interval to expire then generates and sends a text message over a well-defined IPC channel to a client until a maximum number of messages is generated. The client specifies the timer interval and maximum number of messages; the text message is configured at server initialization time and is not dynamically configurable. The tik server can support, at most, 10 separate channels per client.

Commands can be sent from a client to the tik server that:

- Request a connection with the tik server.
- Allocate a logical channel.
- Start timer on a logical channel based upon number of ticks to generate and duration between ticks.
- Stop timer on a logical channel before all tick strings are received.
- Request a channel tear down.

When the tik server generates all tick messages, the logical channel is left active so that another start tick request can be serviced.

If a client sends a stop request string to the server, the server aborts the generation of remaining ticks. The logical channel remains active for further start tick requests.



Using the description of the tik server, the following information summarizes the results of the suggested action items:

Analysis of interaction with tik server

Based on the list of messages to be generated by a tik server client, the abstracted set of API calls are:

- A function to start a ticker
- A function to stop a ticker
- A function to request information concerning the status of the logical connection

Event identification and processing

The following is a list of incoming messages received from the tik server:

- An open channel response message indicating whether a communications channel was established with the server.
- A stop ticking message indicating that a request to abnormally terminate generation of tick strings has been honored.
- A tick message containing the periodic tick string.

The tik server is multiplexed in that all messages to and from the server contain a field denoting the id of the client.

Abstract data structures

The managed resource data structure should contain:

- Communications endpoint for the tik server
- Status fields for communications with the tik server including when a read is pending and how many reads have been performed.
- Storage for the latest message received from the tik server
- Array of logical channels that are active
- The client-specific data structure for the managed resource should contain the:
- Associated Natural Access context handle (the client id)
- Actual logical channel id being used
- State of the logical channel
- Tracemask for per-context tracing

The software timer is the only entity to be manipulated by a tik server client. Similarly, a client can only manipulate one of them at a time. Therefore, it is not necessary to define explicit entity data structures.

Getting a service ID

A numeric service ID is used to internally guarantee uniqueness of services.

To register a new service with NMS Communications, contact Technical Services and provide them with the service's three-letter acronym and a description of the service. Technical Services will verify that the acronym is unique and assign a service ID.

During service development and testing, assign any three-letter acronym and service ID to the service. When the service is ready to be published, register the service with NMS.

The service IDs for services supplied by NMS range from 0 to 0x40 (this includes services such as ADI, VCE, SWI). IDs for services not supplied by NMS typically range from 0x41 to 0xFFF.

tik service

The tik service is registered as:

```
Service name = TIK
Service ID = 0x40
```

Defining the API

The API for a Natural Access service consists of:

- A set of one or more function prototypes (function name, function return, and function arguments).
- A description of each API function.
- A determination of whether the function operates synchronously or asynchronously.

Each API function typically includes either a context handle or a service object handle to identify the unique processing context for the function invocation. Natural Access service APIs should not require a user to pass multiple handles to individual functions. Context handles can be derived from service object handles by using Natural Access dispatcher handle functions. Service object handles are discussed in detail in the Service object handles section.

tik service

Based upon the event analysis (Understanding the managed resource), the following API functions are defined:

- DWORD **tikStartTimer** (CTAHD ***ctahd***, TIK_START_PARMS ****start***)

  Starts the timer function on tik server, asynchronously returns timer ticks. Passes a parameter structure defining the number of ticks to generate and the frequency.

- DWORD **tikStopTimer** (CTAHD ***ctahd***)

  Stops the timer function on tik server, asynchronously returns acknowledgment.

- DWORD **tikGetChannelInfo** (CTAHD ***ctahd***, TIK_CONTEXT_INFO* ***info***, unsigned ***size***)

  Asynchronously gets the operational information from the tik service. Passes a parameter structure to hold the channel data.

Note: The tik service does not create service object handles.

Defining events and reasons

For each asynchronous function defined in the API, define a set of one or more events for subsequent return to the client application. These events typically denote the results of asynchronous API functions invoked by the client application, but they can also represent unsolicited events that were generated by the managed resource.

Each event structure has a value field that can contain additional information or classification for the event. By convention, the value field of a DONE event contains either a reason for completion or an error code. See *Error codes* on page 245 and Events for details.

Set up your service so that it uses existing Natural Access event and reason codes whenever possible. Natural Access events are defined in *ctadef.h.* Natural Access errors and reason codes are defined in *ctaerr.h.*

tik service

Based upon the event analysis (described in *Understanding the managed resource* on page 37), the following event codes and associated reasons are defined:

- TIKEVN_TIMER_TICK

  Event generated by the tik service and sent back to the client for all tick strings.

- TIKEVN_TIMER_DONE

  Event generated by the tik service and sent back to the client to indicate that all tick strings were sent to the client. Can have the following reason codes:

  - CTA_REASON_FINISHED - Indicates that the tik service has successfully generated all requested tick strings at the requested frequency.
  - TIK_REASON_UNKNOWN_SERVER_REASON - indicates an unexpected error has occurred at the tik service.
  - TIK_REASON_TIMER_STOPPED - Indicates **tikStopTimer** was successful.

- TIK_REASON_INVALID_CHANNEL

  Reason code for CTAEVN_DISP_OPEN_SERVICE_DONE. Indicates a failure in initializing communication with the tik service.

- TIK_REASON_CHANNEL_ACTIVE

  Reason code for CTAEVN_DISP_OPEN_SERVICE_DONE. Indicates that the requested channel is occupied.

Defining errors

Each API function must specify a function return. The return codes can be whatever makes sense for the function. Use existing Natural Access error codes as much as possible. For example, generic error codes include CTAERR_OUT_OF_MEMORY, CTAERR_OUT_OF_RESOURCES, and CTAERR_BOARD_ERROR.

Generic error codes are provided in *ctaerr.h*.

tik service

- TIKERR_COMM_FAILURE

  Error communicating with server.
- TIKERR_CHANNEL_NOT_OPENED

  Server connection has not been established.
- TIKERR_OWNER_CONFLICT

  Owner conflict for Natural Access context.
- TIKERR_UNKNOWN_SERVER_RESPONSE

  Unknown server response.
- TIKERR_CAN_NOT_CONNECT

  Cannot establish connection to server.

Defining parameters

Natural Access parameters are a mechanism to allow the operating characteristics of a service to be dynamically modified. If the service being developed needs this type of extensibility, then define a set of one or more parameters. Parameters are named entities with the following syntax:

xxx.category.fieldname

A parameter category is like a directory in a modern file system. It allows multiple logically-related ***fieldnames*** to be nested. For example, the ADI service contains multiple categories, each of which contain multiple ***fieldnames***:

```
ADI.CALLPROG.busycount
ADI.CALLPROG.leakagetime
ADI.CALLPROG.maxbusy

ADI.COLLECT.firsttimeout
ADI.COLLECT.intertimeout
```

A data type and a default value are associated with each parameter. Supported data types are:

Data type	Description
WORD	16 bit unsigned integer
DWORD	32 bit unsigned integer
INT16	16 bit signed integer
INT32	32 bit signed integer
STRING	Arbitrary length string which may contain spaces
BYTE_ARRAY	Array of bytes

tik service

For the tik service, there is one category comprised of two field names:

Number of tick strings to be received = `tik.start.NumTicks`

```
data type = DWORD, default = 2
```

Frequency of tick strings to be received = `tik.start.Frequency`

```
data type = DWORD, default = 2
```

Defining tracing

Natural Access provides a mechanism to log warning, error, and information messages (trace records) to a central location. These trace records are grouped into trace record categories.

A tracemask is used to determine which trace record categories are active (that is, which messages should be logged). Tracemasks are set by the application using **ctaSetTraceLevel** and by *ctdaemon*. See the *Natural Access Developer's Reference Manual* for more information on setting tracemasks.

The Natural Access dispatcher automatically generates trace records for the following trace record categories:

- Service API calls invoked by an application.
- Service SPI calls invoked by another service.
- Events generated by a Natural Access service and destined for the calling application.
- Events generated by a Natural Access service and destined for the calling service.

This topic presents:

- Service-specific trace record categories
- Defining trace tags

Service-specific trace record categories

For service-specific trace record categories, the service must explicitly generate the trace records. There are three service-specific trace record categories explicitly defined by Natural Access. They are:

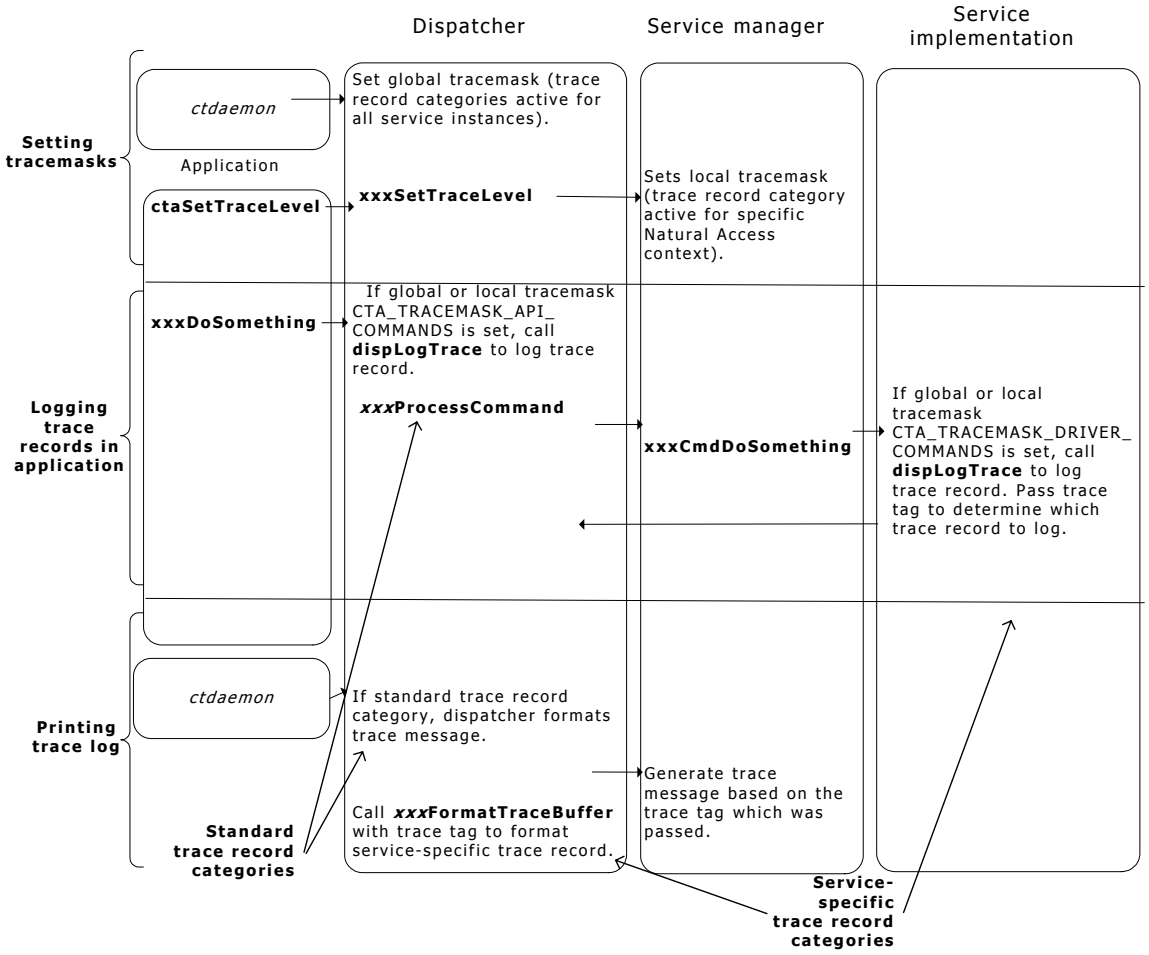
- Commands issued by the service to a low level driver (or another process).
- Events received from the low level driver (or another process).
- Internal service errors.

There are four other service-specific categories (debug categories) that can be used for any remaining trace records that are deemed appropriate for your service and do not fit into the explicitly defined categories.

To support service-specific trace categories, a service must:

- Define trace tags to represent the trace records.
- Check if appropriate tracemasks are set and call the trace logging function (**dispLogTrace**).
- Create binding functions to set local tracemasks (**xxxSetTraceLevel**) and generate trace records (**xxxFormatTraceBuffer**).

As shown in the following illustration, the application determines which trace record categories are logged. The dispatcher handles standard trace record categories (for example, logging service API function calls). Service-specific trace record categories (for example, logging driver calls) must be checked and generated by the service.



Tracing overview

Defining trace tags

For each service, decide which service-specific trace record categories to implement. For each category, define a set of one or more trace tags. The trace tag is used by the service to uniquely identify the trace record.

Although you are not required to define and implement service-specific trace records, it is strongly recommended that you do so. Similarly, it is not required to implement trace records for each of the explicitly defined service-specific trace categories.

tik service

The tik service supports service-specific trace processing. For the tik service, there are no trace records for the internal service error category. Also, there is only one trace record per category.

The specific tracing to be done is:

Category	Associated trace mask	tik-specific trace tags
Driver commands	CTA_TRACEMASK_DRIVER_COMMANDS	TIK_TRACETAG_CMD - Log messages sent from tik service to tik server.
Driver events	CTA_TRACEMASK_DRIVER_EVENTS	TIK_TRACETAG_RSP - Log events received from tik server.
Service errors	CTA_TRACEMASK_SVC_ERRORS	None needed.
Debug 1	CTA_TRACEMASK_DEBUG_BIT0	TIK_TRACETAG_BIT0 - Log internal state transitions.
Debug 2	CTA_TRACEMASK_DEBUG_BIT1	None needed.
Debug 3	CTA_TRACEMASK_DEBUG_BIT2	None needed.
Debug 4	CTA_TRACEMASK_DEBUG_BIT3	None needed.

7 Implementing the API

API implementation overview

Implementing the API for a service includes:

- Creating the header files that contain the API function prototypes, error and event codes, and command code definitions.
- Writing the code that contains the API implementation.
- Declaring parameters in a parameter source (*.pf*) file.

Create the following files when implementing the service API:

File	Description
xxxapi.c	Implements the service API by calling the corresponding SPI function and handling API errors.
xxxdef.h	Contains API prototypes, event and error definitions, and public data structures.
xxxparm.pf	ASCII definition of each parameter and hierarchical structure.
xxxparm.c	Parameter source file automatically generated from *xxxparm.pf*.
xxxparm.h	Parameter header file automatically generated from *xxxparm.pf*.
xxxsys.h	Internal header file containing API command codes.

Creating the header files

When implementing the API, declare the following information in the service's header files:

- API function prototypes
- API function command codes
- Error codes
- Event and reason codes
- Trace tag codes

Defining the API function prototypes

Define each API function prototype in *xxxdef.h.* Include a description of the function and its arguments. API functions must follow the API coding conventions described in *Coding conventions* on page 32.

tik service

<<<< excerpt from *tikdef.h* **>>>>**

```
/*-------------------------------------------------------
Request Tick strings to be generated (i.e., start the ticker).
Operation of this API call is asynchronous.
ctahd: Handle for Natural Access context to start.
start: Timer start parameter structure includes:
* a count field specifying the number of tick strings
* a frequency field specifying how often to send tick strings.
-------------------------------------------------------*/
DWORD NMSAPI tikStartTimer( CTAHD ctahd, TIK_START_PARMS *start );

/*-------------------------------------------------------
Request termination of Tick strings (i.e., stops a running ticker).
Operation of this API call is asynchronous.
ctahd: Handle for Natural Access context.
-------------------------------------------------------*/
DWORD NMSAPI tikStopTimer( CTAHD ctahd );
```

Defining the API function command codes

Assign each service API function a command code in *xxxsys.h*. The command codes are placed in the *.sys* header file because the information is not publicly needed. Construct the command code using the service ID, the CTA_CODE_COMMAND type code, and a sequence number:

((***service_id*** << 16) | CTA_CODE_COMMAND | ***sequence***)

tik service

<<<< excerpt from *tiksys.h* **>>>>**

```
/* tik command codes are constructed using this formula: */
/* (( TIK_SVCID<<16) | CTA_CODE_COMMAND | SEQUENCE ) */

#define TIKCMD_START            0x403000
#define TIKCMD_STOP             0x403001
#define TIKCMD_GET_CHANNEL_INFO 0x403002
```

To facilitate browsing, the defines are typically assigned to the resultant hexadecimal number rather than constructed.

Defining error codes

Define each service API error an error code in *xxxdef.h*. Construct the error code using the service ID, the CTA_CODE_ERROR type code, and a sequence number:

((***service_id*** << 16) | CTA_CODE_ERROR | ***sequence***)

tik service

<<<< excerpt from *tikdef.h* **>>>>**

```
/* tik error codes are constructed using this formula: */
/* (( TIK_SVCID<<16) | CTA_CODE_ERROR | SEQUENCE ) */

TIKERR_COMM_FAILURE            0x400001
TIKERR_CHANNEL_NOT_OPENED      0x400002
TIKERR_OWNER_CONFLICT          0x400003
TIKERR_UNKNOWN_SERVER_RESPONSE 0x400004
TIKERR_CAN_NOT_CONNECT         0x400005
```

Defining event and reason codes

Assign each service API event an event code in *xxxdef.h*. Construct the event code using the service ID, the CTA_CODE_EVENT type code, and a sequence number:

((***service_id*** << 16) | CTA_CODE_EVENT | ***sequence***)

tik service

<<<< excerpt from *tikdef.h* **>>>>**

```
/* tik event codes are constructed using this formula: */
/* (( TIK_SVCID<<16) | CTA_CODE_EVENT | SEQUENCE ) */

TIKEVN_TIMER_TICK 0x402001
TIKEVN_TIMER_DONE 0x402002
```

Assign each reason a reason code in *xxxdef.h*. Construct the reason code using the service ID, the CTA_CODE_REASON type code, and a sequence number:

((***service_id*** << 16) | CTA_CODE_REASON | ***sequence***)

tik service

<<<< excerpt from *tikdef.h* **>>>>**

```
/* tik reason codes are constructed using this formula: */
/* (( TIK_SVCID<<16) | CTA_CODE_REASON | SEQUENCE ) */

TIK_REASON_TIMER_DONE            0x401001
TIK_REASON_INVALID_CHANNEL       0x401002
TIK_REASON_CHANNEL_ACTIVE        0x401003
TIK_REASON_UNKNOWN_SERVER_REASON 0x401004
```

Defining the trace tags

Declare service-specific trace tags in *xxxsys.h*. Construct the trace tag by using the service ID, the CTA_CODE_TRACETAG type code, and a sequence number:

((***service_id*** << 16) | CTA_CODE_TRACETAG | ***sequence***)

tik service

<<<< excerpt from *tiksys.h* **>>>>**

```
/* tik trace tag codes are constructed using this formula: */
/* (( TIK_SVCID<<16) | CTA_CODE_TRACETAG | SEQUENCE ) */

#define TIK_TRACETAG_CMD  0x404001
#define TIK_TRACETAG_RSP  0x404002
#define TIK_TRACETAG_BIT0 0x404003
```

Writing the API function code

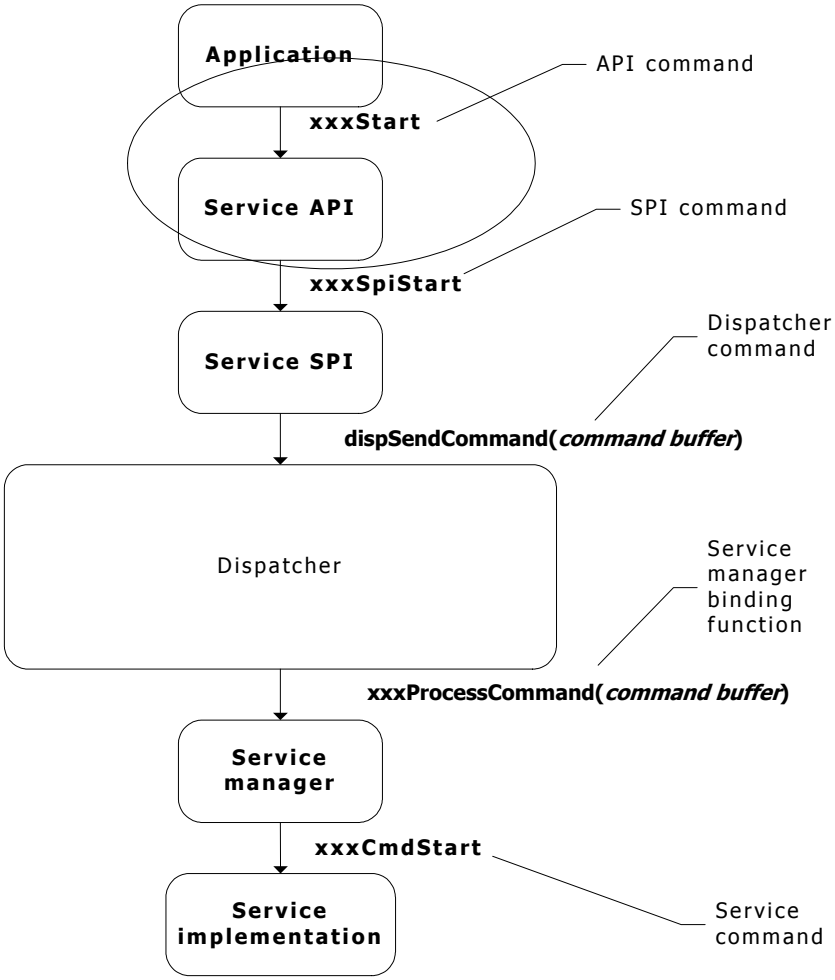
The *xxxapi.c* file contains the service API functions. Each API function calls a corresponding SPI function and logs any errors returned using the Natural Access error macros.

This topic presents:

- Application API command flow
- Calling the SPI function
- Error handling macros
- Example API function

Application API command flow

The following illustration shows the flow of a command initiated by an application API call:



Service API call to SPI function

Calling the SPI function

The signature of an SPI function is typically identical to that of its associated API function with one exception. The SPI function contains an additional parameter to indicate the invoker of the SPI function (an application or another service). Since an API function is only invoked by an application, each API function calls the SPI function using CTA_APP_SVCID.

Error handling macros

To ensure consistent error handling for all Natural Access services, the service API function should use the Natural Access error macros for reporting API errors. The Natural Access error macros expand to calls to **dispApiError**.

The macro CTAAPIERROR is called with an error code and the API function name:

```
#define CTAAPIERROR (ctahd, err) dispApiError((ctahd), (err),\
                     funcname,  NULL)
```

CTAAPIERRORINFO also includes an additional information string to further qualify the error:

```
#define CTAAPIERRORINFO (ctahd, err,info)dispApiError((ctahd),\
                        (err),  funcname, info)
```

The Natural Access error macros must be called by API functions when an error is returned so that the dispatcher can invoke the application registered error handler and log a trace message to the *ctdaemon*.

To use the error macros, the service API function must include a call to the macro CTABEGIN with the function name. This macro defines a local variable ***funcname*** that stores the function name that is passed to **dispApiError** when the other error macros are invoked.

Example API function

tik service

<<<< excerpt from *tikapi.c* **>>>>**

```
/*-------------------------------------------------------------
tikStartTimer
- tik service API function to start timer function on tik server
-------------------------------------------------------------*/
DWORD NMSAPI tikStartTimer( CTAHD ctahd, TIK_START_PARMS *start)
{
DWORD ret ;

/* Needed by CTAAPIERROR */
CTABEGIN( "tikStart" ) ;

/* Call corresponding SPI function with CTA_APP_SVCID */
ret = tikSpiCmdStartTimer( ctahd, start, CTA_APP_SVCID ) ;

/* Log error on failure */
if ret != SUCCESS )
{
CTAAPIERROR( ctahd, ret ) ;
}
return ret ;
}
```

Declaring parameters

The following files contain the service parameters:

File	Description
xxxparm.pf	ASCII definition of each parameter and hierarchical structure.
xxxparm.c	Parameter source file automatically generated from *xxxparm.pf* during the make process.
xxxparm.h	Parameter header file automatically generated from *xxxparm.pf* during the make process.

This topic presents:

- Creating the parameter definition file
- Generating the parameter files

Creating the parameter definition file

The parameter definition file (*.pf* file) specifies the service name, service ID, the parameter category, and the parameter name. The following table provides an overview of the *.pf* file. Refer to the *Natural Access Developer's Reference Manual* for more details on constructing and using a *.pf* file.

Field	Description
SvcName	Specifies the service name.
SvcID	Specifies the service ID.
Category	Specifies the name of the category being declared. Terminate a category block with the keyword End. `Category` ***`category_name`*** ***`field_descriptor`*** `or` ***`substructure`*** . . . `End`
SubStructure	Specifies the name of the substructure being declared. Terminate a substructure block with the keyword End. `SubStructure` ***`substructure_name`*** ***`field_descriptor`*** . . . `End`

Field	Description
FieldDescriptor	Declare a field descriptor within a Category or SubStructure block. A FieldDescriptor specification does not begin with a keyword. It begins with the name of the field itself, as follows: ***name***, ***type***, ***unit***, ***default_value*** • ***name***: A field name must be unique within its scope (for example, there can only be one busycount field within the CALLPROG category. However, there could also be a busycount field in the COLLECT category. A field name is not case sensitive (for example, CALLPROG.busycount and CALLPROG.busyCount are duplicates). • ***type***: A field descriptor must contain a data type. The supported data types are: WORD, DWORD, INT16, INT32, STRING, BYTE_ARRAY • ***unit***: A field descriptor must also contain a units specification. Supported units are: INTEGER, COUNT, MASK, HZ, MS, DB, DBM, IDU, PERCENT The STRING units specification must be used when the field data type is a string. When declaring a STRING, you must also declare a maximum string size (in bytes). The declared maximum size must be sufficient to store the default string. • ***default_value***: A field descriptor must contain a default value. String values must be surrounded by double quotes. Integer values may either be entered in hexadecimal or decimal format. For example: `NumTicks, DWORD, COUNT, 2`

tik service

<<<< excerpt from *tikparm.pf* **>>>>**

```
#-------------------------------------------------------------
# NAME: tikparm.pf
# This file defines (ASCII format) the TIK Service parameters
# that are accessible through the Natural Access API parameter functions.
# This file is used by the pf_tosrc conversion program to
$ generate the following 'C' files: tikparm.h and tikparm.c
#-------------------------------------------------------------
SvcName  TIK  # TIK Service Name
SvcId    0x40 # TIK Service ID

# Only one category (two fields) is required by the tik service;
# The TIK parameters (usable through Natural Access) are:
# tik.start.NumTicks
# tik.start.Frequency

Category             Start
    NumTicks, DWORD, COUNT, 2 # number of tick strings rcvd
    Frequency, DWORD, COUNT, 2 # frequency of tick strings rcvd
End
```

Generating the parameter files

Use the utility *pf2src* to automatically generate the parameter files *xxxparm.c* and *xxxparm.h* from the parameter file (*.pf*).

To generate the source file for the *.pf* file, enter:

```
pf2src pf_file
```

pf_file is the *.pf* input file.

The generated header file *xxxparm.h* defines the service ID assigned to the service and the API level parameter data structure on a per category basis. Each parameter structure uses the naming convention _PARMS as described in *Parameter structures* on page 36.

tik service

<<<< excerpt from *tikparm.h* **>>>>**

```
/***************************************************
* Auto Generated Source Module.
* Manual edits will be lost when file is regenerated!!
* UDP Compiler ( Version 1.1 )
***************************************************/
#ifndef TIKPARMPARM_DEFINED
#define TIKPARMPARM_DEFINED

#include "ctadef.h"

#ifdef __cplusplus
extern "C"
{
#endif

#define TIK_SVCID 0x40

#define TIK_START_PARMID 0x400001

typedef struct
{
    DWORD size;
    DWORD NumTicks;
    DWORD Frequency;
} TIK_START_PARMS;

extern const CTAPARM_DESC * const _tikParmDescTable[];


#ifdef __cplusplus
}
#endif

#endif
```

In the generated *xxxparm.c* file, the parameter description tables are declared and a pointer to the tables is created. You must include the pointer to the parameter descriptor table in the call to **dispRegisterService** during service initialization (see **xxxDefineService**).

tik service

<<<< excerpt from *tikparm.c* **>>>>**

```
/*************************************************
* Auto Generated Source Module.
* Manual edits will be lost when file is regenerated!!
* UDP Compiler ( Version 1.1 )
*************************************************/
#include <nmstypes.h>
#include <ctadisp.h>
#include <tikparm.h>
TIK_START_PARMS _tikStartDefaults =
{
    12,
    2,
    2,
};
#define _F_(f,s,u) CTAFIELDDESC(TIK_START_PARMS,f,s,u)
const CTAPARM_FIELD _tikStartStructDef[] =
{
    "START", 0, 12 , 0, 0,
    _F_( NumTicks, DWORD, COUNT ),
    _F_( Frequency, DWORD, MS ),
};
#undef _F_
CTAPARM_DESC _tikStartParmDescr =
{
    TIK_START_PARMID,
    &_tikStartDefaults,
    _tikStartStructDef,
    CTAPARM_FIELDCNT(_tikStartStructDef)
};
const CTAPARM_DESC * const _tikParmDescTable[] =
{
&_tikStartParmDescr,
NULL
};
```

8 Implementing the SPI

SPI implementation overview

Implementing the SPI includes:

- Declaring SPI function prototypes.
- Implementing the SPI functions.
- Implementing the client-side binding functions.

Create or modify the following files when implementing the service SPI:

File	Description
xxxspi.c	Implements the service SPI functions.
xxxspi.h	Contains SPI prototypes.

Declaring SPI function prototypes

The service header file, *xxxspi.h*, contains the function prototype for each SPI function.

The signature of an SPI function is typically almost identical to that of its associated API function, except the SPI function contains an additional parameter to indicate the invoker of the SPI function. This value may be a service ID (if the SPI function is called by another service) or CTA_APP_SVCID (if the SPI function is called by an API function).

tik service

<<<< excerpt from *tikspi.h* **>>>>**

```
/*-------------------------------------------------------------
SPI function for tikStartTimer API.
-------------------------------------------------------------*/
DWORD NMSAPI tikSpiCmdStartTimer( CTAHD ctahd,
                          TIK_START_PARMS *start,
                          WORD source) ;
```

Implementing the SPI functions

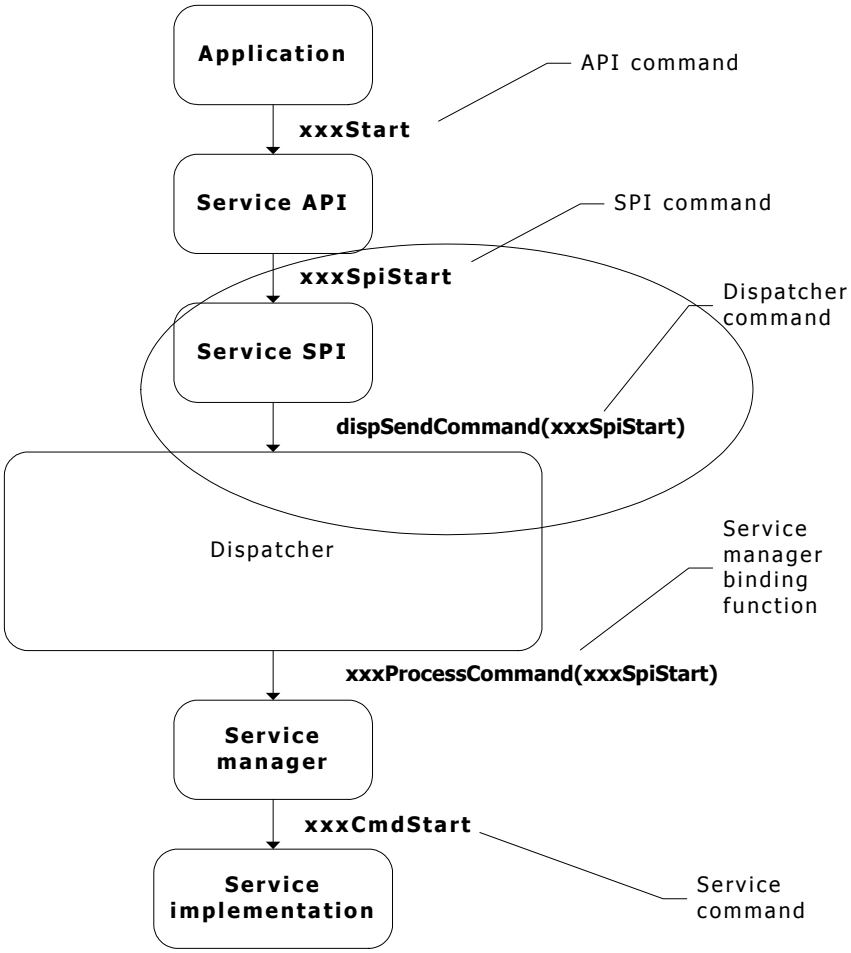
Each service API function has a corresponding SPI function in the *xxxspi.c* file. The SPI function marshals the parameters into an SPI command message structure (DISP_COMMAND) and calls **dispSendCommand** to send the command to the service implementation. When the reply is received, the SPI function unmarshals the results (if any).

This topic presents:

- Service SPI command flow
- DISP_COMMAND structure
- DONE events associated with asynchronous API functions
- Example SPI function

Service SPI command flow

The following illustration shows the service SPI command flow:



Service SPI command flow

DISP_COMMAND structure

The SPI function marshals the parameters into the command message structures (DISP_COMMAND). The DISP_COMMAND structure contains the following fields:

```
typedef struct
{
  DWORD id;           /* Command ID (and dest. service ID)           */
  CTAHD ctahd;        /* Natural Access context handle               */
  DWORD size1;        /* If dataptr1 !=NULL => size1 is buffer       */
                      /*   size- else message specific data          */
  void *dataptr1;     /* Data pointer                                */
  DWORD size2;        /* If dataptr2 !=NULL => size2 is buffer       */
                      /*   size- else message specific data          */
  void *dataptr2;     /* Data pointer                                */
  DWORD size3;        /* If dataptr3 !=NULL => size3 is buffer       */
                      /*   size- else message specific data          */
  void *dataptr3;     /* Data pointer                                */
  DWORD objHd;        /* Service object handle (formerly "reserved") */
  MESSAGE_ADDR addr; /* Source/Destination service ID               */
} DISP_COMMAND;

typedef struct
{
  WORD source          /* Source service ID                          */
  WORD destination     /* Destination service ID                     */
} MESSAGE_ADDR
```

To marshal the parameters into the DISP_COMMAND structure, set the fields in the structure as follows:

Field	Description
id	Service's command identifier for the particular API function.
ctahd	Natural Access context handle. This handle can be passed directly to the API function. It can also be derived from a client-side service object handle by calling **dispGetHandleValue**.
objhd	If a service object handle is being used, set this field to the server-side service object handle. Otherwise, set it to NULL_CTAHD. (Refer to the Service object handles section for more information about service object handles.)
The remaining fields	Use to pass function specific arguments to the service manager.

Three non-pointer value arguments can be packed directly into a message using the DISP_COMMAND fields size1, size2, and size3, and then setting their corresponding data buffer pointer fields (dataptr1, dataptr2, and dataptr3) to NULL.

If a size***x*** field's associated data pointer is NULL, then the size***x*** field in the command structure is automatically treated as an ***inout*** parameter between the service interface and the service manager. That is, size***x*** fields in the DISP_COMMAND structure are transferred in both directions by the dispatcher if their associated data pointer is NULL. For example:

```
xxxSpiDoSomething( CTAHD ctahd, DWORD one, DWORD two,
                   DWORD three, DWORD *retData,
                   WORD source )
{
    DISP_COMMAND cmd = {0};
    DWORD  ret;

    cmd.size1 = one;
    cmd.dataptr1 = NULL;
    cmd.size2 = two;
    cmd.dataptr2 = NULL;
    cmd.size3 = three;
    cmd.dataptr3 = NULL;

    /* etc. */
    ret = dispSendCommand( &cmd );
    if ( ret == SUCCESS )
        /* Service manager returns data in size3 field */
        *retData = cmd.size3;
}
```

It is invalid to pass a value in one of the dataptr***x*** fields with a zero in the corresponding size***x*** field. In this situation, **dispSendCommand** does not return an immediate error, though ultimate processing of the command message by the service implementation will likely fail.

If an API function has more than three non-pointer value arguments, then an internal data structure (a transport structure) must be created to hold multiple arguments. This is then passed using one of the DISP_COMMAND data buffer pointers. For example:

```
xxxSpiDoSomething( CTAHD ctahd,
                   DWORD one, DWORD two, DWORD three, DWORD four,
                   WORD source )
{
    DISP_COMMAND cmd = {0};
    MY_PARAMS   *pparms;

    cmd.size1 = one;
    cmd.dataptr1 = NULL;
    cmd.size2 = two;
    cmd.dataptr2 = NULL;

    /* Bundle multiple arguments into a transport buffer */
    pparms = malloc( sizeof(MY_PARMS) );
    pparms->value1 = three;
    pparms->value2 = four;

    cmd.size3 = sizeof(MY_PARMS);
    cmd.dataptr3 = (void *) pparms;

    /* etc */

    dispSendCommand( &cmd );

    /* Free this synchronous input buffer. */
    free( pparms );
}
```

If you are passing API function arguments that are pointer values, the distributed environment in which these services run requires that Natural Access be informed of how these pointers are being used. This is needed for Natural Access' own internal buffer management (see the Buffer management summary for more information on Natural Access buffer management).

A function argument that is a pointer can be an address of one of four types of buffers:

API buffer type	Example use
Synchronous input	Service interface passes a buffer of data to the service manager.
Synchronous output	Service interface passes a pointer to a buffer that the service manager synchronously fills with data.
Asynchronous input	Service interface passes a pointer to a buffer of data for the service implementation to operate on (for example, voice data to play). The service implementation later generates an event to inform both the application and Natural Access that its internal transport buffer can be freed.
Asynchronous output	Service interface passes a pointer to a buffer that the service manager fills after the API function returns. The service implementation later generates an event to inform the caller that the buffer has been filled.

The service informs Natural Access of how the pointers are being used by setting flags in the three most significant bits of the buffer pointer's related 32-bit size***x*** field, limiting the maximum allowed value in the field. The following table indicates the flag to set for each buffer type:

API buffer type	Flag
Synchronous input	None (default)
Synchronous output	CTA_VIRTUAL_BUF
Asynchronous input	CTA_ASYNC_BUF_IN
Asynchronous output	CTA_ASYNC_BUF_OUT

For example:

```
MY_BUFFER *pbuff;
DISP_COMMAND cmd;

pbuff = malloc( sizeof(MY_BUFFER) );
cmd.size1 = sizeof(MY_BUFFER) | CTA_VIRTUAL_BUF;
cmd.dataptr1 = (void *) pbuff;
```

Implementing client-side binding functions shows the use of the client side binding function **xxxEventProcessing** for the subsequent asynchronous release of this buffer.

Natural Access provides dispatcher functions, **dispSetAsyncIn** and **dispSetAsyncOut**, for setting these asynchronous flags. Because of the Natural Access dispatcher's internal logic, a service API function cannot have both an asynchronous input buffer and an asynchronous output buffer.

Note: For reasons of efficiency, the content of DISP_COMMAND messages are currently not converted to network byte order. This affects the interoperability between computers of different byte ordering (that is, Big-endian versus Little-endian).

DONE events associated with asynchronous API functions

Parameters to certain service API functions may be asynchronous input buffers. That is, although the function returns immediately, the service implementation still considers an input buffer in use until it generates an associated service DONE event (XXXEVN_SOMETHING_DONE).

Due to the client-server architecture, the service implementation requires that this DONE event contain a flag in the size field that allows the underlying Natural Access transfer buffers to be freed. Even though a buffer of data is not sent in the DONE event, the value of the buffer field in the event structure must be set to the same value originally sent in the DISP_COMMAND message as the asynchronous input buffer.

```
xxxCmdDoSomething( void *mgrcontext, DISP_COMMAND *cmd )
{
     // Assume dataptr1 is an asynchronous input buffer.
     // Save input buffer pointer for later
     mgrcontext->cmd.bufferSize  = cmd.size1;
     mgrcontext->cmd.inputBuffer = cmd.dataptr1;
     // etc.
}
------------------------------ Later, when operation is complete
xxxFetchAndProcess( CTAQUEUEHD qhd, CTA_WAITOBJ *waitobj, void *arg;
{
     XXXMGRCONTEXT *mgrcontext = (XXXMGRCONTEXT *) arg;
     CTA_EVENT evt = {0};
     if( mgrcontext->cmd.inputBuffer )
     {
          evt.size = mgrcontext->cmd.bufferSize | CTA_EVT_ASYNC_BUF_IN;
          ev->buffer = mgrcontext->cmd.inputBuffer;
     }
     // etc.
     dispQueueEvent( &evt );
}
```

Note: No special flag needs to be set in the size field when the event's buffer is an asynchronous output buffer.

Example SPI function

tik service

<<<< excerpt from *tikspi.c* **>>>>**

```
DWORD NMSAPI tikSpiStartTimer( CTAHD ctahd,
TIK_START_PARMS *start,
WORD source )
{
    void *dataptr1;
    DWORD size1;
    DISP_COMMAND m;
    if ( start != NULL )
    {
        /* Use application supplied TIK parameters. */
        size1 = start->size;
        dataptr1 = (void *)start;
    }
    else
    {
        /* Use TIK parameter default values stored in Natural Access.*/
        size1 = 0;
        dataptr1 = NULL;
    }
    /* Marshal function id, function arguments, destination
    * service, and source identifier into message buffer. */
    m.id = TIKCMD_START;
    m.ctahd = ctahd;
    m.size1 = size1;
    m.dataptr1 = (dataptr1);
    m.size2 = 0;
    m.dataptr2 = NULL;
    m.size3 = 0;
    m.dataptr3 = NULL;
    m.reserved = 0;
    m.addr.destination = TIK_SVCID;
    m.addr.source = source;

    /* Send command to tik service via Natural Access dispatcher*/
    return dispSendCommand(& m);
}
```

Implementing client-side binding functions

Client-side binding functions are called by the Natural Access dispatcher rather than by the application. These functions provide adjunct functionality. The names of client-side binding functions are standardized and known to the Natural Access dispatcher. The dispatcher determines at runtime whether particular binding functions are implemented by the service interface and calls them appropriately.

Currently, there is only one client-side binding function: **xxxEventProcessing**. The Natural Access dispatcher calls this synchronous function prior to posting a service event to the client's application queue. (***xxx*** is the service acronym; for example: **tikEventProcessing**). Natural Access calls this function to enable the service interface to make updates to an event generated by the service implementation before the event is received by the client application. This function is often used in the processing of service object handles as described in the Service object handles section. The function is also used to clear and release event buffers.

The following sample code shows how **xxxEventProcessing** is used:

```
xxxSpiGetInfo( CTAHD ctahd )
{
    MY_BUFFER *pbuff;
    DISP_COMMAND cmd = {0};

    cmd.id = XXXCMD_GETINFO;

    pbuff = malloc( sizeof(MY_BUFFER) );
    cmd.size1 = sizeof(MY_BUFFER) | CTA_VIRTUAL_BUF;
    cmd.dataptr1 = (void *) pbuff;

    /* etc. */
    dispSendCommand( &cmd );

    /* Data will be returned in asynchronous DONE event */
    return SUCCESS;
}

/* Later, when the service manager generates a XXXEVN_GETINFO_DONE event. */

xxxEventProcessing( CTA_EVENT *event )
{
    /* Must check event's buffer flags just as an
    *  application is required to do.
    */
    bool bCtaOwnsBuffer = pevent->size & CTA_INTERNAL_BUFFER;
    /* Clear any flags in the size field before using it. */
    pevent->size &= ~CTA_INTERNAL_BUFFER;

    if ( event->id == XXXEVN_GETINFO_DONE )
    {
        /* Save information to global area and release the buffer pointer
        *  that was allocated by the SPI function.
        */
        memcpy( globalBuff, event->buffer, event->size );

        if ( bCtaOwnsBuffer )
        {
             /* This buffer is owned by Natural Access. This event
             *  is being sent on a shared context.
             */
            dispFreeBuffer( event->buffer );
        }
        else
        {
            /* The SPI function invoked by this client allocated this
            *  buffer. Release it.
            */
            free( event->buffer );
        }

        /* Clear the buffer pointer so that it is not used by the
        *  application when it receives this DONE event.
        */
        event->size = 0;
        event->buffer = NULL;
    }
}
```

9 Implementing the service manager

Service manager overview

The service manager provides the set of binding functions that connects the service implementation to the Natural Access infrastructure.

The dispatcher calls binding functions. Each service has a unique set of binding functions. Binding functions are named ***xxxFunctionName***, where ***xxx*** is the service acronym.

A service must implement a subset of the binding functions. For example, each service must implement the binding function to process API commands (**xxxProcessCommand**). Optional binding functions need only be implemented if required by the service (for example, **xxxSetTraceLevel**).

xxxInitializeManager is known by the dispatcher. The remaining binding function names are registered with the dispatcher during service initialization.

Create or modify the following files when implementing the service manager binding functions:

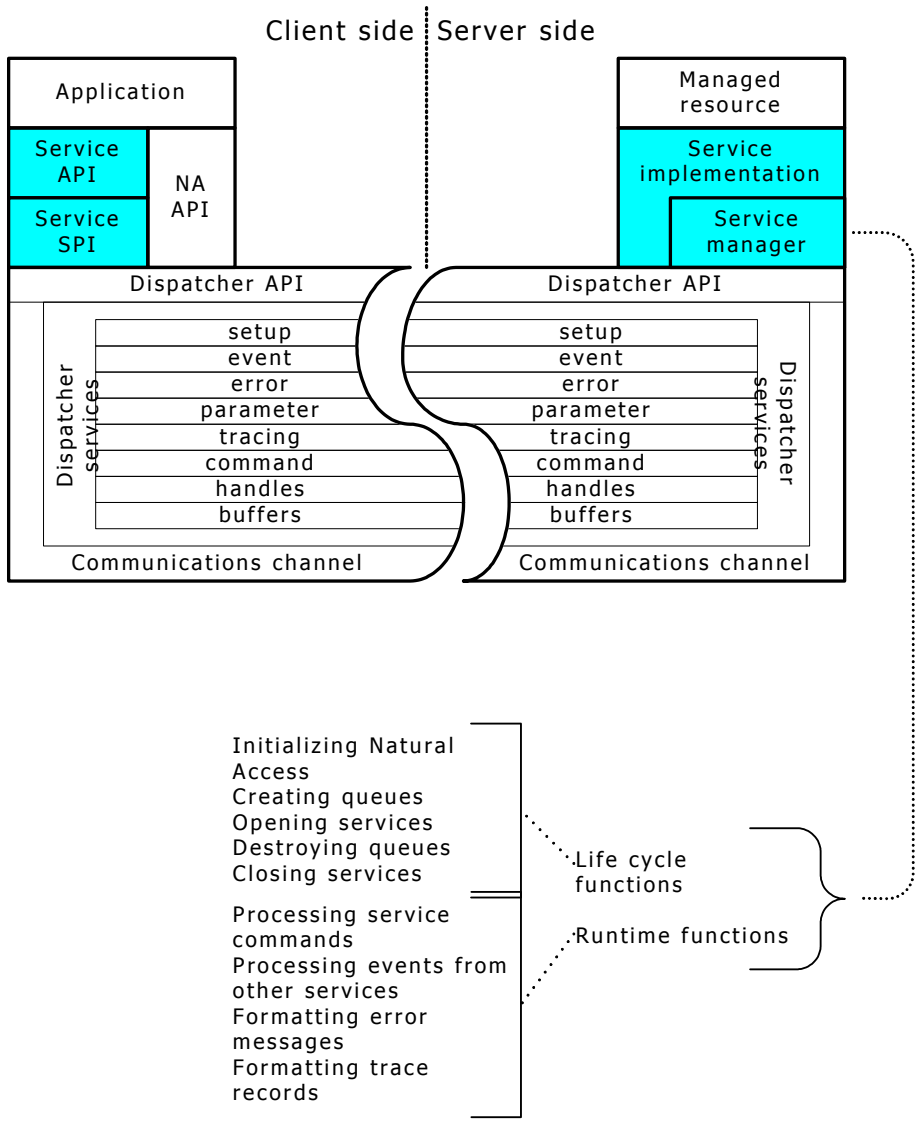
File	Description
xxxbnd.c	Contains the service manager binding functions.
xxxdef.h	Contains version IDs and compatibility levels of the service.
xxxsys.h	Contains trace tag codes.

Binding functions

The binding functions in the service manager are divided into two groups:

Function type	Description
Life cycle binding functions	Called during service startup and shutdown. These binding functions initialize the service and handle any service specific operations required when creating and destroying Natural Access queues and contexts.
Runtime binding functions	Called while the application is running. These functions include processing API commands and handling errors and events.

The following illustration shows the binding functions:



Service manager binding functions

Life cycle binding functions

The life cycle binding functions are the service specific functions called by Natural Access to perform:

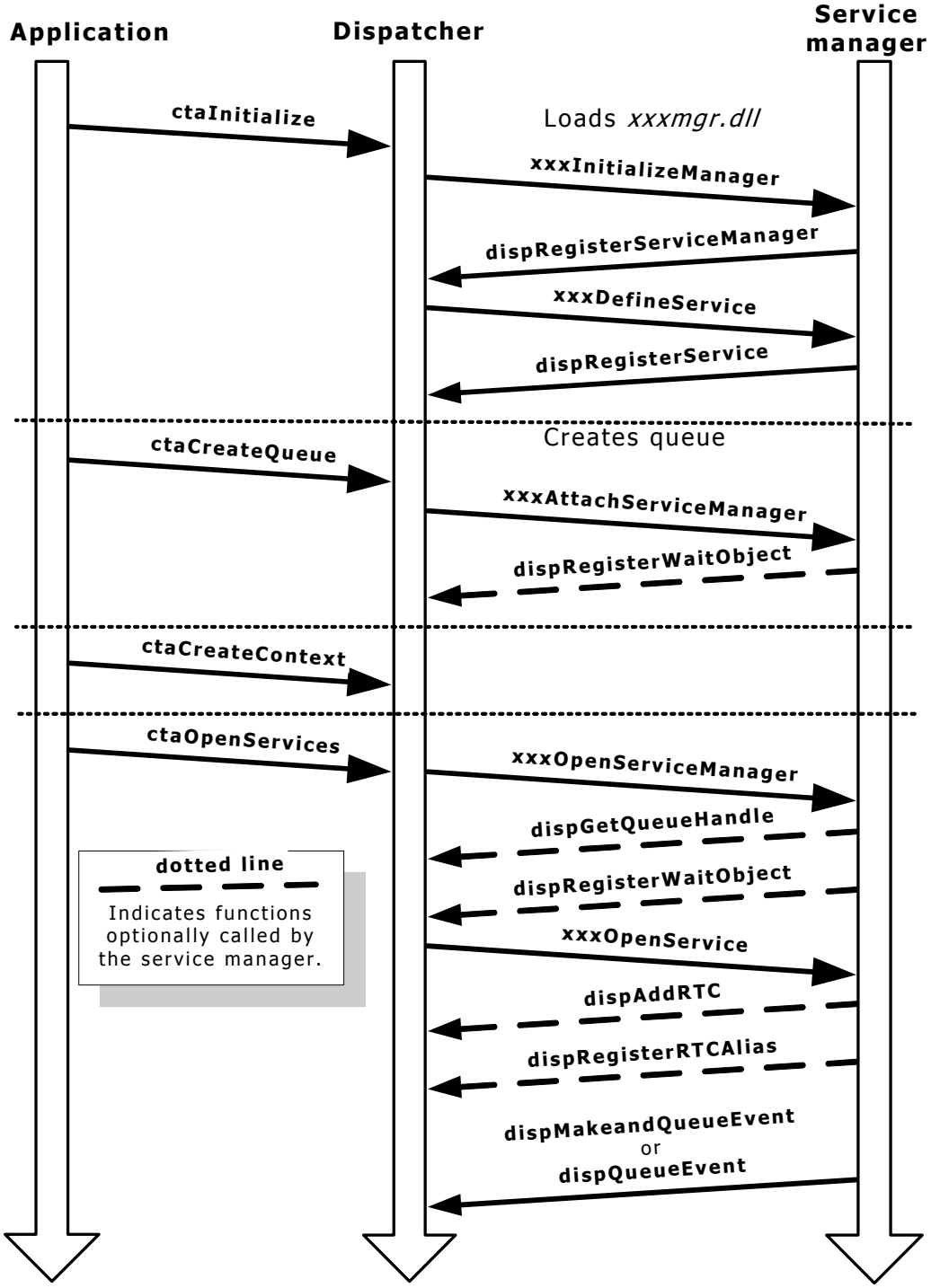
- Service registration
- Event handling initialization and shutdown
- Service startup
- Service shutdown

The dispatcher invokes the life cycle binding functions in response to Natural Access application calls. The life cycle binding functions can return SUCCESS, SUCCESS_RESPONSE, or an error.

Note: The service writer is not required to implement all life cycle binding functions.

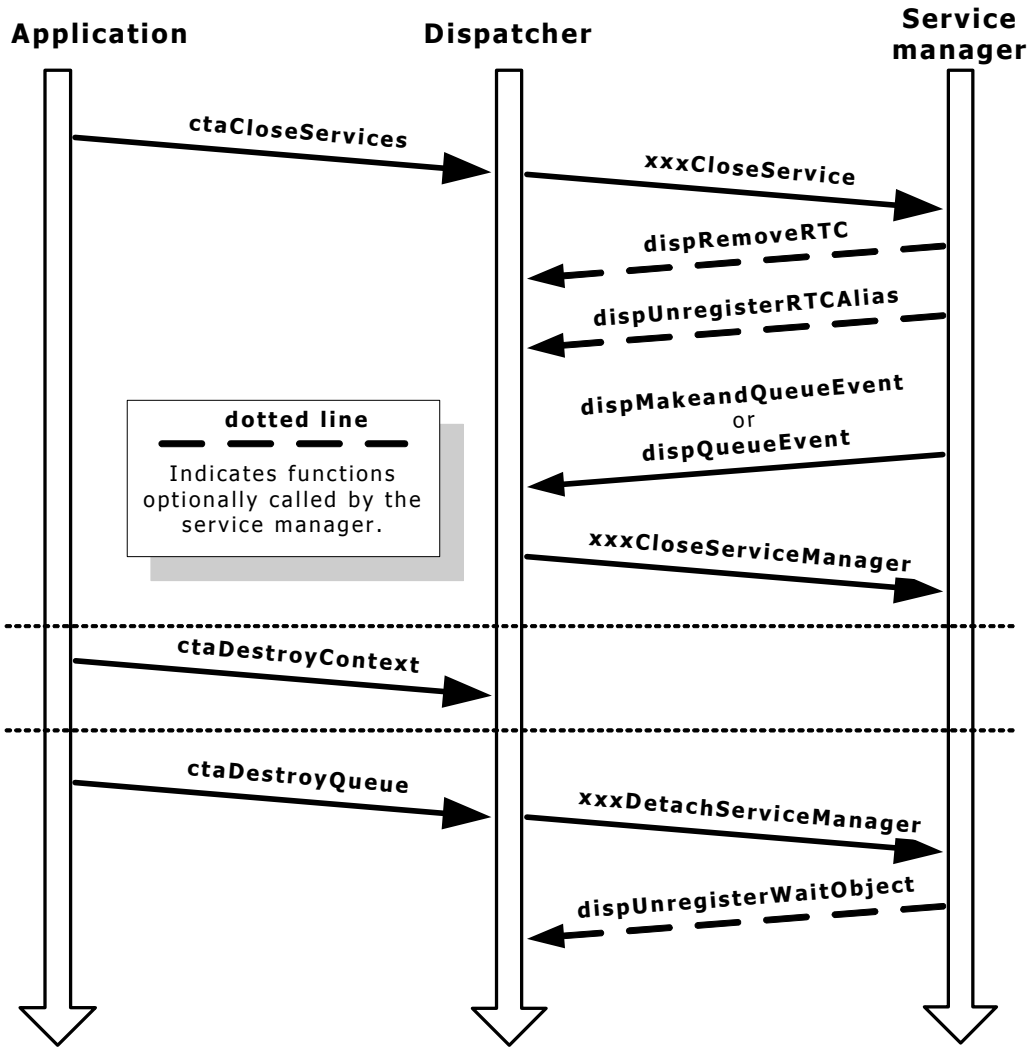
Most life cycle binding functions are synchronous functions with two exceptions: **xxxOpenService**, which initializes the managed resource by requesting resources to support a client, and **xxxCloseService**, which releases resources. These functions provide a return code indicating initialization success or failure. The actual results of the functions return later as events.

The following illustration shows the binding functions called in response to Natural Access application startup:



Life cycle binding functions -- startup

The following illustration shows the binding functions called in response to Natural Access application shutdown:



Life cycle binding functions -- shutdown

The following tables list the life cycle binding functions and give an overview of each function including the key dispatcher functions and data structures:

Life cycle binding functions: registration

xxxInitializeManager

Description:	• Acts as an entry point into service manager for service manager and service registration. • Declares version numbers for service manager. • Declares compatibility levels for service manager. • Declares service manager binding functions. • Acquires global tracemask pointer managed by *ctdaemon* (if active).
Behavior:	Synchronous
Dispatcher functions called:	**dispRegisterServiceManager** **dispGetTracePointer**
Important data structure:	CTAINTREV
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaInitialize**
Required?	Must be implemented.

xxxDefineService

Description:	• Declares version numbers for service. • Declares compatibility levels for service. • Declares needed API compatibility levels of related services. • Declares the parameter descriptor tables.
Behavior:	Synchronous
Dispatcher functions called:	**dispRegisterService**
Important data structure:	CTAINTREV CTAPARM_DESC CTAREQ_SVC
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaInitialize**
Required?	Must be implemented.

Life cycle binding functions: event handling

xxxAttachServiceManager

Description:	• Declares wait objects (if interacting with a multiplexed managed resource). • Initializes communications with managed resource. • Allocates ***queuecontext*** object for processing incoming events.
Behavior:	Synchronous
Dispatcher functions called:	**dispRegisterWaitObject**
Important data structure:	Managed resource specific ***queuecontext.***
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaCreateQueue**
Required?	Optional, required if managed resource is multiplexed.

xxxDetachServiceManager

Description:	Deallocates all data structures allocated with **xxxAttachServiceManager.**
Behavior:	Synchronous
Dispatcher functions called:	**dispUnregisterWaitObject**
Important data structure:	Managed resource specific ***queuecontext***.
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaDestroyQueue**
Required?	Must be implemented if **xxxAttachServiceManager** is implemented.

Life cycle binding functions: service startup

xxxOpenServiceManager

Description:	• Allocates ***mgrcontext*** object, which contains data pertaining to the managed resource but is specific to a context. • Declares wait objects (if interacting with a non-multiplexed managed resource).
Behavior:	Synchronous
Dispatcher functions called:	**dispRegisterWaitObject** **dispGetQueueHandle**
Important data structure:	Managed resource specific ***mgrcontext***.
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaOpenServices**
Required?	Must be implemented.

xxxOpenService

Description:	Initializes the managed resource by requesting resources to support a client (for example, a context).
Behavior:	Asynchronous
Dispatcher functions called:	**dispQueueEvent** or **dispMakeAndQueueEvent** **dispAddRTC** (optional) **dispRegisterRTCAlias** (optional)
Important data structures:	CTA_MVIP_ADDR CTA_SERVICE_ARGS
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaOpenServices**
Required?	Optional

xxxAddRTC

Description:	Actions taken by the service when this function is called is at the discretion of the service writer.
Behavior:	Synchronous
Dispatcher functions called:	Service specific
Important data structure:	Service specific
Returns:	SUCCESS
Initiated through the Natural Access dispatcher:	**dispAddRTC**
Required?	Optional

Life cycle binding functions: service shutdown

xxxCloseServiceManager

Description:	Deallocates all data structures allocated with **xxxOpenServiceManager.**
Behavior:	Synchronous
Dispatcher functions called:	**dispUnregisterWaitObject**
Important data structure:	Managed resource specific ***mgrcontext***.
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaCloseServices**
Required?	Must be implemented.

xxxCloseService

Description:	Releases resources from managed resource for that context.
Behavior:	Asynchronous
Dispatcher functions called:	**dispQueueEvent** or **dispMakeAndQueueEvent** **dispRemoveRTC** (optional) **dispUnregisterRTCAlias** (optional)
Important data structure:	None
Returns:	SUCCESS or an error.
Initiated through Natural Access function call:	**ctaCloseServices**
Required?	Must be implemented if **xxxOpenService** is implemented.

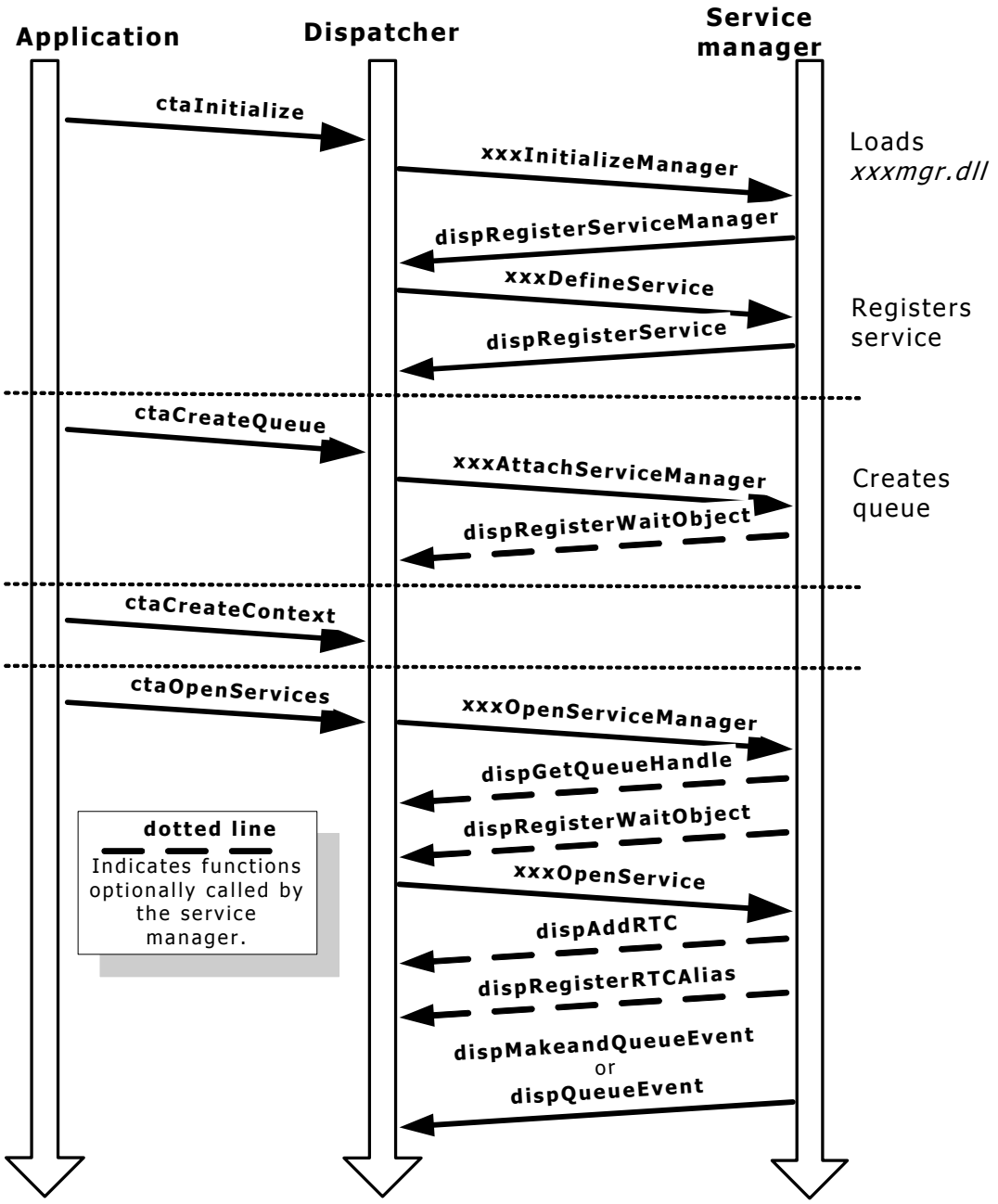
xxxRemoveRTC

Description:	Actions taken by the service when this function is called is at the discretion of the service writer.
Behavior:	Synchronous
Dispatcher functions called:	Service specific
Important data structure:	Service specific
Returns:	SUCCESS
Initiated through Natural Access dispatcher function:	**dispRemoveRTC**
Required?	Optional

Registering services

Service registration occurs when the application calls **ctaInitialize**. Two binding functions are called during initialization:

- **xxxInitializeManager**
- **xxxDefineService**



Service registration

ctaInitialize can only be called once per process; therefore the call will be thread-safe. **xxxInitializeManager** and **xxxDefineService** do not need to be coded to be reentrant since **ctaInitialize** can only be called once per process (and only from one thread within that process).

If the service manager is not loaded, **ctaInitialize** loads it into memory. Each service manager must reside in its own DLL or library that conforms to the naming convention *xxxmgr.dll* (for example, *tikmgr.dll*) or *libxxxmgr.so* in UNIX.

The dispatcher then calls the service manager entry point **xxxInitializeManager**.

xxxInitializeManager

xxxInitializeManager is the only required exported function from a service manager DLL or shared library. **xxxInitializeManager** performs the following functions:

- Declares service manager binding functions
- Declares version numbers and compatibility levels for service manager
- Acquires the global tracemask pointer

Registering binding functions

The binding function names are registered with the dispatcher during service initialization. This registration makes the binding functions of the service manager available to the dispatcher to be invoked when needed. Binding functions are declared in the CTASVCMGR_FNS function pointer table. A NULL entry in the table indicates that the default implementation for that function should be used (that is, no service-specific implementation of the binding function is required). The following table lists the default implementation of each binding function:

Binding function	Default implementation
xxxDefineService	Return SUCCESS
xxxAttachServiceManager	Return SUCCESS
xxxDetachServiceManager	Return SUCCESS
xxxOpenServiceManager	Return SUCCESS
xxxCloseServiceManager	Return SUCCESS
xxxOpenService	Enqueue CTAEVN_DISP_OPEN_SERVICE_DONE event, reason CTA_REASON_FINISHED
xxxCloseService	Enqueue CTAEVN_DISP_CLOSE_SERVICE_DONE event, reason CTA_REASON_FINISHED
xxxProcessEvent	Return SUCCESS
xxxProcessCommand	Return SUCCESS
xxxAddRTC	Return SUCCESS
xxxRemoveRTC	Return SUCCESS
xxxGetText	Return NULL
xxxFormatMessage	Format event or command by printing the IDs
xxxSetTraceLevel	Return SUCCESS
xxxFormatTraceBuffer	Return SUCCESS
xxxGetFunctionPointer	Return NULL

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
STATIC CTASVCMGR_FNS tikSvcmgrFns =
{
    sizeof(CTASVCMGR_FNS),
    tikDefineService,         /* xxxDefineService()              */
    tikAttachServiceManager,  /* xxxAttachServiceManager()       */
    tikDetachServiceManager,  /* xxxDetachServiceManager()       */
    tikOpenServiceManager,    /* xxxOpenServiceManager()         */
    tikCloseServiceManager,   /* xxxCloseServiceManager()        */
    tikOpenService,           /* xxxOpenService()                */
    tikCloseService,          /* xxxCloseService()               */
    NULL,                     /* xxxProcessEvent()               */
    tikProcessCommand,        /* xxxProcessCommand()             */
    NULL,                     /* xxxAddRTC()                     */
    NULL,                     /* xxxRemoveRTC()                  */
    tikGetText,               /* xxxGetText()                    */
    tikFormatMessage,         /* xxxFormatMessage()              */
    tikSetTraceLevel,         /* xxxSetTraceLevel()              */
    tikFormatTraceBuffer,     /* xxxFormatTraceBuffer()          */
    NULL,                     /* xxxGetFunctionPointer() -future */
};
```

Version and compatibility information

There are two types of version information for a Natural Access service:

- Version ID
- Compatibility level

The version ID consists of revision information that is intended to be for human-readable identification purposes. The version information consists of a revision number and a build date.

The revision number is reported in the format ***major_rev***.***minor_rev*** (for example, 2.2). The major revision number is typically incremented when a major change in functionality is released. The minor revision number is typically incremented when small changes (for example, bug fixes) are released. Typically, the minor revision number is reset to zero when the major revision number is incremented.

The build date is in ***mm dd yyyy*** format. It is typically encoded using the __DATE__ compiler built-in.

Compatibility levels are used for runtime interoperability checking between a target software module and a caller of that software module. For example, the compatibility level of the dispatcher (acting as the target software module) may be checked by a Natural Access service (acting as a caller).

A compatibility level consists of a 1-based integer that is assigned to a target software module. This level is incremented when the software module is modified such that it is no longer compatible with existing clients. Examples of such modifications include:

- Changes to the signature of a public function call
- Removal of a public function call

There are three interoperability scenarios to consider when writing a Natural Access service:

- Dispatcher to Natural Access service
- Natural Access service API to Natural Access application
- Natural Access service SPI to another Natural Access service

The following table describes processing aspects of these scenarios:

Function call	Assigning a compatibility level	Requesting a compatibility level	Comparing assigned to requested compatibility levels
Dispatcher function called from Natural Access service	Done internally by Natural Access; transparent to service writer.	Use the DISP_COMPATLEVEL macro as the value for ***reqdisplevel*** in CTAINTREV_INFO structure	**dispRegisterServiceManager** compares the internal value of the dispatcher with the value in ***reqdisplevel***. If they are equal, processing continues. Otherwise, processing aborts.
Natural Access service function called from Natural Access application	Set the ***exapilevel*** field of the CTAINTREV_INFO structure.	Get the XXXAPI_COMPATLEVEL macro from the target's *xxxdef.h* file.	Write an application function to call **ctaGetServiceVersion** for the target service. Compare XXXAPI_COMPATLEVEL with the ***compatlevel*** field returned in the CTA_REV_INFO structure.
Natural Access service function called from another Natural Access service	Set the ***exspilevel*** field of the CTAINTREV_INFO structure.	Get the XXXSPI_COMPATLEVEL macro from the target's *xxxdef.h* file. In the client's **xxxDefineService** function, create a CTAREQSVC_INFO structure and set the target ***reqspilevel*** field to the target's XXXSPI_COMPATLEVEL.	**dispRegisterService** compares the client's ***reqspilevel*** with the targets ***exspilevel***. If they are equal, processing continues. Otherwise, processing aborts.

Specifying version and compatibility levels

To specify compatibility levels and version IDs for a service:

- *xxxdef.h* should contain the four macros: XXX_MAJORREV, XXX_MINORREV, XXXAPI_COMPATLEVEL, XXXSPI_COMPATLEVEL
- *xxxbnd.c* should initialize the appropriate CTAINTREV_INFO fields for the service manager in **xxxInitializeManager**. Initialize the appropriate CTAINTREV fields for the service implementation in **xxxDefineService**.

The CTAINTREV_INFO structure is defined as follows:

```
typedef struct            /* CTA internal revision info structure */
{
    DWORD size;           /* Size of the returned structure       */
    DWORD majorrev;       /* Major revision of service/manager    */
    DWORD minorrev;       /* Minor revision of service/manager    */
    char  builddate [12]; /* Build date, "Mmm dd yyyy\0"          */
    DWORD reqdisplevel;   /* Required compat level of Dispatcher  */
    DWORD expapilevel;    /* Exported compat level of Service API */
    DWORD expspilevel;    /* Exported compat level of Service SPI */
} CTAINTREV_INFO;
```

tik service

<<<< **excerpt from** *tikdef.h* >>>>

```
-------------------------------------------------------
 TIK Version IDs
    - These ID's are used to identify the revision level of
      this service.
 ------------------------------------------------------------*/
#define TIK_MAJORREV                        1
#define TIK_MINORREV                        0
/*-------------------------------------------------------------
 TIK Compatibility Level IDs
    - These ID's are used to determine runtime compatibility
      between the installed TIK service and clients of the TIK
      service.
    - API_COMPATLEVEL is used by Natural Access application
      developers who want to check the "hard coded" value in
      their application against the "hard coded" value in the
      installed TIK service
    - SPI_COMPATLEVEL is useful to Natural Access service
      writers who want to check the "hard coded" value in
      their client service against the "hard coded" value in
      the installed TIK service.
  ------------------------------------------------------------*/
#define TIKAPI_COMPATLEVEL                  1
#define TIKSPI_COMPATLEVEL                  1
```

Initializing global and local tracemasks

Control tracing in one of two ways: with *ctdaemon*, where a tracemask can be set using the *ctdaemon* command processor, or with an internal, service-specific tracemask that can be set within an application.

With two mechanisms to control tracing, an application developer is able to:

- Trace a service on a global basis across multiple applications. By setting the global tracemask to a particular value, the tracemask is seen by all instances of a service in all applications.
- Trace a service on an application specific basis by calling **ctaSetTraceLevel** for a specific service in a specific application on a per context basis.

To determine whether to generate a trace record associated with a trace category, the service must check both the global tracemask and the local (service-specific) tracemask.

To check the global tracemask, the service, upon initialization, must first call **dispGetTracePointer** to acquire a pointer to the global tracemask. It then stores this pointer so it can be accessed through all service manager and service implementation functions.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/* ------------------------------------------
Global service trace mask
------------------------------------------------*/
volatile DWORD *tikTracePointer;
```

To define a local tracemask, define a tracemask on a per context basis. A tracemask is defined as a DWORD.

tik service

<<<< excerpt from *tikdef.h* **>>>>**

```
/* ------------------------------------------------------------
   A DWORD tracemask is included in the definition of a
   TIK_CHANNEL_OBJECT since it is allocated (as a mgrcontext) on a
   per CTA context basis.
------------------------------------------------------------*/
typedef struct _TIK_CHAN
{
   DWORD        size;             /* Size of this structure           */
   DWORD        timestamp;        /* Context creation timestamp.      */
   CTAHD        ctahd;            /* Opening CTA context handle       */
   DWORD        channelId;        /* Opened channel number.           */
   DWORD        state;            /* current state of channel.        */
   DWORD        owner;            /* Current owner of channel.        */
   DWORD        tracemask;        /* Context specific tracemask       */
   TIK_DEVICE_OBJECT *ptikDevice; /* Pointer to "tick device" object. */
   TIK_CHANNEL_INFO channel_info; /* Channel-specific info.           */
} TIK_CHANNEL_OBJECT;
```

xxxInitializeManager example

tik service

<<<< **excerpt from** *tikbnd.c* >>>>

```
DWORD NMSAPI tikInitializeManager( void )
{
    DWORD ret;

CTABEGIN("tikInitializeManager");

static BOOL initialized = FALSE;
    if (initialized)
    {
       return CTALOGERROR(NULL_CTAHD, CTAERR_ALREADY_INITIALIZED, TIK_SVCID);
    }
    /* Set manager revision information. */
    tikMgrRevInfo.size = sizeof(CTAINTREV_INFO);
    tikMgrRevInfo.majorrev = TIK_MAJORREV;
    tikMgrRevInfo.minorrev = TIK_MINORREV;
    strcpy (tikMgrRevInfo.builddate, tikBuildDate);
    tikMgrRevInfo.expapilevel = TIKAPI_COMPATLEVEL;
    tikMgrRevInfo.expspilevel = TIKSPI_COMPATLEVEL;
    tikMgrRevInfo.reqdisplevel = DISP_COMPATLEVEL;

    dispGetTracePointer(&tikTracePointer);

    ret = dispRegisterServiceManager("TIKMGR", &tikSvcmgrFns,
              &tikMgrRevInfo);
    if (ret != SUCCESS)
    {
       return CTALOGERROR(NULL_CTAHD, ret, TIK_SVCID);
    }
    initialized = TRUE;
    return SUCCESS;
}
```

xxxDefineService

After initializing a service manager and its binding functions, services are initialized. The dispatcher initializes each service specified in the call to **ctaInitialize** by invoking the appropriate **xxxDefineService** binding function.

xxxDefineService performs the following tasks:

- Declares the parameter descriptor tables.
- Declares the version numbers for the service.
- Requests SPI compatibility level checking for other services whose SPI will be called from this service.

The dispatcher manages parameters for registered services. During service registration (during the call to **ctaInitialize**), the service must pass the pointer to the parameter description table defined in *xxxparm.c* to the dispatcher through **dispRegisterService**.

After the parameter table is registered with the dispatcher, it must not be modified. Each individual parameter descriptor must remain valid and unmodified. All dynamic changes to parameter descriptor tables and their contents must take place before service registration.

tik service

<<<< **excerpt from** *tikbnd.c* >>>>

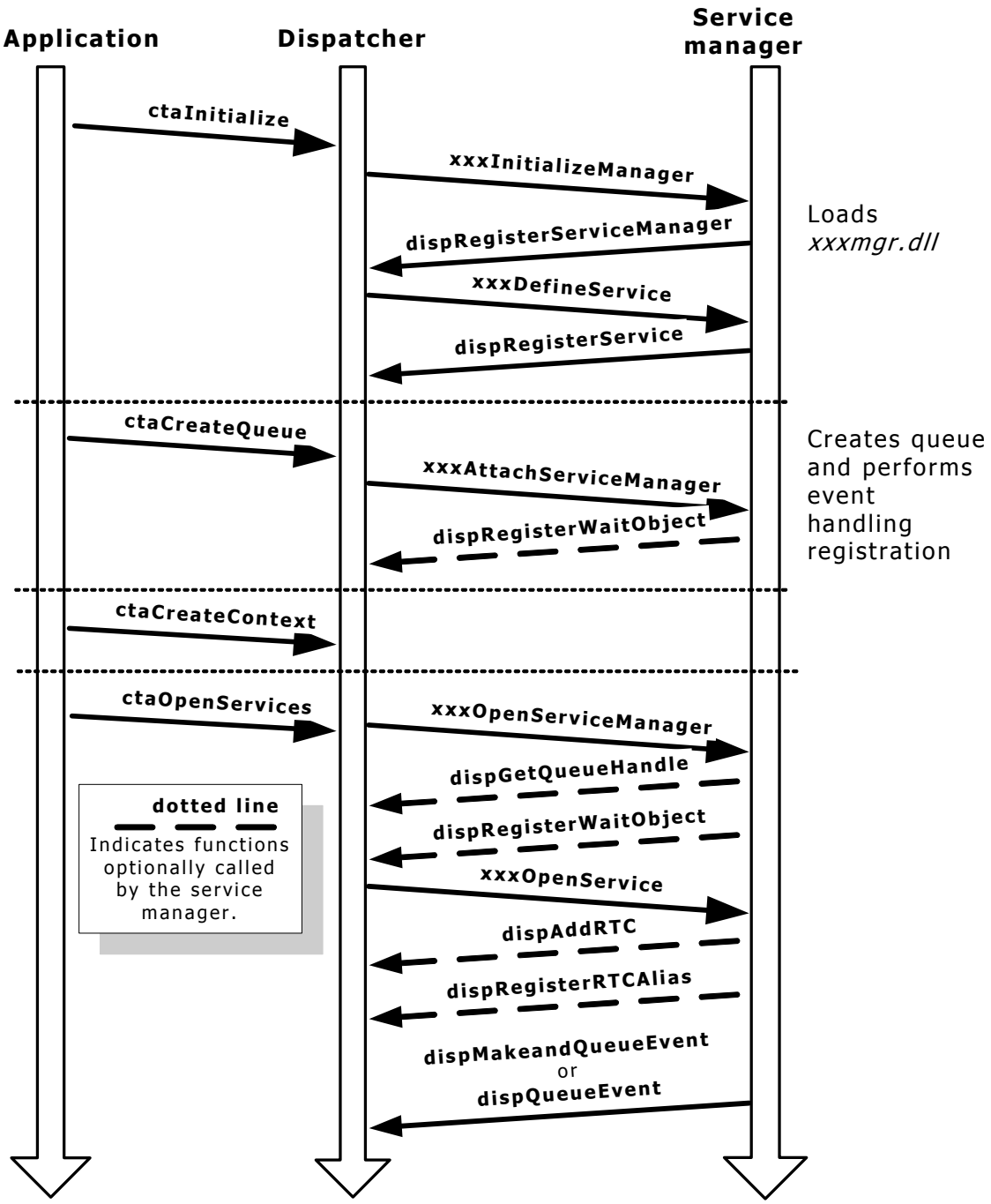
```
STATIC DWORD NMSAPI tikDefineService( char* svcname )
{
    DWORD ret;
    /*
     * Needed by Natural Access provided error logging
     * service
     */
    CTABEGIN("tikDefineService");

    /* Set service revision info. */
    tikSvcRevInfo.size = sizeof(CTAINTREV_INFO);
    tikSvcRevInfo.majorrev = TIK_MAJORREv;
    tikSvcRevInfo.minorrev = TIK_MINORREV;
    tikSvcRevInfo.expapilevel = TIKAPI_COMPATLEVEL;
    tikSvcRevInfo.expspilevel = TIKSPI_COMPATLEVEL;
    tikSvcRevInfo.reqdisplevel = DISP_COMPATLEVEL;
    strcpy(tikSvcRevInfo.builddate, __DATE__);


    /* The most important phase of initializing a service! */
    ret = (dispRegisterService("TIK",
                               TIK_SVCID,
                               "TIKMGR",
                               &tikSvcRevInfo,
                               NULL,
                               0,
                               _tikParmDescTable,
                               NULL));
    if ( ret != SUCCESS )
    {
       return CTALOGERROR(NULL_CTAHD, ret, TIK_SVCID);
    }
    return SUCCESS;
}
```

Initializing managed resource event handling

xxxAttachServiceManager should be implemented for a service that handles events from a multiplexed managed resource. If the managed resource is not multiplexed, use **xxxOpenServiceManager** to initialize event handling on a per context basis (see *Starting services* on page 89).



Event handling initialization

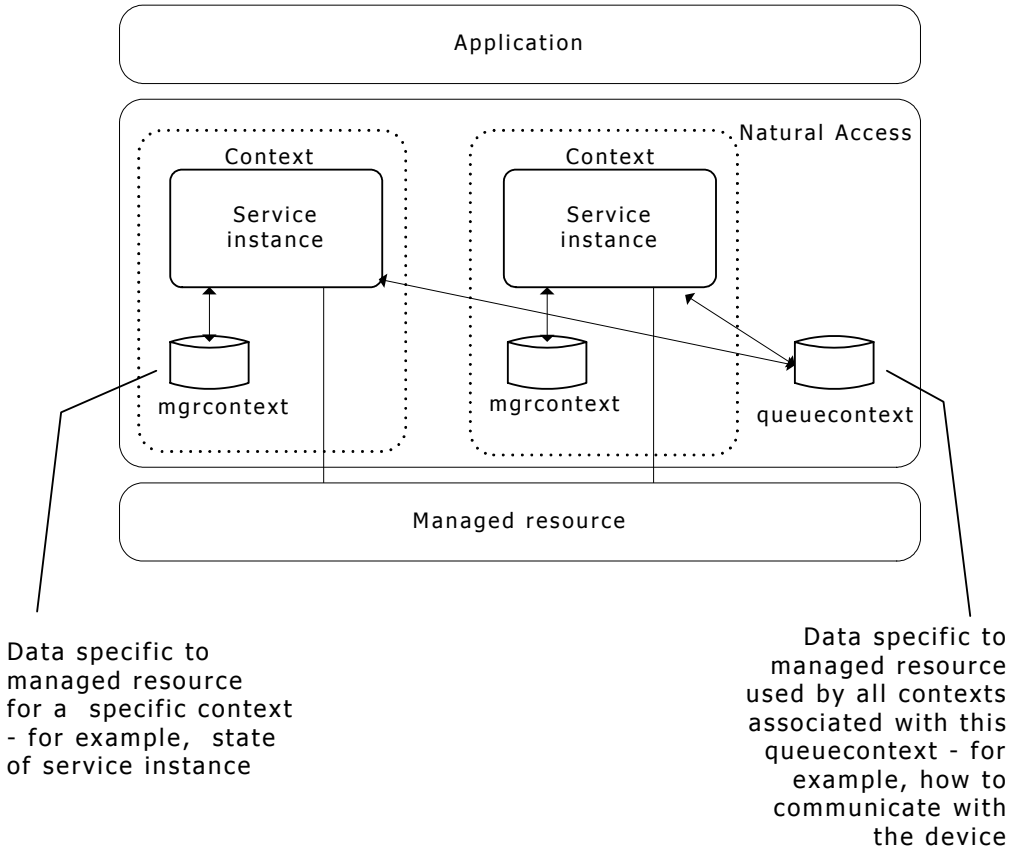
Communicate with the managed resource during the **xxxOpenServiceManager** invocation only if it is possible to guarantee that the resource will be available at this time. Otherwise the application is prevented from creating a queue. If you cannot guarantee that the resource will be available, defer communication with the managed resource until when the service is being opened.

In **xxxAttachServiceManager**, the following actions should take place:

1. Define and allocate a data structure containing:
    - Information about the managed resource as a whole (as discussed in Understanding the managed resource).
    - Demultiplexing information to ensure that the event is processed on behalf of the correct context.

    Once this data structure is allocated, a pointer to it is returned to Natural Access as a (void *)***queuecontext***. This ***queuecontext*** is then passed to other binding functions as appropriate. It is cast to (void *) to allow developers to create any data structure deemed appropriate and have Natural Access pass it around transparently to other binding functions (see the following illustration).

    Note: Do not communicate with the managed resource when **xxxAttachServiceManager** is invoked until you are sure that the resource is available. Otherwise, the application cannot create a queue. If you cannot guarantee that the resource is available, defer communication with the resource until the service is opened.



Device specific data objects

tik service

Since the tik server is a multiplexed managed resource, the following ***queuecontext*** is defined and allocated.

<<<< excerpt from *tiksys.h* **>>>>**

```
/*-------------------------------------------------------------
"TICK DEVICE" Object
  - Used as a "queuecontext"
  - refers to an instance of a "ticker" which can support
    up to 10 logical channels
-------------------------------------------------------------*/
struct _TIK_CHAN;

typedef struct
{
    DWORD              size;            /* Size of this structure    */
    TSIIPC_STREAM_HD   connector;       /* Client side IPC handle.   */
    BOOL               pendingread;     /* Flag for pending IPC read */
    unsigned           readcount;       /* Outstanding requested     */
                                        /* read count                */
    TIK_RSP_COM_OBJ    response;        /* Server response via       */
                                        /* comm channel              */
    struct _TIK_CHAN   *channelTbl[10]; /* Array of logical          */
                                        /* channels for this device  */
}
```

2. A single wait object is allocated and registered with the dispatcher (within the newly created Natural Access event queue) with **dispRegisterWaitObject**. This wait object fires whenever a message is received from the managed resource. The arguments to **dispRegisterWaitObject** include the ***queuecontext*** and a function pointer to the event handler callback function (**xxxFetchAndProcess**) that performs event demultiplexing and processing (initiated by the call to **ctaWaitEvent**). Wait objects are operating system-specific entities.

   The following table identifies the appropriate wait object per operating system:

Operating system	Native wait object
Windows	Manual reset event object
UNIX	pollfd structure

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
tikAttachServiceManager
  - Declare Wait object
  - Allocate "Device" object which represents the "tick device"
  - Register the wait object with Natural Access via dispRegisterWaitObject
  - Set the device object to the "queuecontext" since
    demultiplexing of events is done a per "tick device" basis
-------------------------------------------------------------*/
STATIC DWORD NMSAPI tikAttachServiceManager (CTAQUEUEHD ctaqueuehd,
                       unsigned mode,
                      void **queuecontext)
```

```
{
  TIK_DEVICE_OBJECT  *ptikDevice = NULL;
  TSIIPC_WAIT_OBJ_HD waitobj;
  CTA_WAITOBJ        *ctawaitobj;
  DWORD              ret;

  /* Needed by Natural Access provided error logging service */
  CTABEGIN("tikAttachServiceManager");

  /* Initialize queuecontext */
  *queuecontext = NULL;
  /* Create Tick "device" object. */
  if ( (ptikDevice=tikCreateDeviceObject()) == NULL )
  {
      return CTALOGERROR( NULL_CTAHD,CTAERR_OUT_OF_MEMORY,TIK_SVCID );
  }

  /* ESTABLISH IPC CONNECTION
  * In the process of establishing IPC connection,
  * OS specific server-to-client communication
  * wait object is created within NMS tsi library.
  * Under Windows: wait object(manual reset event) is
  * created via Window Socket API
  * (WSACreateEvent).
  * Under UNIX : wait object(pollfd structure) is allocated
  * via alloc(). */
  else if ( (ret=tikSetupIPC( ptikDevice )) != SUCCESS )
  {
      tikDestroyDeviceObject( ptikDevice );
      return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
  }
  /* Get tsi library created waitobject to register
  * with Natural Access. */
  else if ( tsiIPCGetWaitObject(ptikDevice->readhd, &waitobj)!= SUCCESS )
  {
      tikDestroyDeviceObject( ptikDevice );
      return CTALOGERROR( NULL_CTAHD, CTAERR_OS_INTERNAL_ERROR, TIK_SVCID );
  }
  /* Register created waitobject with Natural Access. */
  else if ( (ret=dispRegisterWaitObject(  ctaqueuehd,
                                          (CTA_WAITOBJ *)waitobj,
                                          tikFetchAndProcess,
                                          (void *)ptikDevice )) != SUCCESS )
  {
      tikDestroyDeviceObject( ptikDevice );
      return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
}
/* The following two function calls(dispFindWaitObject,
  *tsiIPCRegisterWaitObjectCopy) are required in order to
  *  make use of NMS's abstracted IPC communications layer. */

  /* Acquire registered CTA WaitObject to register with *TSIIPC. */
  else if ( (ret=dispFindWaitObject(  ctaqueuehd,
                (CTA_WAITOBJ *)waitobj,
                &ctawaitobj) ) != SUCCESS )
  {
    tikDestroyDeviceObject( ptikDevice );
    return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
  }
  /* Register CTA WaitObject with TSIIPC. */
  else if ( (ret=tsiIPCRegisterWaitObjectCopy(ptikDevice->readhd,
            (TSIIPC_WAIT_OBJ_HD)ctawaitobj )) != SUCCESS )
  {
    tikDestroyDeviceObject( ptikDevice );
    return CTALOGERROR(NULL_CTAHD,TIKERR_CAN_NOT_CONNECT,TIK_SVCID);
  }

  /* Assign the device object to the "queuecontext" so that
  * event handling can be performed on a "device" basis.
  * Using Natural Access terminology,"Mux Handling will
  * be on a per queue basis (as opposed to a per context
```

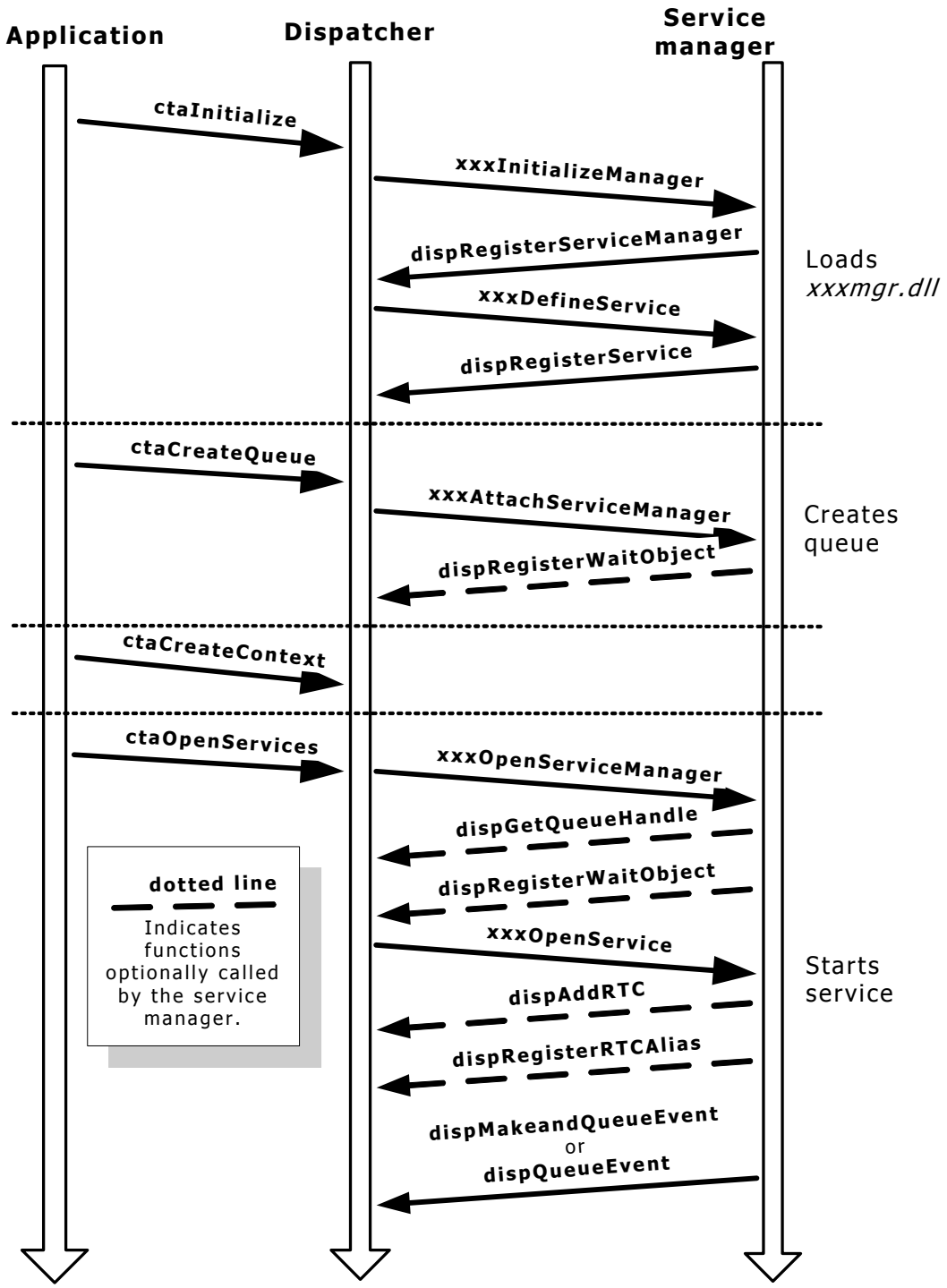
```
  * basis)". */
  else
    *queuecontext = (void *)ptikDevice;

  return SUCCESS;
} /* end tikAttachServiceManager() */
```

Starting services

Service startup occurs when the application calls **ctaOpenServices**. Two binding functions are called during service startup:

- **xxxOpenServiceManager**
- **xxxOpenService**



Service startup

xxxOpenServiceManager

xxxOpenServiceManager accomplishes the following tasks:

1. Defines and allocates a data structure containing information pertaining to a client (context) of the managed resource (as discussed in Understanding the managed resource).

   Once this data structure is allocated, a pointer to it is returned to Natural Access as a (void *) ***mgrcontext***. This ***mgrcontext*** is then passed to other binding functions as appropriate. It is cast to (void *) to allow developers to create any data structure deemed appropriate and to enable Natural Access to pass it around transparently to other binding functions.

   tik service

   <<<< **excerpt from** *tiksys.h* >>>>

```
/*-------------------------------------------------------------
 "TICK CHANNEL" Object
 - used as a "mgrcontext"
 - only one TICK CHANNEL object per CTA context (ctahd)
 - a TICK CHANNEL is one of the up to 10 logical channels
   supported by a TICK DEVICE
-------------------------------------------------------------*/
typedef struct _TIK_CHAN
{
   DWORD             size;        /* Size of this structure         */
   DWORD             timestamp;   /* Context creation timestamp.    */
   CTAHD             ctahd;       /* Opening context handle         */
   DWORD             channelId;   /* Opened channel number.         */
   DWORD             state;       /* current state of channel.      */
   DWORD             owner;       /* Current owner of this channel. */
   DWORD             tracemask;   /* Context specific tracemask     */
   TIK_DEVICE_OBJECT *ptikDevice; /* pointer to "tick device" object*/
   TIK_CHANNEL_INFO  ChannelInfo; /* Channel-specific information   */
} TIK_CHANNEL_OBJECT;
```

2. If the managed resource is not a multiplexed managed resource, **xxxOpenServiceManager** must also allocate a wait object and register it with the dispatcher using **dispRegisterWaitObject**. This wait object fires only for those messages received from the managed resource that are destined for this specific context. The arguments to **dispRegisterWaitObject** include the ***mgrcontext*** and a function pointer to the event handler callback function (**xxxFetchAndProcess**) that performs event processing. No demultiplexing processing is required. Wait objects are operating system specific entities. The following table identifies the appropriate wait object per operating system:

Operating system	Native wait object
Windows	Manual reset event object
UNIX	pollfd struct

Note: In this case, a wait object is created and registered with the dispatcher for every context that is created on this service. This may be problematic for those operating systems with a limited number of wait objects available for general use.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
 tikOpenServiceManager
 - Allocate Service Manager specific data (a manager context)
 - In this case, a manager context corresponds to a Tick
CHANNEL object.
-------------------------------------------------------------*/
STATIC DWORD NMSAPI tikOpenServiceManager( CTAHD ctahd,
                          void *queuecontext,
                          void **mgrcontext )
{
  TIK_DEVICE_OBJECT *ptikDevice=(TIK_DEVICE_OBJECT *)queuecontext;
  TIK_CHANNEL_OBJECT *ptikChannel = NULL;

  /* Needed by Natural Access provided error logging service */
  CTABEGIN("tikOpenServiceManager");
  if ((ptikChannel=tikCreateChannelObject(ctahd,ptikDevice))==NULL)
  {
    *mgrcontext = NULL;
    return CTALOGERROR( NULL_CTAHD,CTAERR_OUT_OF_MEMORY,TIK_SVCID );
  }
  /* Assign mgrcontext to be the Channel Object.
  * A "manager context" is supposed to be data that is
  * specific to a single instance of a context
  * (ctahd); a Channel Object is just that.
  */
  else
    *mgrcontext = (void *)ptikChannel;

  return SUCCESS;
} /* end tikOpenServiceManager() */
```

xxxOpenService

The **xxxOpenService** binding function performs any asynchronous interaction with the managed resource that is necessary to initialize the service and make it ready to accept and process commands on behalf of a client. **xxxOpenService** and **xxxCloseService** are the only asynchronous binding functions. Since they are asynchronous, they must return an event to the dispatcher, and ultimately to the application, when they complete their operation.

The asynchronous open service routine must have a way to be notified asynchronously by an event that the initialization completed. If the service manages a device, this event usually comes through the wait object and wait function registered with the dispatcher at **xxxAttachServiceManager**, or **xxxOpenServiceManager** time. The wait function is responsible for processing the event inside the service, possibly executing other initialization commands and finally passing the DONE event to the dispatcher so that the service can be placed in the opened state.

xxxOpenService must place an event such as CTAEVN_DISP_OPEN_SERVICE_DONE in the Natural Access queue to notify the dispatcher that the service completed its initialization and is ready to process commands. **xxxOpenService** calls **dispMakeAndQueueEvent** when is notified that the initialization is complete. The service must pass the service ID (the ***svcid*** argument passed to **xxxOpenService**) as the source argument and

CTA_SYS_SVCID as the destination. The DISP_EVENT structure is processed into the Natural Access event that gets passed up to the application.

Note: Even services that do not require asynchronous open service processing must queue the CTAEVN_DISP_OPEN_SERVICE_DONE completion event using **dispMakeAndQueueEvent** or **dispQueueEvent**. (**dispMakeAndQueueEvent** is a wrapper function for **dispQueueEvent**.)

If your service will use the Natural Access runtime control features (described in the Natural Access runtime control section), invoke **dispAddRTC**, **dispRegisterRTCAlias**, or both functions during the execution of **xxxOpenService**.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
 tikOpenService
 - Initialize any dedicated resources (typically on a per-
   context basis)
 - For the Tik Service, this means allocating a logical
   channel on the "physical" TICK Device.
-------------------------------------------------------------*/
STATIC DWORD NMSAPI tikOpenService( CTAHD ctahd,
                    void *mgrcontext,
                    char *svcname,
                    unsigned svcid,
                    CTA_MVIP_ADDR *mvipaddr,
                    CTA_SERVICE_ARGS *svcargs )
{
  TIK_CHANNEL_OBJECT *ptikChannel= (TIK_CHANNEL_OBJECT *)mgrcontext;
  TIK_CMD_COM_OBJ cmd = { 0 };
  DWORD ret;
  /* Needed by Natural Access provided error logging service */
  CTABEGIN("tikOpenService");
  ASSERT( tsiStricmp(svcname, "TIK") == 0 );
  ASSERT( svcid == TIK_SVCID );
  /* svcargs->args[0] contain logical number to use in server. */
  ptikChannel->channelId = svcargs->args[0];
  if( ptikChannel->channelId < 0||ptikChannel->channelId > 9 )
  {
    return(CTALOGERROR(NULL_CTAHD,CTAERR_BAD_ARGUMENT,TIK_SVCID));
  }
  else
    ptikChannel->ptikDevice-    >channelTbl[ptikChannel->channelId] = ptikChannel;
  /* If CTA_TRACEMASK_DEBUG_BIT0 is enabled,
  * log state transition information.
  */
  if ( TRACEMASK_BIT0_SET(ptikChannel->tracemask) )
  {
    ret = tikLogStateTransition( ptikChannel,
                                 ptikChannel->state,
                                 CHANNEL_OPENING,
                                 "ctaOpenServices" );
    if ( ret != SUCCESS )
    {
      return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
    }
  }

  ptikChannel->state = CHANNEL_OPENING;
  /* Send TIKSVR_OPEN_CHANNEL request. */
  cmd.size =         sizeof( TIK_CMD_COM_OBJ );
  cmd.timestamp =    time(NULL);
  cmd.client_id =    ptikChannel->channelId;
```

```
  cmd.command =      TIKSVR_OPEN_CHANNEL;
  cmd.data1 =        ptikChannel->channelId;
  cmd.data2 =        0;
  cmd.data3 =        0;

  if( (ret=tikSendClientCommand( ptikChannel, &cmd )) != SUCCESS )
  {
    tikDestroyChannelObject( ptikChannel );
    return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
  }
  else if( (ret=tikReadServerResponse( ptikChannel )) !=SUCCESS )
  {
    tikDestroyChannelObject( ptikChannel );
    return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
  }
  return SUCCESS;
} /* end tikOpenService() */
```

<<<< excerpt from *tikutils.c* **>>>>**

```
DWORD NMSAPI tikFetchAndProcess( CTAQUEUEHD ctaqueuehd,
                CTA_WAITOBJ *waitobj,
                void *arg )
{
  TIK_DEVICE_OBJECT *ptikDevice = (TIK_DEVICE_OBJECT *) arg;
  TIK_CHANNEL_OBJECT *ptikChannel = NULL;
  TIK_RSP_COM_OBJ *rsp = &ptikDevice->response;
  TSIIPC_COMMAND_STATUS status = { 0 };
  DWORD ret = SUCCESS;

  /* Complete IPC read. */
  if ( tikCompleteReadServerResponse( ptikDevice ) != SUCCESS )
  {
    return ret;
  }
  /* etc ... */
  /* Process incoming message from tik server.*/
  switch ( rsp->response )
  {
    case TIKSVR_OPEN_CHANNEL_RSP:
    {
       ret = tikProcessOpenChannelResponse( ptikChannel );
       break;
    }
    /* etc ... */
  }
  /* etc ... */
}

DWORD tikProcessOpenChannelResponse(TIK_CHANNEL_OBJECT *ptikChannel )
{
  TIK_RSP_COM_OBJ *rsp = &(ptikChannel->ptikDevice->response);
  DWORD reason;
  DWORD ret;

  /* etc ... */

  /* Translate reason code from incoming message to
  * associated Natural Access reason.
  */
  switch ( rsp->reason )
  {
    case TIKSVR_CHANNEL_OPENED:
    {
       ptikChannel->state = CHANNEL_TIMER_IDLE;
       reason = CTA_REASON_FINISHED;
       break;
    }
    /* etc ... */
  }
  /* etc ... */
```

```
  /* Create and enqueue open service DONE event. */
  ret = dispMakeAndQueueEvent( ptikChannel->ctahd,
                               CTAEVN_DISP_OPEN_SERVICE_DONE,
                               reason,
                               TIK_SVCID,
                               CTA_SYS_SVCID);
  return( ret );

} /* end tikProcessOpenChannelResponse() */
```

Shutting down services

Service shutdown occurs when **ctaCloseServices** is called. The following binding functions are called:

- **xxxCloseService**
- **xxxCloseServiceManager**

xxxCloseService

The **xxxCloseService** binding function is invoked when a service is closed on a context. This function can shut down the managed resource.

If your service will use the Natural Access runtime control features (described in the Natural Access runtime control section), invoke **dispRemoveRTC**, **dispUnregisterRTCAlias**, or both functions during the execution of **xxxCloseService**.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
  tikCloseService
    - shut down the managed resource
 -------------------------------------------------------------*/
STATIC DWORD NMSAPI tikCloseService ( CTAHD     ctahd,
                                      void     *mgrcontext,
                                      char     *svcname,
                                      unsigned  svcid )
{
   TIK_CHANNEL_OBJECT  *ptikChannel = (TIK_CHANNEL_OBJECT*)mgrcontext;
   DWORD               ret;

  /* Needed by Natural Access provided error logging service */
   CTABEGIN("tikCloseService");

   ASSERT( svcid == TIK_SVCID );

   /* Perform graceful unwinding prior to service close. */
   if( (ret=tikHandleCloseService( ptikChannel )) != SUCCESS )
   {
      return CTALOGERROR( ctahd, ret, TIK_SVCID );
   }

   return SUCCESS;

}  /* end tikCloseService() */
```

xxxCloseServiceManager

Closing a service manager involves cleaning up all that was done when opening the service manager to the queue. If memory was allocated during the open, this memory should be freed during close processing. If a wait object was registered during open, then it must be unregistered using **dispUnregisterWaitObject**.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
 tikCloseServiceManager
    - Deallocate Service Manager specific data (a manager context)
 -------------------------------------------------------------*/
STATIC DWORD NMSAPI tikCloseServiceManager( CTAHD  ctahd,
                                            void   *mgrcontext )
{
TIK_CHANNEL_OBJECT *ptikChannel= (TIK_CHANNEL_OBJECT *)mgrcontext;

        tikDestroyChannel( ptikChannel );

        return SUCCESS;
    }   /* end tikCloseServiceManager() */
```

Shutting down event handling

The **xxxDetachServiceManager** binding function is invoked when the queue is destroyed by **ctaDestroyQueue**. It is a synchronous function and must return to the dispatcher when detach processing is completed.

Detaching a service manager involves cleaning up all that was done when attaching the service manager to the queue. If memory was allocated during attach, this memory should be freed during detach processing. If a wait object was registered during attach, then it must be unregistered using **dispUnregisterWaitObject** during detach processing.

The queue specific data passed back to the dispatcher by **xxxAttachServiceManager** (the ***queuecontext*** argument) is passed into **xxxDetachServiceManager** so that the service manager can clean up these allocated and registered objects.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
 tikDetachServiceManager
    - Undeclare Wait object
    - Deallocate Resource object
 -------------------------------------------------------------*/

STATIC DWORD NMSAPI tikDetachServiceManager( CTAQUEUEHD ctaqueuehd,
                                             void       *queuecontext)
{
   TIK_DEVICE_OBJECT   *ptikDevice = (TIK_DEVICE_OBJECT *)queuecontext;
   TSIIPC_WAIT_OBJ_HD  waitobj;
   DWORD               ret;

   /* Needed by Natural Access provided error logging service */
   CTABEGIN("tikDetachServiceManager");

   /* Get IPC waitobject to un-register. */
   if ( tsiIPCGetWaitObject( ptikDevice->connector, &waitobj ) !=
        SUCCESS )
   {
      tikDestroyDevice( ptikDevice );
      return CTALOGERROR(
               NULL_CTAHD, CTAERR_OS_INTERNAL_ERROR, TIK_SVCID);
   }
   /* Unregister waitobject with Natural Access. */
   else if ( (ret=dispUnregisterWaitObject(
              ctaqueuehd, (CTA_WAITOBJ *)&waitobj)) != SUCCESS)
   {
      tikDestroyDevice( ptikDevice );
      return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
   }
   /* Teardown IPC connection. */
   else if ( (ret=tikTeardownIPC( ptikDevice )) != SUCCESS )
   {
      tikDestroyDevice( ptikDevice );
      return CTALOGERROR( NULL_CTAHD, ret, TIK_SVCID );
   }
   /* Destroy created service manager object. */
   else
      tikDestroyDevice( ptikDevice );

   return SUCCESS;

}  /* end tikDetachServiceManager() */
```

Runtime binding functions

The runtime binding functions include functions for:

- Command processing
- Error handling
- Trace handling
- Event processing

Except for **xxxProcessCommand**, all the runtime binding functions can return either SUCCESS or an error. **xxxProcessCommand** can return SUCCESS, SUCCESS_RESPONSE, or an error.

The following tables list the runtime binding functions:

xxxProcessCommand

Description:	• Defines table of function pointers to API implementation functions. • Performs lookup based on command code within the DISP_COMMAND message buffer. • Performs upcall to appropriate function.
Behavior:	Synchronous
Important data structure:	Array of function pointers to implementation functions in ascending order by command code.
Dispatcher functions called:	None.
Returns:	SUCCESS, SUCCESS_RESPONSE, or an error.
Access through Natural Access:	Service API calls.
Required?	Must be implemented.

xxxGetText

Description:	• Converts error codes, event codes, reason codes, and/or command codes to associated macro tag. • Uses ANSI C #param preprocessor directive.
Behavior:	Synchronous
Important data structure:	None
Dispatcher functions called:	None
Returns:	SUCCESS or an error.
Access Through Natural Access:	**ctaGetText**
Required?	Must be implemented.

xxxFormatMessage

Description:	• Generates ASCII message using information from **xxxGetText.** • Translates dispatcher commands and events to ASCII.
Behavior:	Synchronous
Important data structure:	DISP_MESSAGE
Dispatcher functions called:	None
Returns:	SUCCESS or an error.
Access Through Natural Access:	**ctaFormatEvent**
Required?	Optional (but recommended).

xxxSetTraceLevel

Description:	Sets tracemask value in ***mgrcontext*** to a new value.
Behavior:	Synchronous
Important data structure:	None
Dispatcher functions called:	None
Returns:	SUCCESS or an error.
Access Through Natural Access:	**ctaSetTraceLevel**
Required?	Optional

xxxFormatTraceBuffer

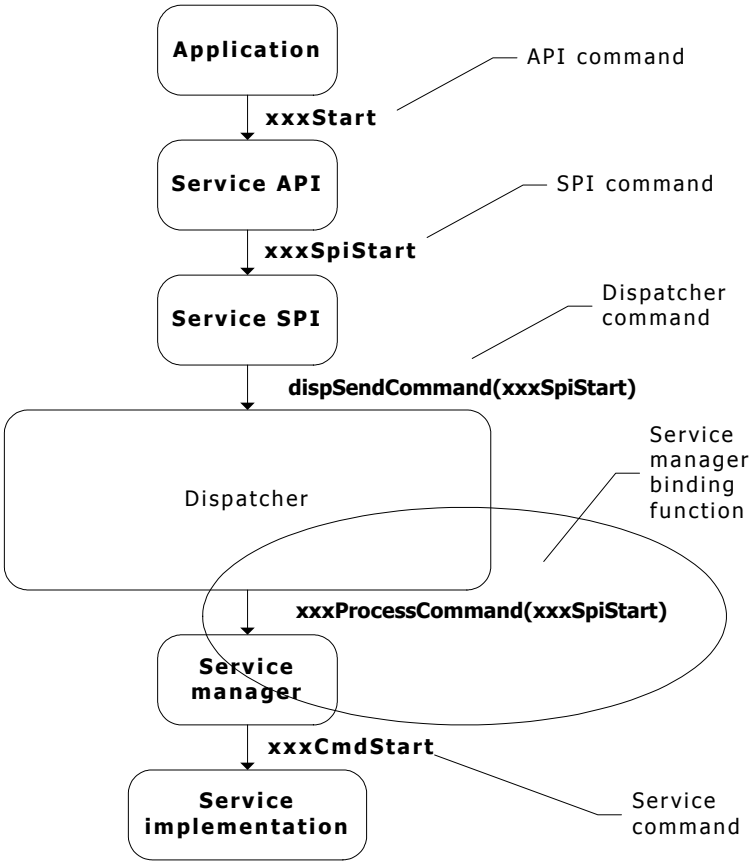
Description:	Converts binary trace messages to ASCII based on unique trace tags defined for service.
Behavior:	Synchronous
Important data structure:	None
Dispatcher functions called:	None
Returns:	SUCCESS or an error.
Access Through Natural Access:	Not accessible through Natural Access functions.
Required?	Optional

xxxProcessEvent

Description:	• Handles events from other services by identifying and processing the event. • Optionally generates a new event and passes it to the application.
Behavior:	Synchronous
Important data structure:	DISP_EVENT
Dispatcher functions called:	**dispMakeAndQueueEvent** **dispQueueEvent**
Returns:	SUCCESS or an error.
Access Through Natural Access:	Not accessible through Natural Access functions.
Required?	Optional

Command processing

When the application invokes a service function, the function calls the corresponding SPI function, which marshals the arguments into a DISP_COMMAND message buffer and passes the message to the dispatcher using **dispSendCommand**. The dispatcher then invokes the binding function **xxxProcessCommand**, which performs an upcall to the appropriate implementation function based on the command code stored within the DISP_COMMAND message buffer.



Command processing

To write a **xxxProcessCommand** binding function, create an array of function pointers to implementation functions indexed by the sequence number portion of the command code.

The ***mgrcontext*** that was allocated when the service manager was opened is passed in as an argument as well as a pointer to the DISP_COMMAND message buffer. These arguments are passed to the implementation function where unmarshalling occurs.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------------
  tikProcessCommand
    - declare table of function pointers to implementation
      functions
    - perform lookup based on incoming ID
    - perform upcall to appropriate implementation function
 -------------------------------------------------------------*/
STATIC DWORD NMSAPI tikProcessCommand( CTAHD         ctahd,
                                       void          *mgrcontext,
                                       DISP_COMMAND  *ctacmd )
{
   typedef DWORD (*TIK_CMD_FCN) ( TIK_CHANNEL_OBJECT *, DISP_COMMAND *);
   static const TIK_CMD_FCN tikCmdFcnTbl[]=
   {
       /* 0x00 */    tikCmdStartTimer,
       /* 0x01 */    tikCmdStopTimer,
       /* 0x02 */    tikCmdGetContextInfo,
   };
   TIK_CHANNEL_OBJECT *ptikChannel = ( TIK_CHANNEL_OBJECT *)mgrcontext;
   DWORD ret;

   /* Needed by Natural Access provided error logging service */
   CTABEGIN("tikProcessCommand");

    /*  Make sure command is defined, before invoking the
        appropriate cmd.  */
    if ((ctacmd->id & 0xfff) < sizeof(tikCmdFcnTbl)/sizeof(tikCmdFcnTbl[0]))
    {
      ret = ((*tikCmdFcnTbl[ ctacmd->id & 0xfff ]) (ptikChannel, ctacmd));
      if ( ret != SUCCESS && ret != SUCCESS_RESPONSE )
      {
        return CTALOGERROR( ctahd, ret, TIK_SVCID );
      }
    } else
      {
        return CTALOGERROR( ctahd, CTAERR_NOT_IMPLEMENTED, TIK_SVCID );
      }
    return SUCCESS;

}  /* end tikProcessCommand() */
```

Error handling

Use the binding functions **xxxGetText** and **xxxFormatMessage** to convert binary codes and trace records to ASCII.

xxxGetText

The binding function **xxxGetText** provides a way for the service to translate any service-specific error, reason, event, or command codes into a character string representation. **xxxGetText** is invoked by the dispatcher when the application calls **ctaGetText**. It is also invoked internally by **xxxFormatMessage** and **dispGetText**.

Errors, events, and commands should use the exact ASCII representation of the symbolic define used to define the code (for example, use the string ADIEVN_PLAY_DONE for the ADIEVN_PLAY_DONE event). Reasons, which are only applicable in the value field of a DONE event, can be translated in any way that makes it more readable during formatting. For example, CTA_REASON_FINISHED is translated to Finished.

It is recommended that services create a macro that expands all the error, reason, event, and command codes so that only one file needs to be updated when a code is added or modified. The macros expand into multiple case statements.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------
 tikGetText
    - convert error, event, reason, and/or command codes to
      associated ASCII macro identifier
 -------------------------------------------------------*/
STATIC const char* NMSAPI tikGetText( unsigned code )
{
    switch (code)
    {
        CTA_GENERIC_ERRORS();   /* see ctaerr.h */
        CTA_GENERIC_REASONS();  /* see ctaerr.h */
        TIK_ERRORS();           /* see tiksys.h */
        TIK_REASONS();          /* see tiksys.h */
        TIK_EVENTS();           /* see tiksys.h */
        TIK_COMMANDS();         /* see tiksys.h */
        default: return NULL ;
    }

}   /* end tikGetText() */
```

The macros for tik errors are shown. Additional macros exist for reasons, events, and commands.

<<<< excerpt from *tiksys.c* **>>>>**

```
/*-------------------------------------------------------
 tikGetText() Macros.
    - These macros convert TIK Service command, event, error,
      and reason codes to their corresponding ASCII macro
      identifier.
 -------------------------------------------------------*/
#define TEXTCASE(e) case e: return #e

#define TIK_ERRORS() \
TEXTCASE(TIKERR_COMM_FAILURE); \
TEXTCASE(TIKERR_CHANNEL_NOT_OPENED); \
TEXTCASE(TIKERR_OWNER_CONFLICT); \
TEXTCASE(TIKERR_UNKNOWN_SERVER_RESPONSE); \
TEXTCASE(TIKERR_CAN_NOT_CONNECT); \
```

xxxFormatMessage

The binding function **xxxFormatMessage** formats commands and events that are specific to the service. The dispatcher invokes the binding function when the application calls **ctaFormatEvent**.

xxxFormatMessage is passed a command or event, a buffer, buffer size, and an indent character. **xxxFormatMessage** converts the command or event to ASCII and places the ASCII string into the supplied buffer. The indent character is prepended to each line of the ASCII string.

The binding function formats the command or event appropriately.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-----------------------------------------------------------
 tikFormatMessage
    - translate commands and events to ASCII
    - use tikGetText to help in the translation
 -----------------------------------------------------------*/
STATIC DWORD NMSAPI tikFormatMessage( DISP_MESSAGE *pmsg,
                                      char         *s,
                                      unsigned     size,
                                      char         *indent)
{
    char *text;
    char  tmp[1024] = "";
    DWORD cnt;

  /* Needed by Natural Access provided error logging service */
    CTABEGIN ("tikFormatMessage");
    ASSERT ( s != NULL && size != 0 );

    /* Switch statement returns pointer to static text. */
    text = (char*)tikGetText(pmsg->id);
    if (CTA_IS_EVENT(pmsg->id))
    {
       if( text == NULL )
       {
          sprintf( tmp, "%sUnknown TIK Event: *** (%08X)",
                   indent, pmsg->id );
       }

       else
       {
          DISP_EVENT *evt = (DISP_EVENT *)pmsg;
          char *reason = (char *)tikGetText(evt->value);

          sprintf(
       tmp, "%sEvent: %s  *** (%08X)  Reason: %s *** (%08X)\n",
       indent, text, pmsg->id, reason, evt->value );
       }
    }
    else /* command */
    {
        if( text == NULL )
        {
            sprintf( tmp, "%sUnknown TIK Command: *** (%08X)\n",
                     indent, pmsg->id );
        }
        else
        {
           DISP_COMMAND *cmd = (DISP_COMMAND *)pmsg;
           sprintf( tmp, "%sCommand: %s  *** (%08X) ",
                    indent, text, pmsg->id );
           switch ( pmsg->id )
           {
              case TIKCMD_START:
              {
                 TIK_START_PARMS  start;
                 DWORD            ret;
                 char             buffer[128];

                 /* Get user specified start parameters. */
                 if ( cmd->size1 == 0 && cmd->dataptr1 == NULL )
                 {
                    /* Command is using system default start
                       parameter.  Get system default start
```

```
                        parameter. */
                          strcat( tmp, "(System defaults) " );
                      ret = dispGetParms( pmsg->ctahd,
                              TIK_START_PARMID,
                              &start, sizeof(TIK_START_PARMS) );
                      if ( ret != SUCCESS )
                      {
                         return ret;
                      }
                   }
                    else
                   {
                /* Get application specified start parameters. */
                      start = *(TIK_START_PARMS *)(cmd->dataptr1);
                   }
                   sprintf(buffer,
                           "NumTicks = %d, Frequency = %d.\n",
                           start.NumTicks, start.Frequency );
                   strcpy( tmp, buffer );
                   break;
                }
                case TIKCMD_STOP:
                case TIKCMD_GET_CONTEXT_INFO:
                default:
                {
                   strcat( tmp, "\n" );
                   break;
                }
             }  /* end switch */
          }  /* end if text != NULL */
    }   /* end command processing */
    cnt = ((strlen( tmp )+1) > (size)) ? (size) : (strlen( tmp ) + 1);

    strncpy( s, tmp, cnt );
    s[cnt-1] = '\0';

    return SUCCESS;

}   /* end tikFormatMessage() */
```

Trace handling

In the service manager binding functions, support for trace handling involves:

- Setting the local tracemask
- Converting trace records to ASCII

Setting local TraceMask

An application sets the local tracemask with a call to **ctaSetTraceLevel**. As part of the call to **ctaSetTraceLevel**, an application specifies:

- The context handle
- The service name of the specific service whose local tracemask is to be set
- The tracemask value itself (for example, CTA_TRACEMASK_DRIVER_CMDS)

The dispatcher invokes **xxxSetTraceLevel** for the service based upon the service name and context handle. The implementation of **xxxSetTraceLevel** simply sets the tracemask associated with the context handle to the passed in value.

tik service

excerpt from <<<< *tikbnd.c* >>>>

```
*-------------------------------------------------------
 tikSetTraceLevel    - update local tracemask
 -------------------------------------------------------*/
STATIC DWORD NMSAPI tikSetTraceLevel( CTAHD ctahd,
                                      void *mgrcontext,
                                      unsigned svcid,
                                      unsigned tracemask )
{
   TIK_CHANNEL_OBJECT *ptikChannel = TIK_CHANNEL_OBJECT *)NULL;
  /* Needed by Natural Access provided error logging service */
   CTABEGIN("tikSetTraceLevel");

   ASSERT( svcid == TIK_SVCID );

   /* Set service context specific tracemask. */
   ptikChannel->tracemask = tracemask;

   return SUCCESS;
}  /* end tikSetTraceMask() */
```

Converting trace records to ASCII

Each trace record logged by the dispatcher or by a Natural Access service is logged in binary format. In order for the *ctdaemon* to print out a human readable string corresponding to a trace record, **xxxFormatTraceBuffer** is called to convert the binary trace record to an ASCII equivalent. This binding function is called with the *ctdaemon* for display purposes. It is not called in a Natural Access application.

xxxFormatTraceBuffer uses the trace tags to determine which trace record to generate. See *Defining trace tags* on page 46 for information about defining trace tags.

tik service

<<<< excerpt from *tikbnd.c* **>>>>**

```
/*-------------------------------------------------------
  tikFormatTraceBuffer
    - convert binary trace messages to ASCII
    - note that only TIK specific trace buffers will be
      converted!
 -------------------------------------------------------*/
STATIC DWORD NMSAPI tikFormatTraceBuffer( unsigned tracetag,
                                          void     *inbuffer,
                                          unsigned insize,
                                          char     *outbuffer,
                                          unsigned outsize )
{
   DWORD ret = SUCCESS;
   DWORD size;
   char  tmp[512] = "";

  /* Needed by Natural Access provided error logging service */
   CTABEGIN("tikFormatTraceBuffer");
    switch (tracetag)
   {
      case TIK_TRACETAG_CMD:
      {
         TIK_CMD_COM_OBJ *cmd = (TIK_CMD_COM_OBJ *)inbuffer ;

         ASSERT( insize >= sizeof(TIK_CMD_COM_OBJ) );
         sprintf( tmp,
         "TIK Service ClientID(0x%08x) Command:(%s) Data1=(%d) "
         "Data2=(%d) Data3=(%d) TimeStamp=(0x%08X).\n",
         cmd->client_id,
          tikTranslateCmdRsp(cmd->command),
         cmd->data1, cmd->data2,
         cmd->data3, cmd->timestamp );
         break;

      case TIK_TRACETAG_RSP:
      {
          TIK_RSP_COM_OBJ *rsp = (TIK_RSP_COM_OBJ *)inbuffer ;

          ASSERT(insize >= sizeof(TIK_RSP_COM_OBJ));
          sprintf( tmp,
          "TIK Service ClientID(0x%08X) Response:(%s) Reason=(%s) "
          "Channel=(%d) TikSeq=(%d) SvrMsg=(%s)"
          " TimeStamp=(%x).\n", rsp->client_id,
          tikTranslateCmdRsp( rsp->response ),
          tikTranslateCmdRsp( rsp->reason ),
          rsp->channel, rsp->seq,
          rsp->msg, rsp->timestamp );
          break;
      }
   case TIK_TRACETAG_BIT0:
      {
         STATE_TRAN_CTCX *statectcx = ( STATE_TRAN_CTCX *)inbuffer;

         ASSERT( statectcx->size >= sizeof(STATE_TRAN_CTCX) );

         sprintf( tmp,
         "TIK Service ClientID(0x%08X) PrevState:(%s) CurrentState(%s) "
         "Agent(%s).\n", statectcx->ptikChannel, statectcx->prev_state,
         statectcx->curr_state, statectcx->agent );
         break;
      }
      default:
      {
         ret = CTAERR_NOT_IMPLEMENTED;
```

```
         break;
      }

   }  /* end switch */

   size = (strlen( tmp ) + 1);
   if ( outsize < size )
   {
      strncpy(outbuffer, tmp, (outsize));
      outbuffer[outsize - 1] = '\0';
   }
   else
   {
      strcpy( outbuffer, tmp );
   }
   return ret;
}  /* end tikFormatTraceBuffer() */
```

Event processing

The **xxxProcessEvent** binding function processes events from other services, not events from the managed resource. (Managed resource events are handled with **xxxFetchAndProcess**.) Events are processed by:

- Identifying the event
- Processing the event
- Generating a new event to send to client application

xxxProcessEvent is executed when **dispQueueEvent** is called by another service.

Additional binding functions

The following function is a placeholder binding function for future Natural Access enhancements:

Function	Description
xxxGetFunctionPtr	Gets a pointer to a new binding function.

Set the entries for these functions in the CTASVCMGR_FNS data structure to NULL.

10 Implementing the service

Service implementation overview

Service implementation includes the following:

- Functions for each service API function.
- Functions for handling events from the managed resource (if necessary).
- Tracing support functions.

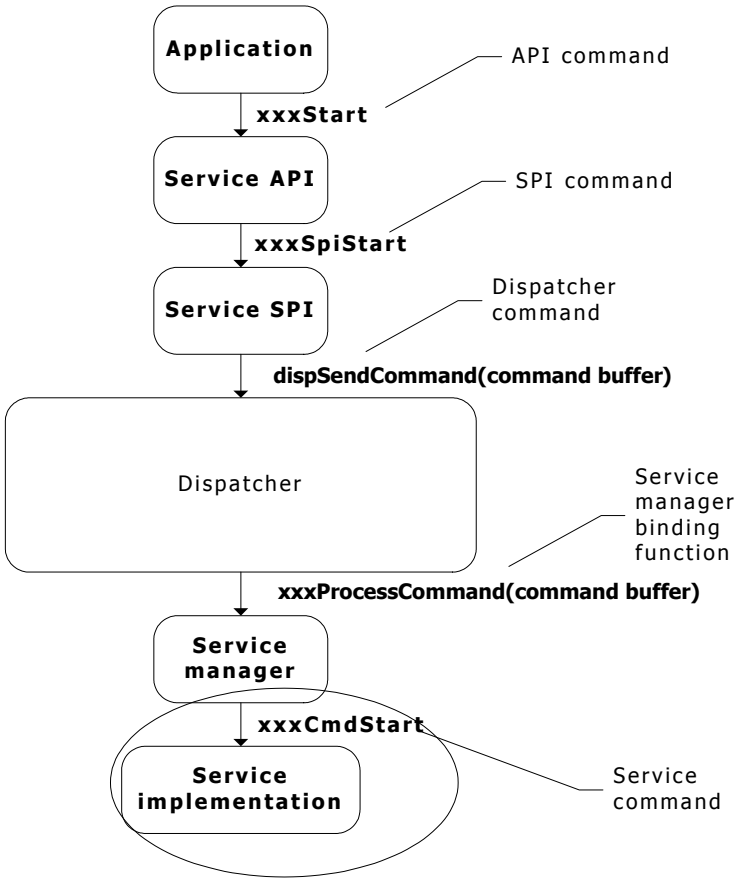
Create the following files when implementing the service:

File	Description
xxxcmd.c	Implements the service functions that interface to the managed resource.
xxxsys.h	Header file containing internal data structures, function prototypes, and definitions.

The service can create additional files as required. For example, the sample tik service contains *tikutils.c* and *tikcomms.c*.

Implementing service functions

For each service API call there must be an associated service implementation function. When the service manager function **xxxProcessCommand** receives the command from the dispatcher, it determines the associated service implementation function to call.



Service command

The service implementation functions perform whatever processing needs to be done. The service can call dispatcher functions and binding functions if necessary. Typical dispatcher functions invoked are:

- **dispGetParms** is used to access the service's default parameters.
- **dispQueueEvent** is used to enqueue a Natural Access DISP_EVENT that was manually allocated and filled in.
- **dispMakeAndQueueEvent** is a wrapper on top of **dispQueueEvent** that also allocates the DISP_EVENT structure.

If a service needs to invoke the functionality of another service, it does so with the service's SPI functions.

Service implementation functions are restricted from calling Natural Access API functions. Natural Access API functions do not operate appropriately if invoked from the service implementation.

Arguments to service implementation functions

The service implementation function receives the DISP_COMMAND message buffer. It unmarshals the function arguments and performs the processing required for the command.

Note: Because services call the SPI functions, it is recommended that the source of the command (DISP_COMMAND addr.source) be preserved (possibly in the ***mgrcontext***) for asynchronous functions. This information is needed later for proper delivery of the associate event.

Once the processing is complete, values for any out arguments must be marshalled back into the DISP_COMMAND message buffer and the proper return code set.

If there are no out arguments and successful completion occurred, return SUCCESS. If there are out arguments and successful completion occurred, return SUCCESS_RESPONSE. If there is an error, return the appropriate error code.

The SUCCESS_RESPONSE return value is used when data is returned to the caller from a synchronous function call. This data is returned in the originating DISP_COMMAND structure passed into the service implementation function. When the Natural Access dispatcher receives SUCCESS_RESPONSE as a return from **xxxProcessCommand**, it adds information to indicate that there is an immediate command response (not a separate new command) and that the message must be copied back over the network to the service interface. The dispatcher also translates the SUCCESS_RESPONSE return value into a SUCCESS return value to the service interface.

Handling service parameters as function arguments

Applications use **ctaGetParms** to retrieve default parameter structures to pass to API functions. Applications may modify the defaults retrieved and pass the new structure to the API or they may pass NULL to tell the service that it should use the context defaults.

If a service implementation function is passed a NULL value for the parameter value, call **dispGetParms** to retrieve the context default values. If a service needs to call another service with its SPI, it can use any parameter passed to it from its SPI/API to determine how to call the next service's SPI. It can also require a call to **dispGetParms** to retrieve the defaults and modify the values before calling the SPI, or it can elect to pass a NULL pointer so that the next service will retrieve its own default parameters.

Note: The modification is only valid for the current SPI call when retrieving and directly modifying parameter structure values to be passed to another service's SPI. To modify parameter defaults permanently, call **dispSetParmByName**.

Default values are only set the first time that a service is defined for an entire system. If two services with the same service interface, but different service implementations, are registered with the dispatcher, only the defaults of the first service to be registered are used.

Services with the same service ID must have the same standard parameter descriptors. They must define the same structures including the fields, formats and units. The dispatcher verifies that matching services register the same standard parameters. An error returns if this is not the case.

tik service

<<<< excerpt from *tikcmd.c* **>>>>**

```
/*-------------------------------------------------------------
  tikCmdStartTimer
    - Implementation function for tikStartTimer
  -------------------------------------------------------------*/
DWORD tikCmdStartTimer( TIK_CHANNEL_OBJECT *ptikChannel, DISP_COMMAND* m )
{
   TIK_CMD_COM_OBJ    cmd = { 0 };
   TIK_START_PARMS    start;
   DWORD              ret = SUCCESS;

   if ( ptikChannel == NULL || m == NULL )
   {
      return CTAERR_BAD_ARGUMENT;
   }

   /*  Check channel owner.  */
   if ( ptikChannel->owner != CTA_NULL_SVCID &&
        ptikChannel->owner != m->addr.source )
   {
      return TIKERR_OWNER_CONFLICT;
   }

   /*  channel state must be either IDLE or STARTED. */
   if ( !(ptikChannel->state == CHANNEL_TIMER_IDLE ||
          ptikChannel->state == CHANNEL_TIMER_STARTED) )
   {
      return CTAERR_INVALID_SEQUENCE;
   }


   /* Check if this function call included explicit
    * TIK_START_PARMS structure.
    *  - If present, use passed in structure
    *  - If not present (i.e., NULL passed), get the default
    *    TIK_START_PARMS structure via dispGetParms and use it.
    */
 if ( m->size1 == 0 && m->dataptr1 == NULL )
   {
      if( (ret=
           dispGetParms( ptikChannel->ctahd,
                         TIK_START_PARMID,
                         &start,
                         sizeof(TIK_START_PARMS) )) != SUCCESS )
      {
         return ret;
      }
   }
   else
   {
      DWORD size = ( m->size1 < sizeof(TIK_START_PARMS) )
                   ? m->size1
                   : sizeof(TIK_START_PARMS);
      memcpy( &start, m->dataptr1, size );
   }
 /* Prepare command (message buffer) to send to TICK server. */
   cmd.size      = sizeof( TIK_CMD_COM_OBJ );
   cmd.timestamp = time(NULL);
```

```
   cmd.client_id = ptikChannel->channelId;
   cmd.command   = TIKSVR_START;
   cmd.data1     = ptikChannel->channelId;
   cmd.data2     = start.Frequency;
   cmd.data3     = start.NumTicks;
 /* Update channel_info structure. */
   ptikChannel->ChannelInfo.requested += cmd.data3;

   /* If CTA_TRACEMASK_DEBUG_BIT0 is enabled,
    * log state transition information.
    */
   if ( TRACEMASK_BIT0_SET(ptikChannel->tracemask) )
   {
      ret = tikLogStateTransition( ptikChannel,
                                   ptikChannel->state,
                                   CHANNEL_TIMER_STARTED,
                                   "tikCmdStartTimer" );
   }
   /* Set channel state to STARTED. */
   ptikChannel->state = CHANNEL_TIMER_STARTED;

   /* Set channel owner.  Also used as reply address. */
   if ( ptikChannel->owner == CTA_NULL_SVCID )
   {
      ptikChannel->owner = m->addr.source;
   }

   if( (ret= tikSendClientCommand( ptikChannel, &cmd )) != SUCCESS )
   {
      return ret;
   }

   /* Expecting ticks, queue an asynchronous read request. */
   if( (ret=tikReadServerResponse( ptikChannel )) != SUCCESS )
   {
      return ret;
   }

   ptikChannel->ChannelInfo.calls++;

   return SUCCESS;

}  /* end tikCmdStartTimer() */
```

Processing events from the managed resource

By convention, **xxxFetchAndProcess** is the name for the managed resource event handler in a service implementation. This is a callback function that is invoked with the dispatcher when the associated wait object fires.

The event handling initialization takes place in either **xxxAttachServiceManager** (for multiplexed managed resources) or **xxxOpenServiceManager** (non-multiplexed managed resources). During initialization, an operating system-specific wait object is registered along with a pointer to **xxxFetchAndProcess** and a pointer to a data structure that can be used to identify and process the incoming event. **xxxFetchAndProcess** is where an incoming raw event from the managed resource can be consumed or forwarded (either to another Natural Access service or to a Natural Access application). Understanding the Managed Resource discusses how to determine which events should be consumed and which should be forwarded (or translated and forwarded).

As part of identifying the event, **xxxFetchAndProcess** must first use the wait object to retrieve the actual raw event. It must then analyze the contents of the event and perform the appropriate processing.

tik service

tikFetchAndProcess gets the raw event and checks it to determine event type. It then calls the correct helper function to process that event. Since the tik server is a multiplexed managed resource, it demultiplexes the event by using the client id value embedded in the event.

<<<< excerpt from *tikutils.c* **>>>>**

```
/*-------------------------------------------------------
  Event Processing from TICK Server
    - Callbak invoked when a corresponding registered wait
      object is signalled.
        ctaqueuehd:  Queue on which event are to be queued.
        waitobj:     Pointer to wait object that was signalled.
        arg:         Pointer to DEVICE object (which was
                     registered by tikAttachServiceManager as
                     a "queuecontext").
  -------------------------------------------------------*/
DWORD NMSAPI tikFetchAndProcess( CTAQUEUEHD    ctaqueuehd,
                                 CTA_WAITOBJ  *waitobj,
                                 void         *arg ){
    TIK_DEVICE_OBJECT     *ptikDevice  = (TIK_DEVICE_OBJECT *) arg;
    TIK_CHANNEL_OBJECT    *ptikChannel = NULL;
    TIK_RSP_COM_OBJ       *rsp = &ptikDevice->response;
    TSIIPC_COMMAND_STATUS  status = { 0 };
    DWORD                  ret = SUCCESS;

    /* Complete IPC read. */
    if ( tikCompleteReadServerResponse( ptikDevice )
         != SUCCESS )
    {
       return ret;
    }

    /* Get Channel object pointer by indexing into the
     * channelTbl array using the client_id field.
     */
    if( rsp->client_id < 0 || rsp->client_id > 9 ||
    (ptikChannel=ptikDevice->channelTbl[rsp->client_id]) == NULL)
    {
```

```
       return TIKERR_UNKNOWN_SERVER_RESPONSE;
    }

    /* Log trace for server response if requested. */
    if ( *tikTracePointer & CTA_TRACEMASK_DRIVER_EVENTS )
    {
       if( (ret=dispLogTrace( ptikChannel->ctahd,
                              TIK_SVCID,
                              CTA_TRACE_SEVERITY_INFO,
                              TIK_TRACETAG_RSP,
                              (void *)(rsp),
                              sizeof(TIK_RSP_COM_OBJ) )) != SUCCESS )
       {
          return( ret );
       }
    }
/* Process incoming message from TICK Server.*/
    switch ( rsp->response )
    {
       case TIKSVR_OPEN_CHANNEL_RSP:
       {
          ret = tikProcessOpenChannelResponse( ptikChannel );
          break;
       }
       case TIKSVR_STOP_TIMER_RSP:
       {
          ret = tikProcessStopTimerResponse( ptikChannel );
          break;
       }
       case TIKSVR_TICK_RSP:
       {
          ret = tikProcessTickResponse( ptikChannel );
          break;
       }
       default:
       {
          /* Unknown server response. */
          ASSERT(FALSE);
          break;
       }
    }

    return ret;

}   /* end tikFetchAndProcess() */
```

To forward (or translate and forward) the raw event to another Natural Access Service or to the Natural Access application, use either **dispMakeAndQueueEvent** or **dispQueueEvent**.

tik service

This is just one of the event processing functions called by **tikFetchAndProcess**. For convenience, we have chosen to use **dispMakeAndQueueEvent**.

<<<< excerpt from *tikutils.c* **>>>>**

```
/*-------------------------------------------------------*/
DWORD tikProcessOpenChannelResponse( TIK_CHANNEL_OBJECT *ptikChannel )
{
  TIK_RSP_COM_OBJ *rsp = &(ptikChannel->ptikDevice->response);
  DWORD            prev_state = ptikChannel->state;
  DWORD            reason;
  DWORD            ret;

   ptikChannel->ChannelInfo.calls     = 0;
   ptikChannel->ChannelInfo.ticked    = 0;
```

```
   ptikChannel->ChannelInfo.requested = 0;
   ptikChannel->ChannelInfo.remaining = 0;
   ptikChannel->ChannelInfo.ms        = 0;
   strcpy( ptikChannel->ChannelInfo.tick_string, "" );

   /* Translate reason code from incoming message to
    * associated Natural Access reason.
    */
   switch ( rsp->reason )
   {
      case TIKSVR_CHANNEL_OPENED:
      {
         ptikChannel->state = CHANNEL_TIMER_IDLE;
         reason             = CTA_REASON_FINISHED;
         break;
      }
      case TIKSVR_CHANNEL_ACTIVE:
      {
         ptikChannel->state = CHANNEL_NOT_ALIVE;
         reason             = TIK_REASON_CHANNEL_ACTIVE;
         break;
      }
      case TIKSVR_CHANNEL_INVALID:
      {
         ptikChannel->state = CHANNEL_NOT_ALIVE;
         reason             = TIK_REASON_INVALID_CHANNEL;
         break;
      }
      default:
      {
         ptikChannel->state = CHANNEL_NOT_ALIVE;
         reason             = TIK_REASON_UNKNOWN_SERVER_REASON;
         break;
      }
   }
   /*  If CTA_TRACEMASK_DEBUG_BIT0 is enabled,
    *  log state transition information.
    */
   if ( TRACEMASK_BIT0_SET(ptikChannel->tracemask) )
   {
      char   txt[128] = "TIKSVR_OPEN_CHANNEL_RSP : ";
      char  *reasontxt = tikTranslateCmdRsp( rsp->reason );
      strcat( txt, reasontxt );
      if ( (ret=tikLogStateTransition( ptikChannel,
                                       prev_state,
                                       ptikChannel->state,
                                       txt )) != SUCCESS )
      {
         return ret;
      }
   }
/* Create and enqueue open service DONE event. */
   ret = dispMakeAndQueueEvent( ptikChannel->ctahd,
                                CTAEVN_DISP_OPEN_SERVICE_DONE,
                                reason,
                                TIK_SVCID,
                                CTA_SYS_SVCID);

   return( ret );

}  /* end tikProcessOpenChannelResponse() */
```

DISP_EVENT structure

Service implementations may need to generate events:

- To deliver unsolicited information to the application.
- To complete asynchronous API command processing.

An event contains the following information:

```
typedef struct _DISP_EVENT
{
    DWORD    id;          /* Event code (and source service id)     */
    CTAHD    ctahd;       /* CT Access context handle               */
    DWORD    timestamp;   /* Timestamp                              */
    DWORD    userid;      /* User id (defined by ctaCreateContext)  */
    DWORD    size;        /* Size of buffer if buffer != NULL       */
    void     *buffer;     /* Buffer pointer                         */
    DWORD    value;       /* Event status or event-specific data    */
    DWORD    extra;       /* Extra internal event field             */
    unsigned  objHd;      /* Service object handle                  */
    MESSAGE_ADDR addr;    /* Source/Destination service id          */
} DISP_EVENT;
```

The dispatcher fills in the timestamp field if it is set to zero. The dispatcher sets the userid field based upon information stored within the context.

The size field can be used to pass service-specific data if no buffer is being passed with the event.

If the event is being generated for completion of an asynchronous API command, (for example: CTAEVN_something_DONE), or if a data buffer is being passed with the event, then special buffer flags must be set in the size field as explained in Unsolicited events containing buffered data and DONE events associated with asynchronous API functions.

See the Buffer management summary section for a summary of buffer management.

Unsolicited events containing buffered data

If an event contains buffered data, you must indicate to the dispatcher whether the event's buffer is static or dynamic. To do this, set one of two flags in the event's size field:

```
DISP_EVENT evt = {0};

// STATIC
static char svcBuffer[  MAX_BUFF_SIZE ];
evt.size   = MAX_BUFF_SIZE | CTA_INTERNAL_BUFFER_STATIC;
evt.buffer = svcBuffer;

// DYNAMIC
evt.size   = MAX_BUFF_SIZE | CTA_INTERNAL_BUFFER;
dispAllocBuffer( &evt.buffer, MAX_BUFF_SIZE );

// Data that is to be delivered in the event can
// now be copied into the buffer.
```

If the service implementation sets the CTA_INTERNAL_BUFFER flag in the event's size field, Natural Access or the application ensures the proper release of this buffer. The service implementation does not need to perform this task. This process allows a service manager to remain unaware of whether the context on which the service has been opened is being shared. Also, if the context is shared, the service manager can remain unaware of when the Natural Access dispatcher has completed delivery of the event buffer to all attached applications.

Service implementations should use the Natural Access dispatcher function **dispAllocBuffer** for the event's buffer allocation. Because service implementations can be written in either C or C++, the use of this function ensures that the appropriate function is subsequently used for freeing the buffer.

DONE events associated with asynchronous API functions

Parameters to certain service API functions can be asynchronous input buffers. That is, although the function returns immediately, the input buffer is still considered in use by the service implementation until the service implementation generates an associated service DONE event (XXXEVN_SOMETHING_DONE).

Due to the client-server architecture, it is a requirement of the service implementation that this DONE event contain a flag in the size field allowing the underlying Natural Access transfer buffers to be freed. Even though a buffer of data is not being sent in the DONE event, the value of the buffer field in the event structure must be set to the same value originally sent in the DISP_COMMAND message as the asynchronous input buffer.

```
xxxCmdDoSomething( void *mgrcontext, DISP_COMMAND *cmd )
{
     // Assume dataptr1 is an asynchronous input buffer.
     // Save input buffer pointer for later
     mgrcontext->cmd.bufferSize  = cmd.size1;
     mgrcontext->cmd.inputBuffer = cmd.dataptr1;

     // etc.
}

------------------------------- Later, when operation is complete

xxxFetchAndProcess( CTAQUEUEHD qhd, CTA_WAITOBJ *waitobj, void *arg;
{
     XXXMGRCONTEXT *mgrcontext = (XXXMGRCONTEXT *) arg;
     CTA_EVENT evt = {0};

     if( mgrcontext->cmd.inputBuffer )
     {
          evt.size = mgrcontext->cmd.bufferSize | CTA_EVT_ASYNC_BUF_IN;
          ev->buffer = mgrcontext->cmd.inputBuffer;
     }

     // etc.

     dispQueueEvent( &evt );
}
```

Note: No special flag needs to be set in the size field when the event's buffer is an asynchronous output buffer.

Generating trace records

In the service implementation functions, the local and global tracemasks are checked to see what trace record categories are active. For active trace record categories, invoke **dispLogTrace** with the following arguments to generate and log a trace record:

- A context handle.
- The service ID.
- A ***traceseverity*** flag indicating whether this is a warning, an error, or an informational trace record. The acceptable values are:
    - CTA_TRACE_SEVERITY_INFO
    - CTA_TRACE_SEVERITY_WARNING
    - CTA_TRACE_SEVERITY_ERROR
- The tracetag.
- A buffer containing trace data. The trace data can be anything that is useful to be presented to a developer during a debug session. The trace data can be in binary or ASCII; there are no restrictions. It is up to **xxxFormatTraceBuffer** to convert this portion of the trace record to human readable (ASCII) format.
- The size of the buffer.

To determine whether or not to invoke **dispLogTrace**, check both the global and local tracemasks to see if the appropriate category is set. If so, invoke **dispLogTrace**.

tik service

One of the things being traced is the set of commands being sent from the tik service to the tik server (when the tracemask has the CTA_TRACEMASK_DRIVER_COMMANDS category set).

<<<< excerpt from *tikcomms.c* **>>>>**

```
/*-------------------------------------------------------------
  Utility function to send a command via IPC synchronously.
  A Natural Access error code is returned to indicate success
  status.
  -------------------------------------------------------------*/
DWORD tikSendClientCommand( TIK_CHANNEL_OBJECT *ptikChannel,
                            TIK_CMD_COM_OBJ    *cmd )
{
   int                     cnt;
   DWORD                   ret;
   TSIIPC_COMMAND_STATUS   status = {0};

   if ( cmd == NULL || ptikChannel == NULL )
   {
      return CTAERR_BAD_ARGUMENT;
   }

   /* Log trace if requested. Both the global tracemask must be
      checked as well as the local, per mgrcontext, tracemask.*/
   if ( *tikTracePointer & CTA_TRACEMASK_DRIVER_COMMANDS ||
        ptikChannel->tracemask & CTA_TRACEMASK_DRIVER_COMMANDS )
   {
      if( (ret=dispLogTrace( ptikChannel->ctahd,
                          TIK_SVCID,
                          CTA_TRACE_SEVERITY_INFO,
                          TIK_TRACETAG_CMD,
                          (void *)cmd,
                          sizeof(TIK_CMD_COM_OBJ) )) != SUCCESS )
      {
         return ret;
      }
   }
   ...
}
```

11 Service object handles

Service objects

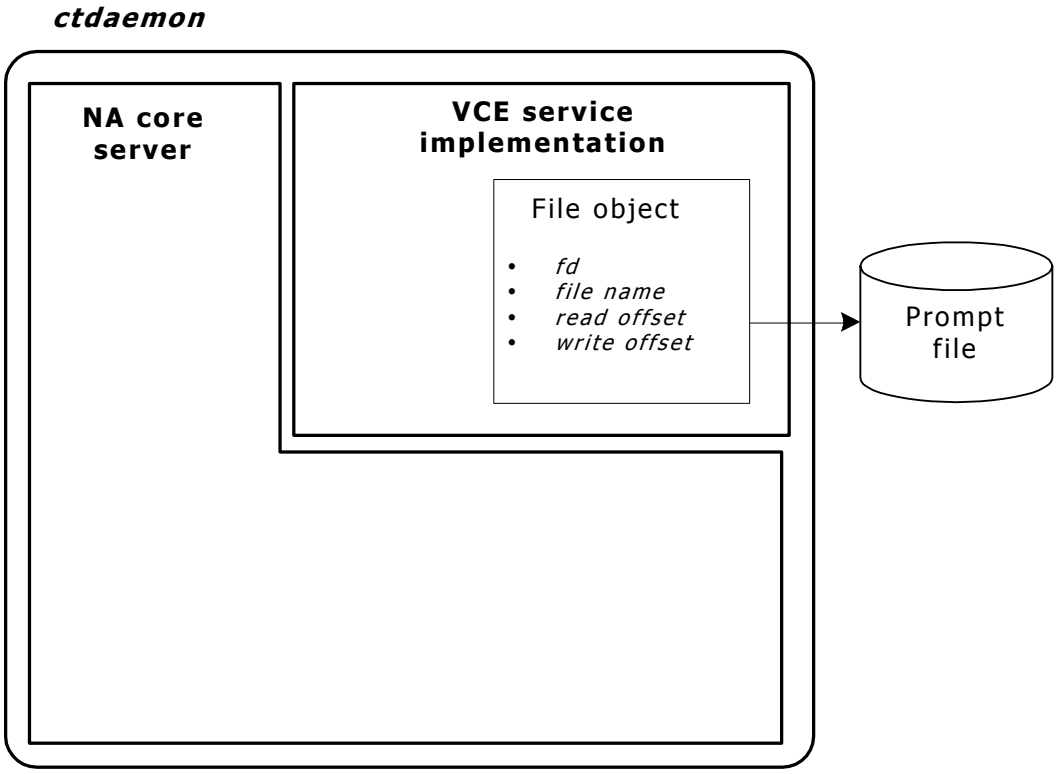
A service object is a functional unit within a service that represents a component of a managed resource. One service that makes use of service objects is the NMS Voice Message (VCE) service. The managed resource that VCE operates upon is a set of voice messages. Since voice messages can exist in either a file or in memory, VCE provides service objects to facilitate access to them depending upon their location.

Files containing one or more playable voice messages are called prompt files. Through the VCE service interface, an application can perform the following on a prompt file:

- Manipulate its contents
- Get a list of the messages it contains
- Play a message
- Determine the highest message number

To support prompt file-related interface functions, the VCE service implementation maintains bookkeeping information about the prompt file. This bookkeeping is done within a service object. Bookkeeping information includes static information such as filename and number of messages in file. It also includes dynamic information such as what message should be played next. This is analogous to the seek position in a typical operating system file operation.

The following illustration depicts the relationship of a file object to both the VCE service and to the associated prompt file.



VCE file object

Service handles

Natural Access supports two kinds of service object handles: server-side object handles and client-side object handles.

- A server-side object handle provides internal (to the service) access to a service object and resides on the server. Only one server-side object handle is associated with a service object.
- A client-side object handle provides external (that is, client) access to a service object. Each client attempting concurrent access to a specific service object has a separate client-side object handle.

This topic presents:

- Server-side object handles
- Server-side context handles
- Client-side context handles
- Client-side object handles
- Mapping server handles to client handles

Server-side object handles

Server-side object handles enable a service implementation to refer to a unique instance of one of its internal service objects. A server-side object handle can be a memory pointer, an index into an array, a hash table value, or any other indicator the service developer chooses. Only the service implementation interprets the handle to access the service object.

The service developer must take care to prevent changes to the actual value of the service handle once it has been allocated for use. For example, assume the handle was implemented as a memory pointer to the service object. Now the service object needs to be reallocated since some internal state needs to grow. Any reference to the service object using the old handle is now invalid since the handle no longer points to the service object. In this situation, it is inappropriate to use a memory pointer as the service object handle implementation. An alternate approach could be to have an array of pointers to service objects and to use an array index as the service object handle. In this case, a memory pointer to the service object can change, yet the handle remains constant.

Because an object handle is particular to a service implementation and is interpreted only by the implementation that created it, there is no need to guarantee that a handle from one service is different than a handle from another service. Though the handle values may be the same, they refer to separate, unique service objects implemented in separate, unique service instances.

Note: Do not use memory pointers as service subject handles.

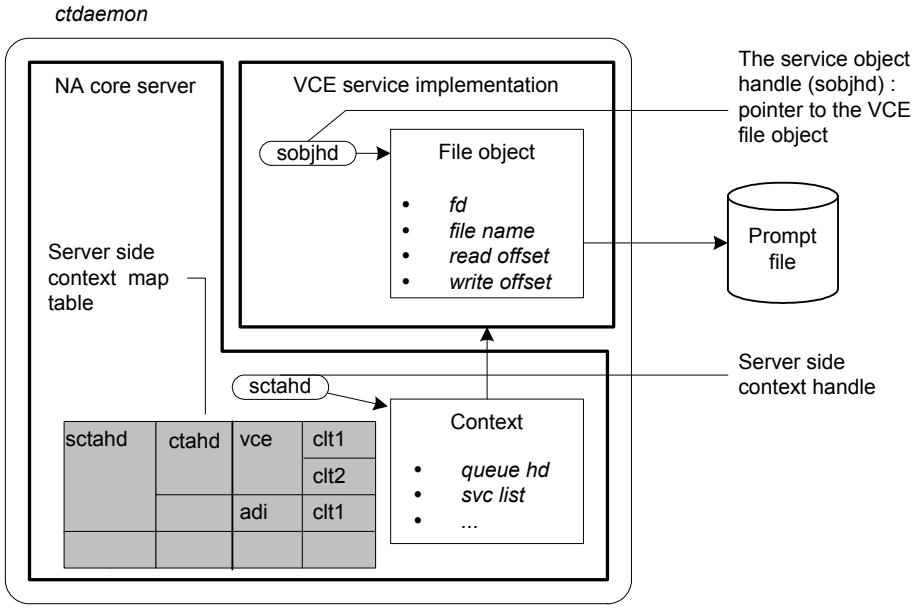
Server-side context handles

The server-side context handle is a special type of server-side service object handle. A context is equivalent to a service object for Natural Access itself. A context contains the bookkeeping information required to allow processing to occur under the Natural Access infrastructure. Some of this information includes:

- The context name
- The queue associated with the context
- A list of services opened on the context

Just as a server-side object handle uniquely identifies an instance of a service object, a server-side context handle uniquely identifies an instance of a context within the Natural Access core. The Natural Access Server (*ctdaemon*) maintains a table of all server context handles. With each of these handles, it associates the list of services opened on the context, and a list of clients using the services within the context (that is, the client-side context handles)

The following illustration shows this information. One can infer that the context entry shown in the table is being shared because two clients (clt1 and clt2) are shown using the VCE service. In this server-side table within the Natural Access core, server-side context handles are mapped to the appropriate client-side context handles.



Server side handles

Client-side context handles

When a Natural Access client application creates a context using **ctaCreateContext**, local storage must be allocated within the client's process space to hold addressing information that uniquely identifies the server-side context that the client is trying to create. In effect, this local storage acts as a proxy for the server-side context. The proxy is bound to the server-side context upon successful completion of the **ctaCreateContext** call. The proxy continues to be maintained by the Natural Access core on the client-side for as long as the context exists. Access to the proxy is with a handle. Since the information in the proxy enables remote access to the server-side context, the handle to the proxy operates as a client-side context handle. This is the CTAHD that is passed back to the caller upon successful creation of the context.

Some of the information maintained in the proxy includes:

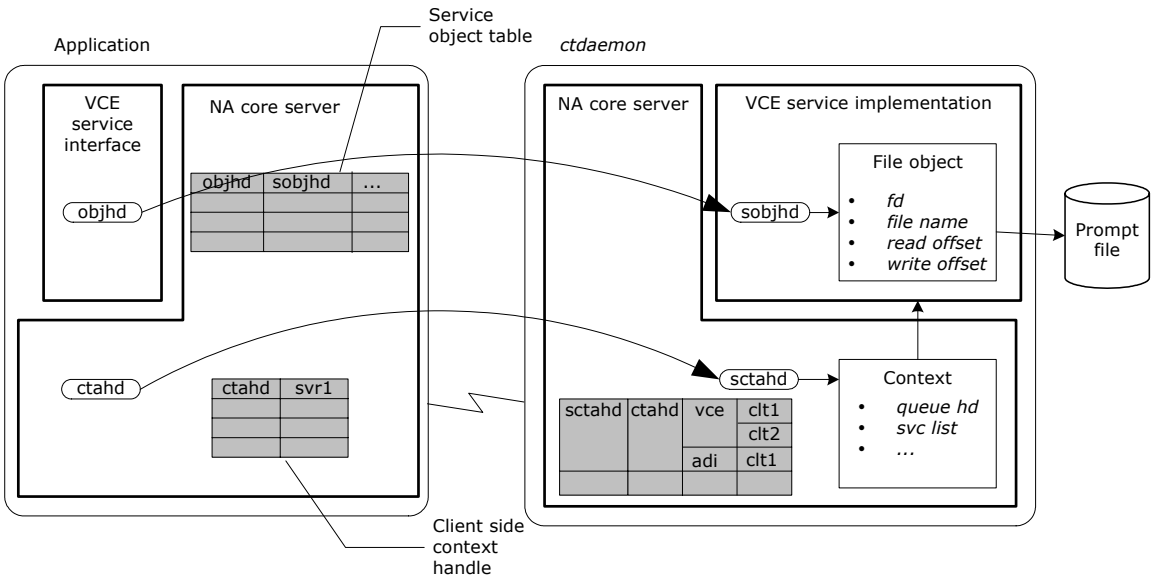
- The address of the Natural Access server containing the context
- The queue associated with the context
- The context name (if any)
- The server-side context handle

A proxy is required because a server-side context handle contains only enough information to reference a context relative to the server's process space. There is not enough inherent information within a server-side context handle to span between processes.

Client-side object handles

A proxy is also required for any service object that is to be accessed by a client. The proxy contains addressing information that uniquely identifies a service object within the distributed Natural Access system. Again, access to the proxy is through a handle; therefore the handle acts as a client-side service object handle.

These proxies are implemented within the Natural Access core as records within a mapping table. As depicted in the following illustration, this table contains records that associate client-side handles with addressing information necessary to uniquely identify server-side service objects. Creation, maintenance, and destruction of records within this table are the responsibility of the service developer as described in *Mapping server handles to client handles* on page 126.



Client side handle

Mapping server handles to client handles

Because services operate solely within the boundaries of a Natural Access context, there must be a context handle associated with every service object handle. Rather than passing in multiple handles to individual API functions, a service interface should examine its own service handles to infer the context for which the function call is being made. (That is, the caller should not have to explicitly pass the context handle ***ctahd*** if it is passing in a service-specific handle.) For example, when an application calls the function **vcePlayMessage**, it passes only a VCEHD (a VCE specific ***objhd***) that it received when it first called **vceOpenFile**. From this handle, the VCE service interface logic determines its associated context handle (***ctahd***).

Thus, if a service implementation uses service handles, it must maintain a mapping of a service object handle to its associated context handle. In addition, as already noted, service interfaces within the distributed Natural Access environment must also map client-side service handles to server-side service handles. Maintenance of this mapping table requires the use of locks to protect underlying data structures while operating in a threaded process environment. Typically, each client context is running as a separate thread within the application.

The Natural Access core provides a set of functions that eliminates having to create redundant mapping table logic. The following table lists these functions:

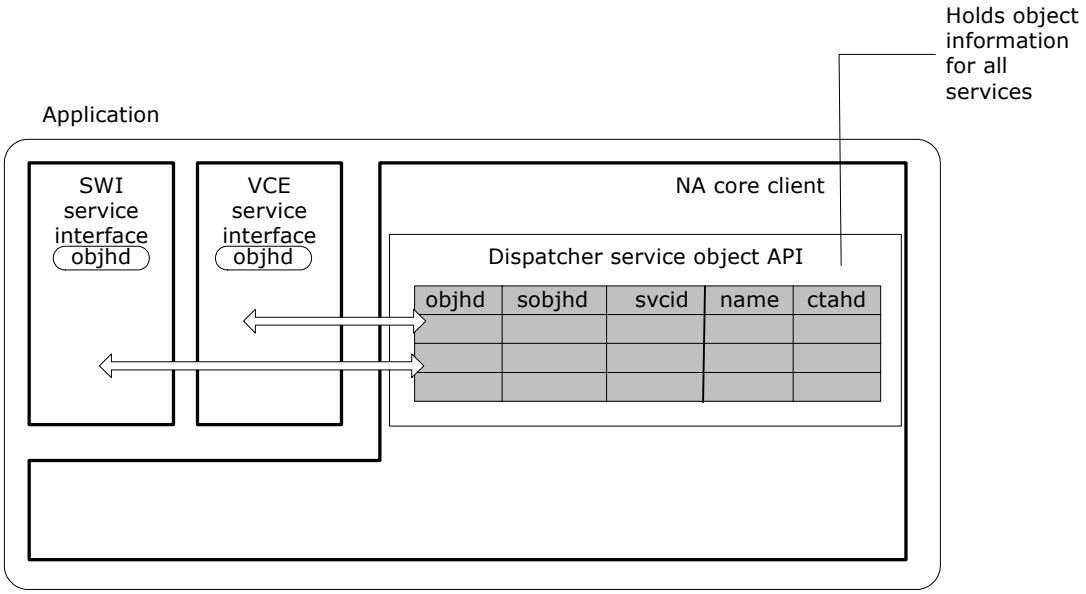
Function	Description
dispCreateHandle	Creates a client-side API handle and associates it with a server-side object handle.
dispDestroyHandle	Destroys a client-side API handle and clears its associated server-side object handle.
dispSetHandleValue	Associates a server-side object handle with a client-side API handle.
dispSetHandleValueEx	Associates a service object name with a client-side API handle.
dispGetHandleValue	Retrieves information associated with a client-side API handle: context handle, server-side service object handle, and service identifier.
dispGetHandleValueEx	Retrieves all information associated with a client-side API handle: context handle, server-side service object handle, service object name, and service identifier.
dispFindHandle	Returns the client-side handle associated with a server-side handle.
dispFindHandleByName	Returns the client-side handle associated with the name of a service object.

There is a single instance of the mapping table running in an application's process space. This mapping table contains information for all services running on all contexts within the application. Each record within the table contains the following information:

Item	Description
objhd	The client-side service object handle
sobjhd	The server-side service object handle
svcid	The unique Natural Access service identifier
name	The unique service object name
ctahd	The client-side context handle

A record in the mapping table corresponds to a proxy. The ***objhd*** is the handle returned from **dispCreateHandle**. By calling this function from within a service interface, the service is automatically provided a unique client-side service object handle that the service interface can return to the caller of its API. After creating a record in the Natural Access core's handle table, the service interface must fill the ***sobjhd*** and the ***name*** fields within the record. The ***sobjhd*** is the value denoted within the dispatcher function names.

The Natural Access core does not transparently map server-side object handles to client-side object handles on behalf of the services. This is because the Natural Access core has no inherent knowledge of service objects. All knowledge of service objects, and their associated handle(s), is encapsulated within the service itself.



Mapping server handle to client handles

The following code sample illustrates how to create handles and set associated values:

```
DWORD dtmSpiStartTrunkMonitor ( CTAHD ctahd, unsigned board,
                                unsigned trunk, DTMHD *dtmhd,
                                DTM_START_PARMS *parms, WORD source)
{
    /* (*dtmhd) returns client side handle if object created
       successfully, otherwise, returns 0 */

    DWORD               ret = SUCCESS;
    DISP_COMMAND        m = {0};
    DTM_START_PARMS_EX  px = {0};
    char                objName[ DTM_OBJ_NAME_LEN] = {0};

    px.trunk = trunk;
    px.board = board;

    m.id = DTMCMD_START_MONITOR;
    m.ctahd = ctahd;
    m.size1 = sizeof px;
    m.dataptr1 = &px;
    m.objHd = 0;
    m.addr.destination  = DTM_SVCID;
    m.addr.source       = source;

    ret = dispSendCommand(&m);

    /* if object created successfully, m.objHd contains the returned
       server side handle */

    if ( ret == SUCCESS )
    {
        ret = dispCreateHandle(ctahd, DTM_SVCID, m.objHd, dtmhd);
        if ( ret == SUCCESS )
        {
            /* Create a name for this object and save in mapping
```

```
                table. */

            _buildObjName(board, trunk, objName);
            ret = dispSetHandleValueEx((*dtmhd), m.objHd, objName);
        }
    }

    return ret;
}
```

Service object sharing

Service object sharing is the ability to allow multiple concurrent access to a single instance of a service specific object under the Natural Access framework.

For example, assume that VCE service objects are made sharable. Two applications can then open the same shared prompt file (the service object). Then one application can add messages to the prompt file, while the other application plays the messages that are added.

To share a service object, each client attach to the same context and to the same service object. This gives each client a unique client-side object handle that is associated with the same service object.

The following scenarios describe how a service object can become shared:

- Peer-to-peer: In this scenario, clients act as peers; they follow a private protocol to decide when to share service objects and which service objects to share. In this case a mechanism is needed to identify a service object to be shared, to transport that information from one peer to the other, and to attach to the underlying object. All Natural Access services support this scenario.
- Query and attach: In this scenario, a Natural Access service provides a client-callable API function to enumerate its own service objects and to attach to those objects. Very few Natural Access services support this scenario.
- Unsolicited event: In this scenario, clients share the same Natural Access context but not an associated service specific object. In this case it is entirely possible that an event containing a service object handle will be broadcast to these clients. For those clients not specifically sharing the service object, this event is an unsolicited event. Appropriate attach processing must occur before access to this service object can occur. This scenario can only be supported by asynchronous services.

Peer-to-peer sharing

This section presents:

- Generic peer-to-peer sharing
- Service-specific peer-to-peer sharing

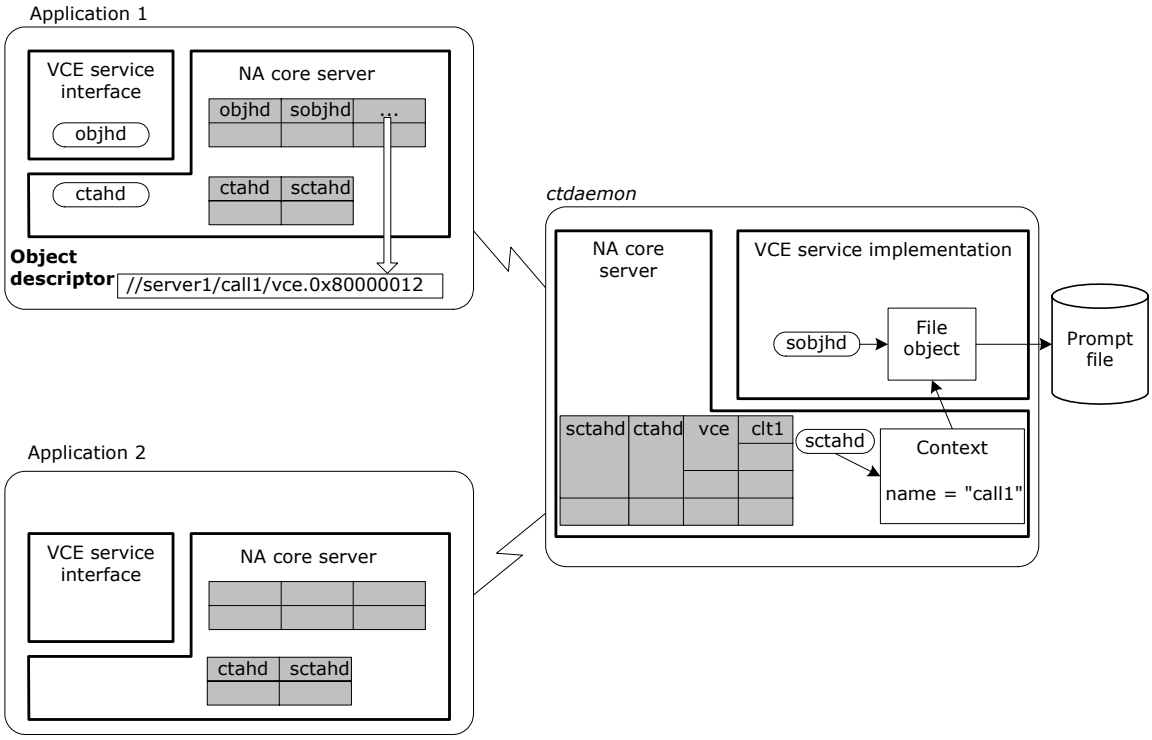
Generic peer-to-peer sharing

Clients sharing a service object as peers use a service object descriptor to specify the service object to share. The service object descriptor is a well-formed string that contains service object addressing information. This descriptor can be passed from client to client using any transport mechanism of choice. Upon receipt of the descriptor, a client can then attach to the underlying service object with an API call, adding a new record to the internal mapping table.

Converting a client handle to an object descriptor

The application should convert the client-side handle into an object descriptor that can be used to locate a particular instance of an object within the multi-server Natural Access environment.

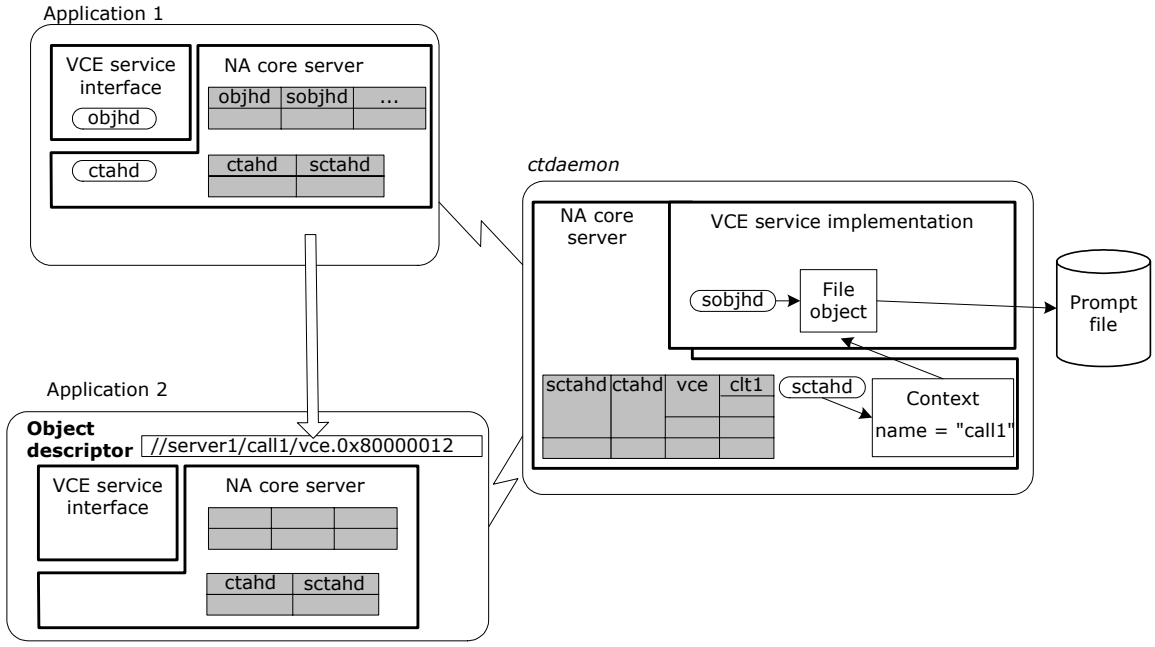
The Natural Access development environment provides application writers with **ctaGetObjDescriptor**, which retrieves the object descriptor associated with a specified object handle. The creation and interpretation of the object descriptor is handled within the Natural Access core. Service developers do not need to provide conversions functions of their own.



Converting client handle to object descriptor

Transporting the object descriptor to another client

Any application that needs to share a service object with another application requires an object descriptor. The descriptor simply identifies to the Natural Access core the particular service object that is to be shared. The Natural Access development environment provides no inherent mechanisms that transparently enable an application to transfer an object descriptor to another application. It is the responsibility of the application writer to translate a service handle into a descriptor (by invoking **ctaGetObjDescriptor**) and to pass it over to the peer as depicted in the following illustration.



Transporting object descriptor to another client

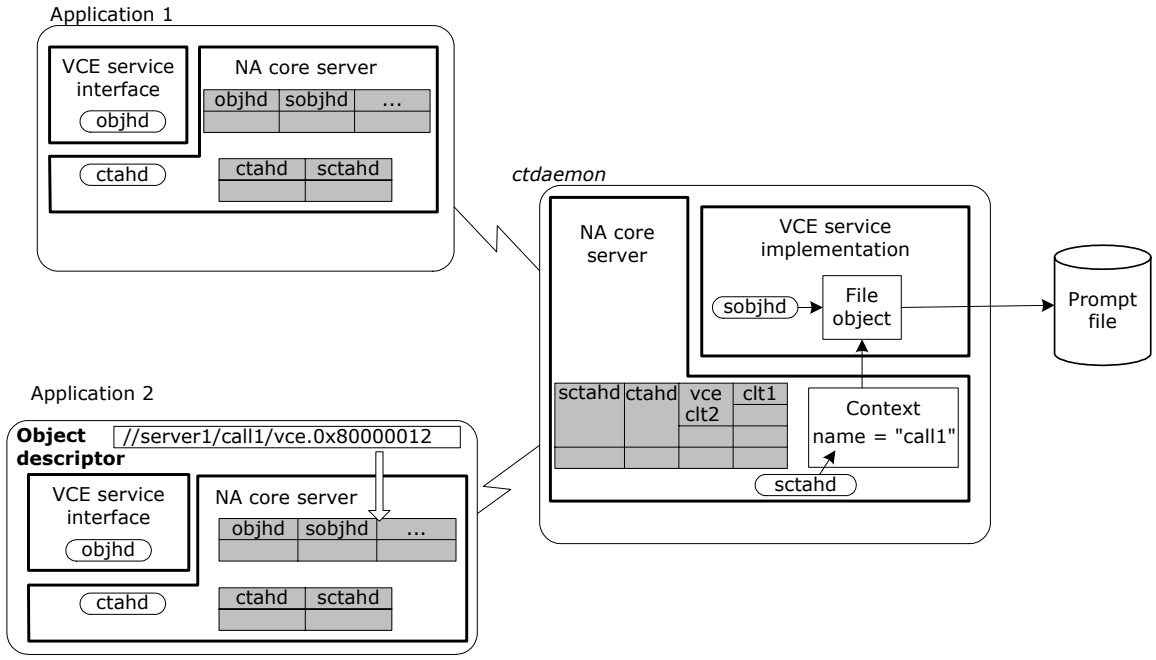
Since the descriptor is a string of characters, any of several simple mechanisms can be used to pass a descriptor to another application:

- One application writes the descriptor to a file. The second application then reads it from the file. The filename can be hard-coded into the application, or passed in as a startup parameter.
- Establish a private IPC channel (for example, using a socket or shared memory) between the applications. The descriptor can then be sent from one application to the other over this private channel using common IPC functions.
- Each application creates or attaches to an agreed upon context name (for example, **TransferCtx**). Then an application can use **ctaQueueEvent** to pass the descriptor within an application event. All clients attached to a shared context receive all events generated within the context. Therefore, the target application client receives the application event containing the object descriptor.

Converting the descriptor back to a handle and attaching objects

When an application receives the object descriptor representing a service object that is being shared with a peer application, it must then turn the descriptor into a client-side service object handle by passing the descriptor to the Natural Access core function **ctaAttachObject**. **ctaAttachObject** returns a client-side handle to the service object. The application is then able to operate on the service object using the newly assigned client-side handle just as if it were the application that originally created the service object.

Note: Before invoking **ctaAttachObject**, the application must attach to the shared context. For more information on the basic concepts of context sharing, refer to the *Natural Access Developer's Reference Manual.*



Converting descriptor to service object handle and attaching object

Service-specific peer to peer sharing

A Natural Access service can implement service object sharing using named service objects. The NMS Point-to-Point Switching (PPX) service supports the concept of named objects. Users of this service can create a connection and assign it a name. Subsequently, another application needing to alter the connection (which is a PPX service object) need only open the connection using the assigned name in its call to **ppxOpenConnection**. **ppxOpenConnection** returns a client-side service handle to the connection object. At this point, the connection object is shared.

Just as with object descriptors, users of service object names need a mechanism for knowing or agreeing to the object's name before they can share access to the object. Unlike object descriptors, there is no standard Natural Access function equivalent to **ctaAttachObject** that operates on service object names for performing attachments. Instead, the service developer must create a service specific attach-object-by-name function (**xxxAttachObject**).

Query and attach

In the query and attach service object-sharing scenario, the service developer implements a query and attaches all capability into the service API. The API allows the application to ask for the number of service objects currently existing in the running instance of the service implementation (**xxxGetNumberObjects**). After allocating adequate space, the application then calls a function to attach to all objects (**xxxAttachAllObjects**). **xxxAttachAllObjects** returns a buffer of client-side service handles that the application can use for normal operations on the service objects.

With this approach, the service interface must get a buffer of service object handles from the service implementation. These handles must then be converted into client-side service object handles within the service interface before they are passed up to the calling application.

Unsolicited event

In the unsolicited event approach, the service interface automatically creates client-side service handles when it receives service events. The service interface takes advantage of the fact that all clients attached to a shared context receive all of the service events on that context.

The service implementation side sends a service-side object handle in the Natural Access event structure CTA_EVENT. The service interface implements the API function **xxxEventProcessing**, which Natural Access automatically calls prior to passing the service's event up to the client. **xxxEventProcessing** causes the service interface to automatically create a client-side service handle (using **dispCreateHandle**) if one did not already exist for the service object handle contained within the event. (The service interface uses the Natural Access core function **dispFindHandle** for making this determination.) The modified event is then sent to the client.

This scenario allows an application to automatically deliver a client-side service handle without making any explicit calls. However, a client of the context must make a service API call that results in the generation of a service event, which is then received by all clients attached to the context. No object sharing occurs until this event is generated.

The following code sample illustrates the use of an unsolicited event for creating a client side handle using the **xxxEventProcessing** client-side binding function:

```
/*-------------------------------------------------------------------------
  dtmEventProcessing
  Client-side binding function called by Natural Access dispatcher, whereby:
        - object may be attached based on objHd field in event
        - object may be destroyed on client side based on event type
  -------------------------------------------------------------------------*/
DWORD dtmEventProcessing ( CTA_EVENT* event )
{
    DWORD shd = event->objHd;     /* server side object handle */
    char  objName[DTM_OBJ_NAME_LEN];
    DWORD ret = SUCCESS;

    /* Create a service specific name for this object based upon
       data sent in event's size field. */
    _buildObjName( event->size, objName);

    if (shd != 0)
    {
        /* Find client-side handle associated with server-side handle */
        ret = dispFindHandle(event->ctahd, DTM_SVCID, shd, &(event->objHd));
        if ( ret != SUCCESS )
        {
            /* An entry in the dispatcher's mapping table was not
               found for this object. Therefore, create a new entry.
               The event objHd field will be set to the new client-side
               object handle. */
            dispCreateHandle(event->ctahd, DTM_SVCID, shd,
                                &(event->objHd));

            /* Save the object name with the new mapping table entry,
               in addition to the server-side object handle. */
            dispSetHandleValueEx(event->objHd, shd, objName);
        }
        else
        {
            /* A client-side object handle already exists for this
               server-side object handle. If this is a DTMEVN_MONITOR_DONE
               event, the object entry in the dispatcher mapping must be
               now be removed. */
     if (event->id == DTMEVN_MONITOR_DONE)
            {
                dispDestroyHandle( event->objHd );
            }
        }
    }
    return ret;
}
```

12 Natural Access runtime control

Runtime control overview

Runtime control (RTC) within the Natural Access development environment enables Natural Access services on a context to take actions based upon events generated by other services on the context.

This topic presents:

- Conditions and actions
- RTC function summary
- Program flow

Conditions and actions

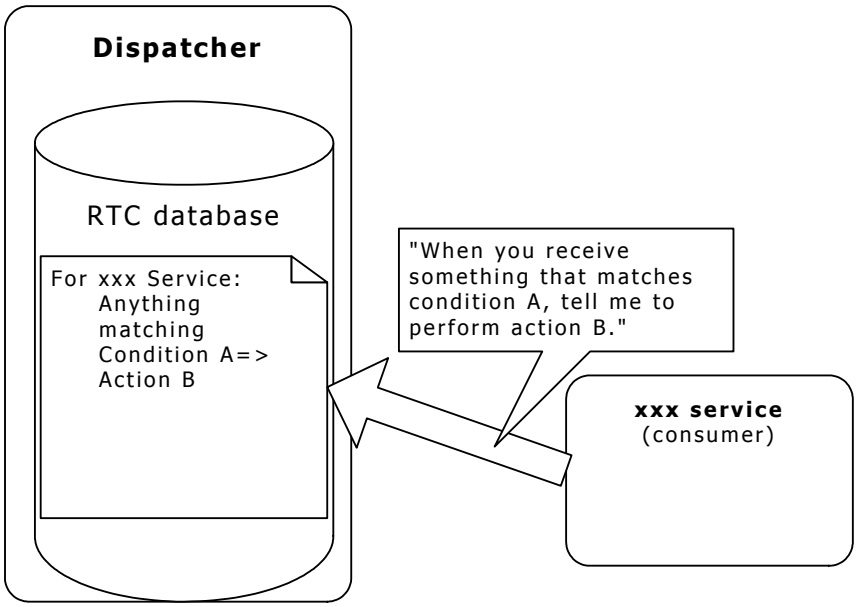
Natural Access runtime control is orchestrated by the dispatcher, which receives all events generated by service managers. A service (the consumer service) can register with the dispatcher to receive RTC notification whenever another service (the provider service) generates an event that matches a condition specified by the consumer service. With each condition, the consumer registers a corresponding action to perform when the condition is met.

Note: All services that generate events are automatically provider services. A service does not need to take any other specific action to become a provider service.

The condition can be any of the following:

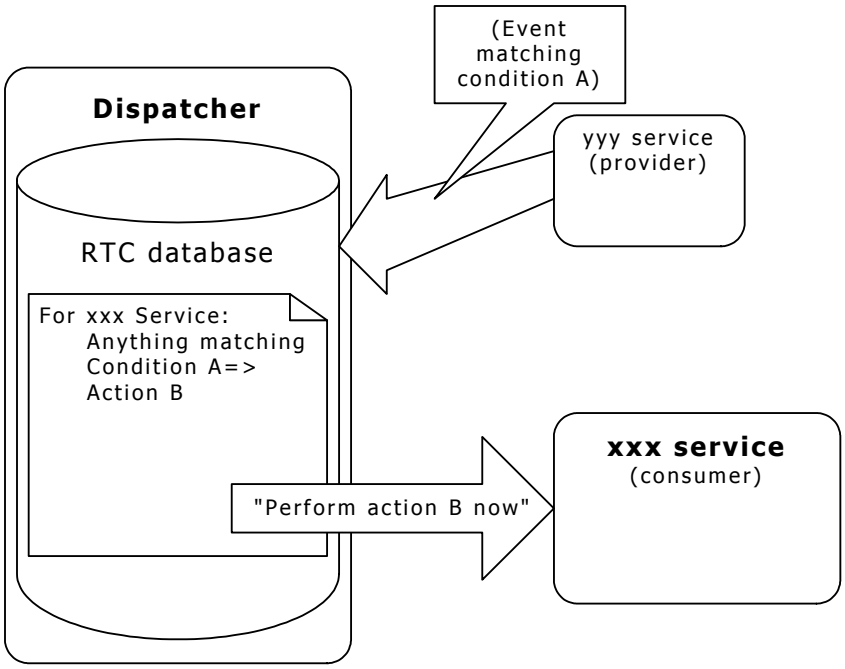
- The dispatcher receives a specific event from the provider service with a specific reason code
- The dispatcher receives a specific event from the provider service with any reason code
- The dispatcher receives any event from the provider service
- The dispatcher receives an event associated with a registered alias

The dispatcher stores the condition and action pairs in a database:



Condition and action pairs stored in database

Whenever the dispatcher receives an event, it consults the database. If the event matches a condition stored there, the dispatcher tells the registering service manager to perform the corresponding action:



Action sent to consumer service

Once it has registered its condition and action pairs in the RTC database, the consumer service does not need to take any further RTC-specific actions (except unregistering its condition and action pairs at the end of the session).

Actual runtime control operations in response to the provider service's events involve only the dispatcher and the consumer service managers. The application is not involved in RTC at all, and does not need to know that runtime control operations are taking place.

Note: Service roles (provider or consumer) are assigned on a per-event basis: a service can issue events (making it a provider service) and then receive RTC notifications (making it a consumer service) within the same session.

RTC function summary

The following dispatcher service functions are directly involved in RTC notification:

Function	Description
dispAddRTC	Registers a condition and corresponding action in the RTC database.
dispRemoveRTC	Removes a registered condition and action pair from the RTC database.
dispRegisterRTCAlias	Defines an alias for a set of service-specific events. The alias is stored in the RTC database.
dispUnregisterRTCAlias	Removes an alias stored in the RTC database.
dispGetNumRTC	Returns the number of condition and action pairs specified within a context.
dispGetNumRTCAliases	Returns the number of aliases specified within a context.
dispGetRTCAliases	Returns a list of all aliases specified within a context.
dispGetActiveRTC	Returns a list of all condition and action pairs specified within a context.

The following table lists optional binding functions for service managers that can be provided to Natural Access when a service manager is registered with the system. The dispatcher invokes both functions. The service writer determines the actions taken by the provider service when these functions are called.

Function	Description
xxxAddRTC	Called into the provider service when the consumer service calls **dispAddRTC** and a valid condition and action pair is added to the RTC database.
xxxRemoveRTC	Called into the provider service when the consumer service calls **dispRemoveRTC** and a condition and action pair is successfully removed from the RTC database.

Program flow

To register a condition and associated action, the consumer service manager invokes the dispatcher function **dispAddRTC**. With **dispAddRTC**, the service manager passes pointers to structures specifying the condition and action. Each successful invocation of **dispAddRTC** adds one condition and action pair to the RTC database. The service manager can register as many pairs as necessary. It can also register multiple actions to perform for a single condition. Any event or set of events from a provider service can be registered.

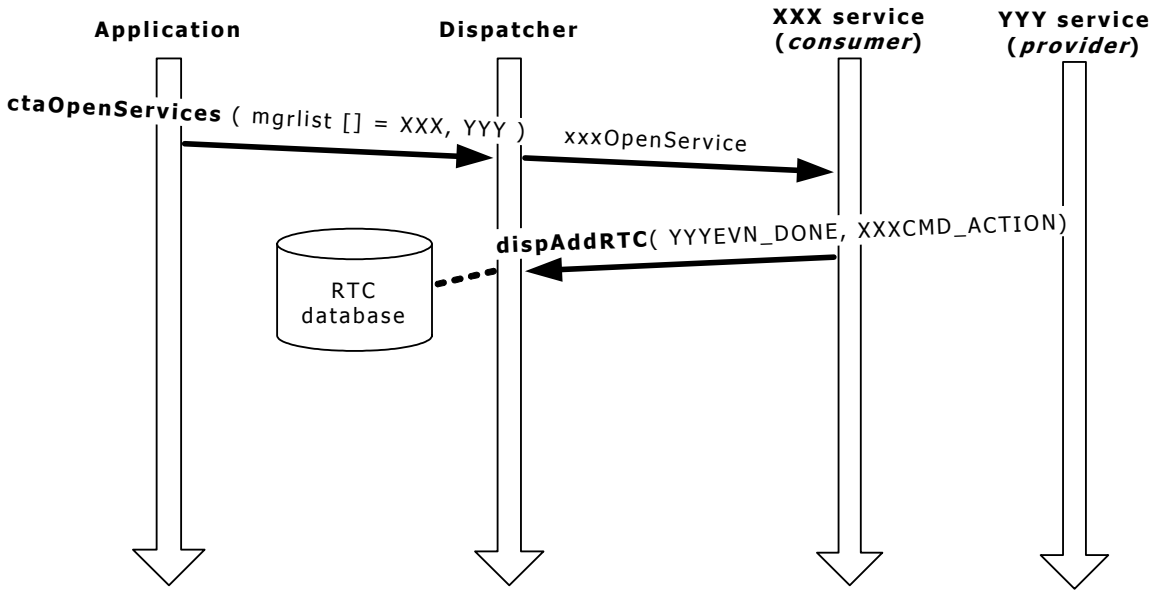
The dispatcher directs services to take actions by issuing Natural Access commands. If the service you are creating is to receive RTC notifications, the service must define one or more action identifier values of type CTA_CODE_COMMAND.

Note: Condition and action pairs are registered on a context-by-context basis. Conditions and actions specified within one context do not apply to other contexts. Also, all services must be located within the same Natural Access server.

Service managers can register for RTC notification at any time: during the initialization of the service or as a result of a function call made by an application. For example, the service manager can register for RTC notification in response to an indication from an application (through a call parameter) that voice play should be stopped on barge-in.

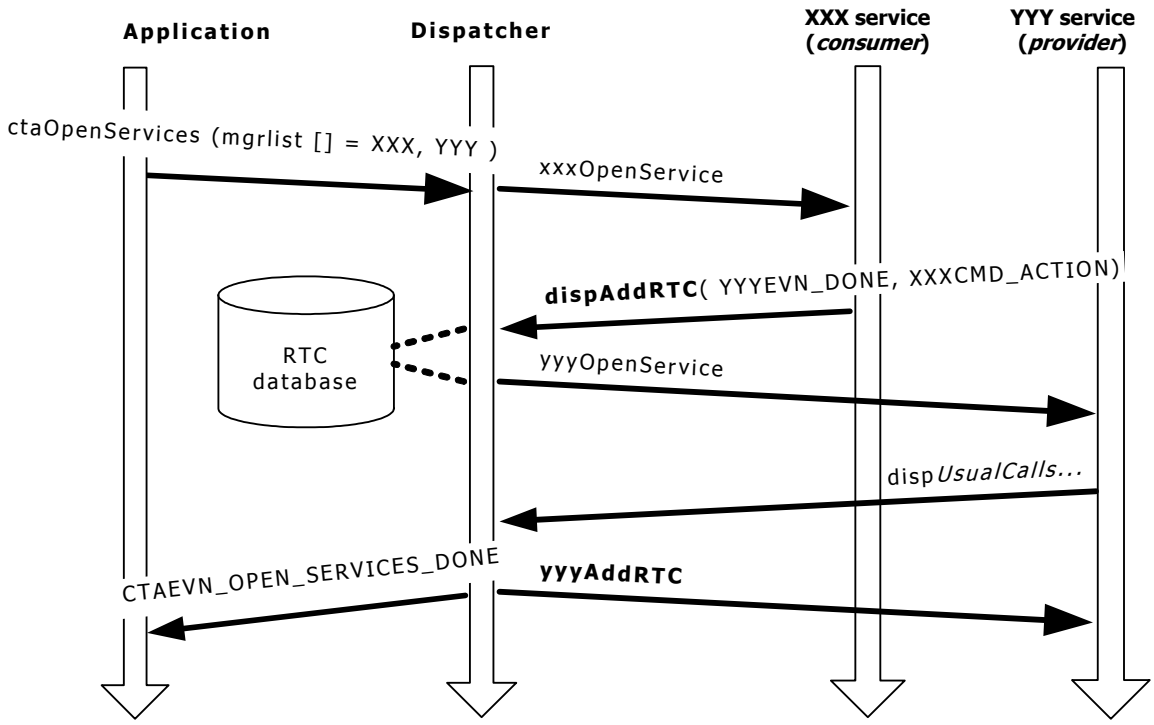
If RTC notifications are set up during service initialization, the consumer service typically makes its invocations of **dispAddRTC** when the dispatcher invokes **xxxOpenService** to open the consumer service. The provider service does not need to be open for the **dispAddRTC** invocation to succeed.

Note: If RTC notifications are set up during service initialization, these requests are typically also removed when **xxxCloseService** is invoked (using the dispatcher function **dispRemoveRTC**).



dispAddRTC

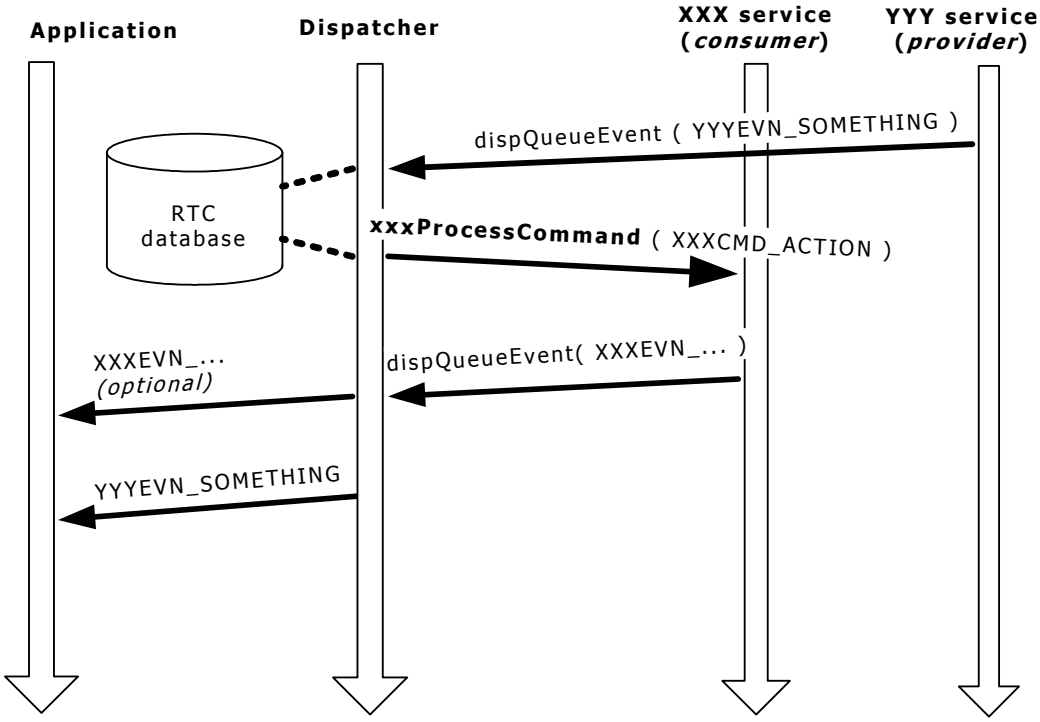
When the provider service specified in the **dispAddRTC** invocation is opened, the dispatcher invokes **xxxAddRTC**. **xxxAddRTC** is an optional binding function. The service writer determines the actions taken by the service when this function is called. Then CTAEVN_OPEN_SERVICES_DONE is returned on the client side:



xxxAddRTC

When a provider service places an event in the event queue that satisfies an RTC condition, the dispatcher consults the database to determine the consumer services to notify and the commands to send to which services. The dispatcher invokes a consumer service's **xxxProcessCommand** function to direct the service to perform an action. During processing of the action, the consumer service can optionally pass events to the dispatcher, which passes them to the client side. When all commands are complete, the original trigger event generated by the provider service is passed to the client side.

xxxProcessCommand is synchronous. It is the responsibility of the consumer service to return to the dispatcher as soon as possible so that performance is not adversely affected by its RTC action processing.



Action sent in response to event

The order in which services receive RTC action notifications from the dispatcher is independent of the order in which they invoke **dispAddRTC**. The order is also independent of the order in which services are listed in the **ctaOpenServices** call.

The action command sent to the service manager with **xxxProcessCommand** is in the form of a generic DISP_COMMAND structure. The trigger event (DISP_EVENT) is mapped into DISP_COMMAND in the following way:

1. The event ID and value are mapped into an RTC_EVT_CMD_DATA structure:

```
/* RTC "DISP_EVENT" to "DISP_COMMAND" data */
typedef struct
{
    DWORD Size;       /* Size RTC_EVT_CMD_DATA         */
    DWORD EventId;    /* "id" field from DISP_EVENT    */
    DWORD EventValue; /* "value" field from DISP_EVENT */
} RTC_EVT_CMD_DATA;
```

2. A pointer to the RTC_EVT_CMD_DATA structure is sent in the DISP_COMMAND structure along with other information, as follows:

DISP_COMMAND field	Contents
id	Action command from the RTC database (RTC_ACTION ActionCmd)
ctahd	ctahd field in DISP_EVENT
size1	sizeof (RTC_EVT_CMD_DATA)
dataptr1	Pointer to RTC_EVT_CMD_DATA
size2	size field in DISP_EVENT
dataptr2	buffer field in DISP_EVENT
size3	extra field in DISP_EVENT
dataptr3	NULL
objHd	objHd field in DISP_EVENT
addr.source	CTA_CODE_SVCID(id) field in DISP_EVENT
addr.destination	Service ID of RTC consumer, from the RTC database (RTC_CONDITION Consumer)

If addr.destination is RTC_BROADCAST, the event sent in DISP_EVENT is not propagated up to the application level. This process enables services to pass informational events to one another using RTC notification without involving the application.

Using aliases

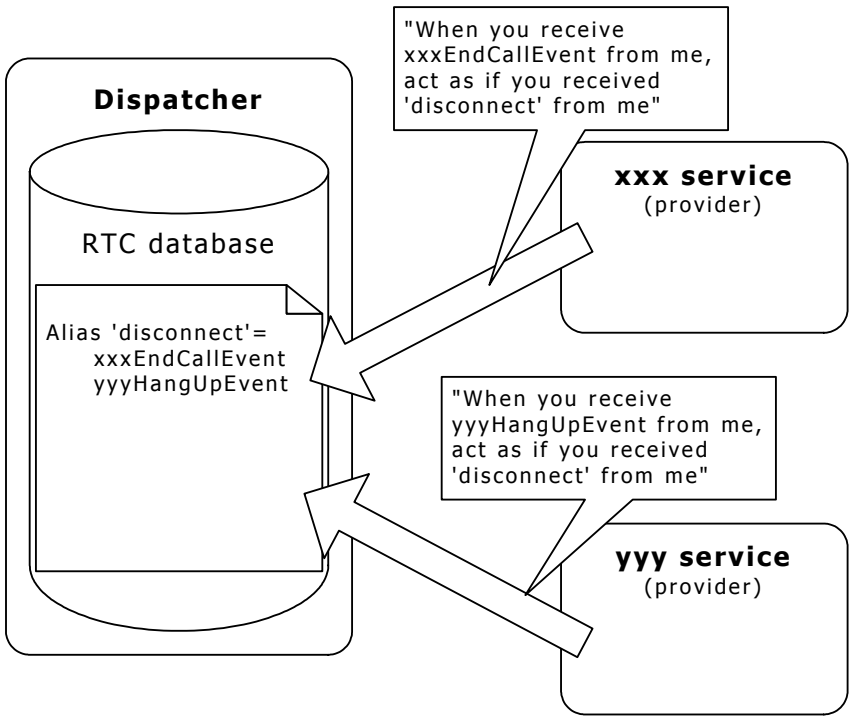
The Natural Access dispatcher enables provider services to associate aliases with their events in the RTC database. A consumer service can register actions to be taken whenever the dispatcher receives an event associated with a given alias.

This topic presents:

- Overview of using aliases
- Program flow

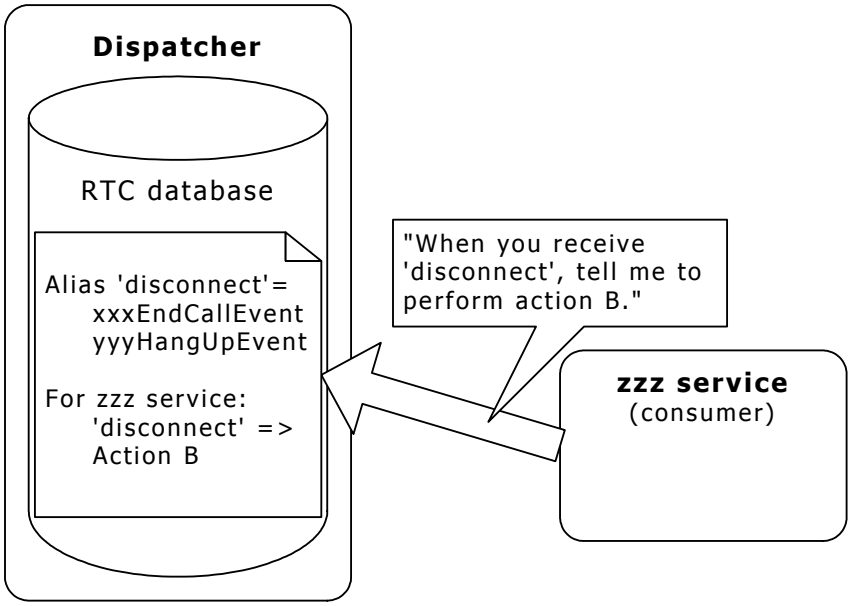
Overview of using aliases

The provider service creates the alias and event association in the database. Multiple services can associate different events with the same alias:



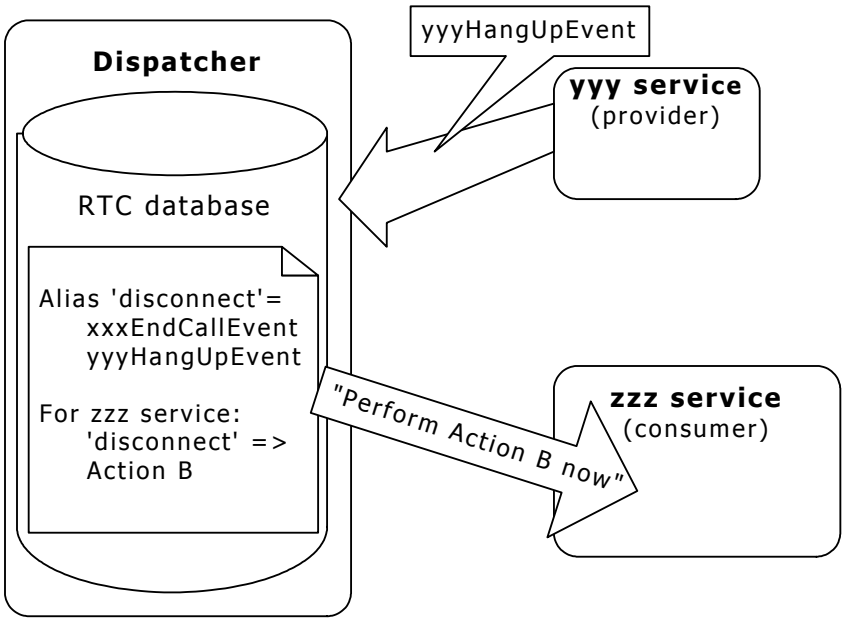
Adding aliases to RTC database

To receive RTC notification whenever an event in the alias group occurs, a consumer service uses **dispAddRTC** to register the alias as a condition and associate an action with it:



Registering an RTC action based on an alias

When an event associated with an alias occurs, the dispatcher notifies any consumer services registered with the alias:



Action performed in response to aliased event

Using of aliases has the following advantages:

- Instead of registering several times for different events, a consumer service can register one time to be notified whenever any event occurs that indicates a certain state or occurrence.
- Some services generate different events indicating the same occurrence. If these events are both listed under the same alias, a registered consumer service can be alerted to an event without needing to know which service generated the event, or without having to register both events.
- Event names can change, and new event names can be added to an alias without changes to the consumer services.

Note: Some Natural Access base services register aliases when they are opened on a context. For a list of these aliases and associated events, refer to *Natural Access base services RTC aliases* on page 152.

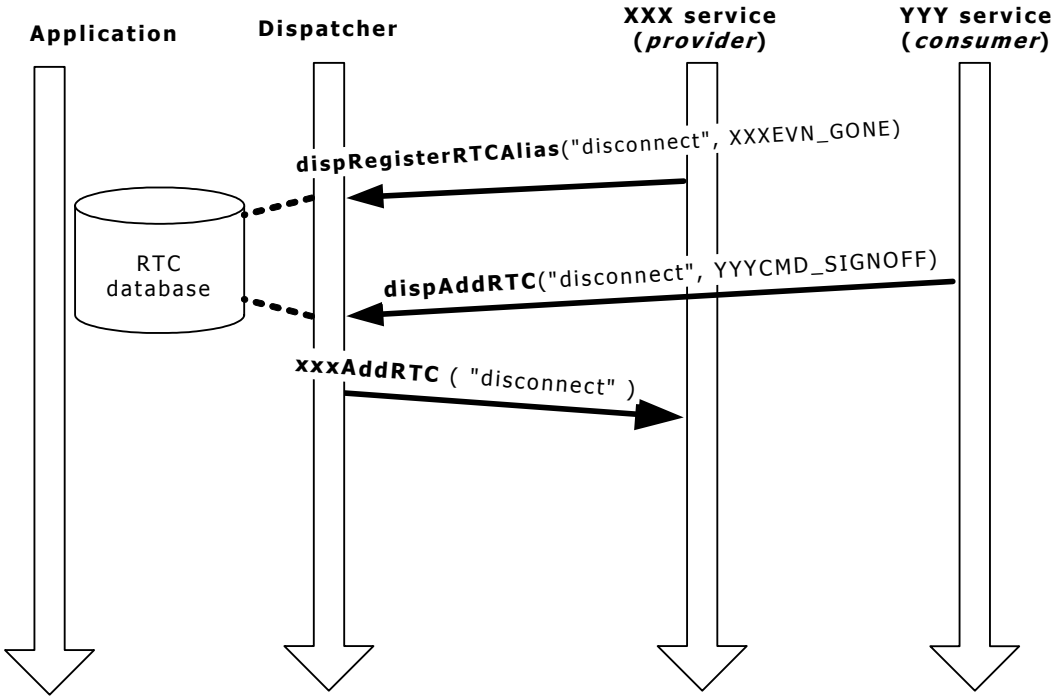
Program flow

The provider service uses **dispRegisterRTCAlias** to create the alias and event association in the database. Each successful invocation of **dispRegisterRTCAlias** adds one alias and event association to the RTC database. The service manager can add as many as necessary. It can also add multiple events to the same alias. Alias names do not need to be unique between service providers.

Provider service managers can add aliases using **dispRegisterRTCAlias** at any time. Typically, however, this task is performed when the service is opened (the dispatcher invokes **xxxOpenService**). If aliases are added at this time, the aliases are also typically removed when **xxxCloseService** is invoked (using the dispatcher function **dispUnregisterRTCAlias**).

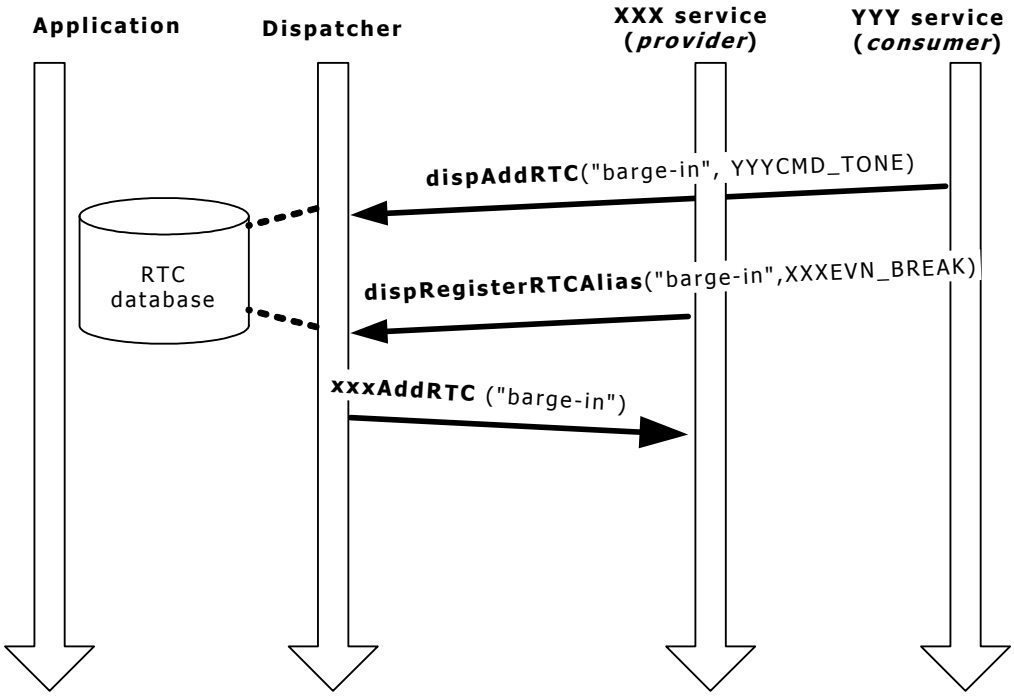
To register an action to take when a provider service generates an event associated with a given alias, a consumer service invokes **dispAddRTC**. With this function, the service manager passes pointers to structures specifying the alias as the condition, and a corresponding action command.

When a provider service has created an alias and event association in the database, and at least one consumer service has invoked **dispAddRTC** to associate an action with the alias, the dispatcher invokes the provider service's **xxxAddRTC** command. The alias is passed back with the command:



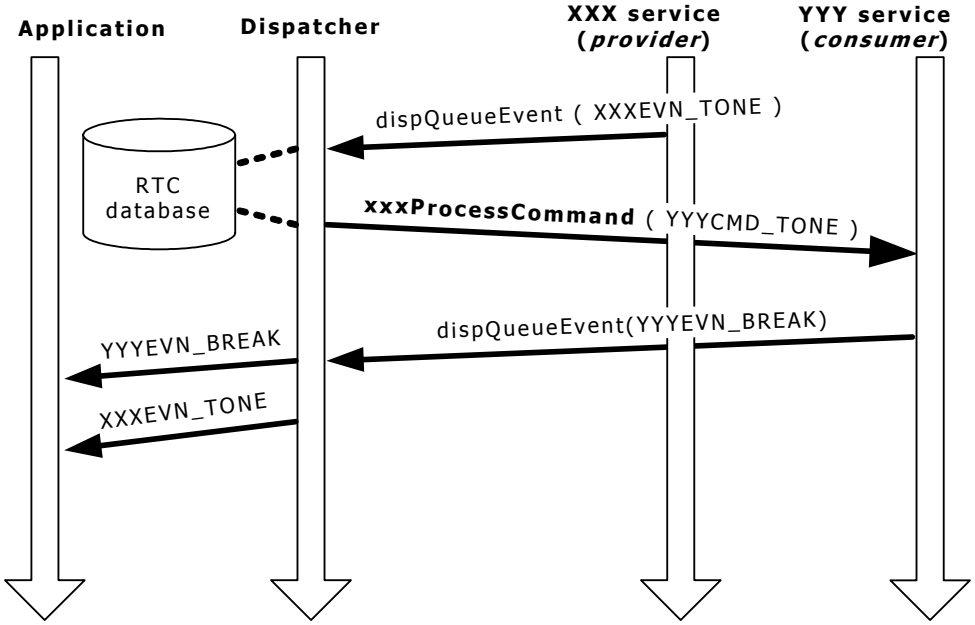
Command flow: registering aliases

Consumer services can associate actions with a given alias even before the provider service has created it using **dispRegisterRTCAlias**. In either case, the provider service receives **xxxAddRTC** only after it has created the alias and at least one consumer service has requested RTC notification based on the alias:



xxxAddRTC returned only after request for RTC notification

When the provider service generates an event, the dispatcher consults its database. If the event is associated with an alias, the dispatcher sends action commands to any consumer services that registered to be notified:



Action sent in response to event with alias

Registering for RTC notification

To register a condition and associated action, the service manager invokes **dispAddRTC** and passes pointers to the RTC_CONDITION and RTC_ACTION structures specifying the condition and action. The RTC_CONDITION and RTC_ACTION structures are defined in *ctadisp.h*:

```
typedef struct _RTC_CONDITION
{
  RTC_ATTRIBUTES Attribute;  /* Characteristics of RTC condition       */
  DWORD          EventID;    /* Service's event ID for condition       */
  DWORD          EventValue; /* Specific value sent with the event     */
                             /* - used with Attribute RTC_WITH_VALUE   */
  char           Alias[ RTC_NAME_LEN ]; /* ASCII name of RTC condition */
} RTC_CONDITION;

typedef struct _RTC_ACTION
{
  DWORD          Consumer;   /* Service ID of RTC consumer             */
  DWORD          ActionCmd;  /* Special command sent to service on     */
                             /* recognition of condition               */
} RTC_ACTION;
```

When specifying a condition using **dispAddRTC**, fill the fields in RTC_CONDITION as directed in the following table:

For the registering service to perform RTC_ACTION whenever...	Fill the fields in RTC_CONDITION as follows:
The dispatcher receives a specific event with any reason code	Attribute: RTC_NONE EventID: The event ID of the trigger event EventValue: Any value (will be ignored) Alias: Zero-length string
The dispatcher receives a specific event with a specific reason code	Attribute: RTC_EVT_WITH_ ONE_VALUE EventID: The event ID of the trigger event EventValue: The value to look for Alias: Zero-length string
The dispatcher receives any event from a specific service	Attribute: RTC_ALL_ PROVIDER_EVENTS EventID: The event ID of any event from the service. (The provider service is implied from the service identifier within the event ID.) EventValue: Any value (will be ignored) Alias: Zero-length string
The dispatcher receives an event associated with a specific alias	Attribute: RTC_NONE EventID: Any value (will be ignored) EventValue: Any value (will be ignored) Alias: The alias string (for example: disconnect) By default, alias strings can be up to 12 characters in length (this value is specified by the RTC_NAME_LEN constant in *ctadisp.h*). Aliases are case-insensitive: the RTC logic converts all alias name strings to lowercase before storing them in the database.

Canceling RTC notification

A consumer service can cancel RTC notification using the dispatcher function **dispRemoveRTC**. The service can cancel all RTC notifications, or unregister only for specific conditions.

A consumer service can invoke **dispRemoveRTC** at any time. Typically, however, if RTC notifications are set up when a service is initialized, notifications are removed when the service is closed (**xxxCloseService** is invoked).

With **dispRemoveRTC**, the service manager passes pointers to RTC_CONDITION and RTC_ACTION structures specifying the condition and action pair to remove from the database. When removing a condition using **dispRemoveRTC**, fill the fields in RTC_CONDITION and RTC_ACTION as directed in the following table:

For the registering service to STOP RTC notification for events that match these criteria:	Fill the fields in RTC_CONDITION as follows:	Fill the fields in RTC_ACTION as follows:
A specific provider A specific event ID A specific reason code Any action command	Attribute: RTC_EVT_WITH_ ONE_VALUE EventID: The event ID of the trigger event EventValue: The value to look for Alias: Zero-length string	Consumer: The ID of the consumer service ActionCmd: Any value (will be ignored)
A specific provider A specific event ID Any reason code Any action command	Attribute: RTC_EVT_WITH_ ANY_VALUE EventID: The event ID of the trigger event EventValue: Any value (will be ignored) Alias: Zero-length string	Consumer: The ID of the consumer service ActionCmd: Any value (will be ignored)
A specific provider A specific event ID Any reason code A specific action command	Attribute: RTC_NONE EventID: The event ID of the trigger event EventValue: Any value (will be ignored) Alias: Zero-length string	Consumer: The ID of the consumer service ActionCmd: The action command
A specific provider Any event ID Any reason code Any action command	Attribute: RTC_ALL_ PROVIDER_ EVENTS EventID: The event ID of any event from the service. (The provider service is implied from the service identifier within the event ID.) EventValue: Any value (will be ignored) Alias: Zero-length string	Consumer: The ID of the consumer service ActionCmd: Any value (will be ignored)
Any provider Any event ID Any reason code Any action command	Attribute: RTC_ALL_ CONSUMER_RTCS EventID: Any value (will be ignored) EventValue: Any value (will be ignored) Alias: Zero-length string	Consumer: The ID of the consumer service ActionCmd: Any value (will be ignored)
A specific alias	Attribute: RTC_NONE EventID: Any value (will be ignored) EventValue: Any value (will be ignored) Alias: The alias string (for example: disconnect)	Consumer: The ID of the consumer service ActionCmd: Any value (will be ignored)

Reading the RTC database

The dispatcher functions **dispGetActiveRTC** and **dispGetNumRTC** enable a service to query the database for all condition and action pairs stored within it for the specific context on which the service is opened. **dispGetActiveRTC** returns an array of the condition and action pairs associated with a context. **dispGetNumRTC** returns the number of condition and action pairs associated with a context.

The dispatcher functions **dispGetRTCAliases** and **dispGetNumRTCAliases** enable a service to query the database for all aliases stored within it for the specific context on which the service is opened. **dispGetRTCAliases** returns an array of the aliases stored for a context. Call **dispGetNumRTCAliases** to determine the amount of space that must be allocated for the **dispGetRTCAliases** return buffer.

Sample code

The following code sample demonstrates how to register and unregister an RTC condition and action pair:

Note: The enum definition and structures shown are in *ctadisp.h*.

```
/* RTC-related symbol definitions and data structures.      */
/* Specific characteristics about RTC being requested.      */
typedef enum {
  RTC_NONE = 0,            /* Nothing special               */
  RTC_ALL_CONSUMER_RTCS,   /* All consumer RTC actions      */
  RTC_ALL_PROVIDER_EVENTS, /* All provider RTC events       */
  RTC_EVT_AND_ONE_VALUE,   /* Use value as well as event ID */
  RTC_EVT_AND_ANY_VALUE    /* Use for all values w event ID */
}  RTC_ATTRIBUTES;

/* RTC "DISP_EVENT" to "DISP_COMMAND" data */
typedef struct
{
  DWORD    Size;       /* Size RTC_EVT_CMD_DATA            */
  DWORD    EventId;    /* Event ID of RTC condition        */
  DWORD    EventValue; /* "value" from RTC condition event */
}  RTC_EVT_CMD_DATA;

/* RTC "event ID" to "name" alias. Length include terminating NULL */
#define RTC_NAME_LEN 20
typedef struct
{
  DWORD   EventID;               /* RTC condition event ID           */
  Char    Alias[ RTC_NAME_LEN ]; / *ASCII name associated with event */
}  RTC_ALIAS;

typedef struct _RTC_CONDITION
{
  RTC_ATTRIBUTES  Attribute;      /* Characteristics of condition   */
  DWORD    EventID;               /* Service event ID for condition */
  DWORD    EventValue;            /* (RTC_WITH_VALUE)               */
  Char    Alias[ RTC_NAME_LEN ];  /* ASCII name of RTC condition    */
}  RTC_CONDITION;

typedef struct _RTC_ACTION
{
  DWORD    Consumer;    /* Service ID of RTC consumer           */
  DWORD    ActionCmd;   /* Special command sent by service on   */
                        /* recognition of condition             */
}  RTC_ACTION;

/* An example of  registering and canceling a pair of RTC_CONDITION */
/* and RTC_ACTION. */

RTC_CONDITION    condition;
RTC_ACTION       action;

/* RTC_CONDITION: one event code, one reason code, from one */
/*   provider service  (TIK)                                */
condition.Attribute = RTC_EVT_AND_ONE_VALUE;  /* Event + reason as */
                                              /*   condition       */
condition.EventID   = TIKEVN_TIMER_DONE;      /* Event code        */
condition.Reason    = CTA_REASON_FINISHED;    /* Reason code       */

/* RTC_ACTION: EVT service as RTC consumer, EVTCMD_SHOW_EVENT */
/*   as callback function                                     */
action.Consumer     = EVT_SVCID;         /* RTC consumer service ID */
action.ActionCmd    = EVTCMD_SHOW_EVENT; /* Call back function when */
                                         /*   condition met         */

/* Register the pair of RTC_CONDITION and RTC_ACTION. */
dispAddRTC(m->ctahd,&condition,&action);  /* Condition-action pair */
                                          /*   added to RTC        */
```

```
/* Cancel the pair of RTC_CONDITION and RTC_ACTION. */
condition.Attribute = RTC_EVT_AND_ONE_VALUE;  /* Event + reason as */
                                              /*   condition       */
condition.EventID   = TIKEVN_TIMER_DONE;      /* Event code        */
condition.Reason    = CTA_REASON_FINISHED;    /* Reason code       */

action.Consumer     = EVT_SVCID;         /* RTC consumer service ID  */
action.ActionCmd    = EVTCMD_SHOW_EVENT; /* Call back function when  */
                                         /*   condition met          */

dispRemoveRTC(m->ctahd,&condition,&action); /* Condition-action pair */
                                            /*   removed from RTC    */
```

The following example shows the callback function provided by the event tracking service (EVT) as part of the RTC action command:

```
DWORD evtCmdShowEvent( EVT_PERCTX* pevtPerCtx, DISP_COMMAND* m )
{
   DWORD ret = SUCCESS;
   RTC_EVT_CMD_DATA rtcData;

   /* Validate the arguments. */
   ret = evtValidate( pevtPerCtx, m );
   if ( ret != SUCCESS )
      return ret;

   /* Update the event counts for queuecontext and mgrcontext */
   /*   whenever an event qualifies an RTC_CONDITION.         */
   pevtPerCtx->pevtPerQue->evcount++;
   pevtPerCtx->evcount++;

   /* Check if the application log switch is on. */
   if (!pevtPerCtx->logStarted)
   {
      return SUCCESS;
   }
   /* Retrieve the Event ID and Event Value fields. */
   memcpy(&rtcData,m->dataptr1,m->size1);

   if (evtLogFileChoice == 1)
   {
      /* Log event into the user-chosen event log file. */
      ... ... ... ...
   }
   else
   {
      /* Log event into the standard output file. */
      ... ... ... ...
   }
   return ret;
}
```

Natural Access base services RTC aliases

The following table identifies RTC aliases that are registered by Natural Access base services.

RTC alias registered	Base service	Service events
Connected	NCC	NCCEVN_CALL_CONNECTED
Disconnected	NCC	NCCEVN_CALL_DISCONNECTED
Released	NCC	NCCEVN_CALL_RELEASED
Media_Ready	NCC	NCCEVN_READY_FOR_MEDIA
Media_Not_Ready	NCC	NCCEVN_NOT_READY_FOR_MEDIA

13 Service manager functions

Service manager function summary

This topic presents a summary of the service manager functions:

- Life cycle binding functions
- Runtime binding functions

Life cycle binding functions

Registration

The following binding functions are used to register Natural Access services and service managers. Both of these functions are synchronous:

Function	Description
xxxInitializeManager	Acts as an entry point into service. Declares binding functions and version numbers and compatibility levels for service manager.
xxxDefineService	Declares version numbers, compatibility levels, and parameters for service.

Event handling

The following binding functions are used to initialize and shut down event handling. Both of these functions are synchronous:

Function	Description
xxxAttachServiceManager	Initializes communication with managed resource. Can declare wait objects and allocate queuecontext data structure.
xxxDetachServiceManager	De-allocates data structures allocated with **xxxAttachServiceManager**.

Service startup/shutdown

The following binding functions are used during service startup and shutdown:

Function	Synchronous/ asynchronous	Description
xxxOpenServiceManager	Synchronous	Allocates managed resource data structure that pertains to the specific context.
xxxOpenService	Asynchronous	Initializes the managed resource.
xxxCloseServiceManager	Synchronous	De-allocates any data structures opened with **xxxOpenServiceManager**.
xxxCloseService	Asynchronous	Shuts down the managed resource.

Runtime control

The following binding functions are invoked when a service associates a runtime control action with a service specific event. Both of these functions are synchronous:

Function	Description
xxxAddRTC	Service-specific actions taken when **dispAddRTC** function is called.
xxxRemoveRTC	Service-specific actions taken when **dispRemoveRTC** function is called.

Runtime binding functions

Command processing

The **xxxProcessCommand** binding function enables you to process service commands. This function is synchronous.

Trace handling

The following binding functions enable you to control tracing for specific services. Both of these functions are synchronous:

Function	Description
xxxFormatTraceBuffer	Formats a service-specific trace buffer.
xxxSetTraceLevel	Sets the tracemask for a specific service on a per context basis.

Error handling

The following binding functions are used to convert binary codes and trace records to ASCII. Both of these functions are synchronous:

Function	Description
xxxFormatMessage	Formats a command or event that was generated by the service.
xxxGetText	Retrieves the service-specific textual representation of an error, reason, event, or command code.

Event processing

The **xxxProcessEvent** binding function is used to process events from other services. This function is synchronous.

Using the service manager function reference

This section provides an alphabetically ordered reference to the service manager functions. A prototype of each function is shown with the function description and details of all arguments and return values. A typical function description includes:

Prototype	The prototype is shown followed by a listing of the function's arguments. Data types include: • WORD16-bit unsigned • DWORD32-bit unsigned • INT1616-bit signed • INT3232-bit signed • BYTE8-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.
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.

xxxAddRTC

Invoked by the dispatcher into a provider service when an RTC condition and action is set up involving events from the service.

Prototype

DWORD **xxxAddRTC** (CTAHD ***ctahd***, void ****mgrcontext***, RTC_CONDITION ****pcondition***, RTC_ACTION ****paction***)

Argument	Description
ctahd	Handle to a Natural Access context.
mgrcontext	Context pointer allocated when service manager opened.

pcondition	Pointer to an RTC_CONDITION structure describing the RTC condition:
	`typedef struct _RTC_CONDITION`
	`{`
	`  RTC_ATTRIBUTES Attribute; /*Characteristics of RTC condition */`
	`  DWORD  EventID;  /* Service's event ID for condition */`
	`  DWORD  EventValue;  /* Specific value sent with the event.*/`
	`  /* Used with Attribute RTC_WITH_VALUE */`
	`  char  Alias[ RTC_NAME_LEN ]; /* ASCII name of RTC condition */`
	`} RTC_CONDITION;`

paction	Pointer to an RTC_ACTION structure describing the RTC action:
	`typedef struct _RTC_ACTION`
	`{`
	`  DWORD  Consumer;  /* Service ID of RTC consumer */`
	`  DWORD  ActionCmd;  /* Special command sent to service on */`
	`  /* recognition of condition */`
	`} RTC_ACTION;`

Return values

Return value	Description
SUCCESS	Dispatcher to send RTC action to RTC consumer.
Any service-specific error	

Details

xxxAddRTC is invoked by the dispatcher into a provider service when an RTC condition and action is set up involving events from the service.

The function is invoked in the following cases:

- A consumer service successfully registered a condition and action pair using **dispAddRTC**, where the condition involves one of the provider service's events.
- The provider service added an alias for one or more of its events to the database using **dispRegisterRTCAlias**, and a consumer service has successfully registered a condition and action pair using **dispAddRTC**, where the condition involves the alias.

In either case, the dispatcher waits until the provider service is open on the context before invoking **xxxAddRTC**.

Any error returned from this binding function is returned back to the calling service.

See also

dispRemoveRTC, xxxRemoveRTC

xxxAttachServiceManager

Called when a service manager is attached to a Natural Access queue.

Prototype

DWORD **xxxAttachServiceManager** (CTAQUEUEHD ***ctaqueuehd***, unsigned ***mode***, void ****queuecontext***)

Argument	Description
ctaqueuehd	Handle created by dispatcher on call to **ctaCreateQueue**.
mode	Reserved for future use.
queuecontext	Pointer to a queue context returned by the service manager. A ***queuecontext*** is a device level data structure.

Return values

SUCCESS or any Natural Access or service-specific error code.

Details

The dispatcher invokes **xxxAttachServiceManager** when a Natural Access queue is created and the service manager is listed in the arguments to **ctaCreateQueue**. Only service managers in this list are attached and are available to be opened on the queue.

queuecontext provides a way for the service manager to allocate a device object that it will associate with the queue. If the ***queuecontext*** is not needed, ***queuecontext*** can have NULL assigned as its value.

Wait objects should be registered with the dispatcher if the managed resource is multiplexed with this function.

For more information about this function, refer to *Initializing managed resource event handling* on page 85.

See also

dispRegisterWaitObject, **xxxDetachServiceManager**

xxxCloseService

Called when a service is closed.

Prototype

DWORD **xxxCloseService** (CTAHD ***ctahd***, void ****mgrcontext***, char ****svcname***, unsigned ***svcid***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to context returned by **xxxOpenServiceManager**.
svcname	Pointer to the name of the service to close.
svcid	Identifier of the service to close.

Return values

SUCCESS or any Natural Access or service-specific error code.

Events

CTAEVN_CLOSE_SERVICE_DONE with the reason CTA_REASON_FINISHED, or an error code.

Details

xxxCloseService is invoked when a service is closed on a context using **ctaCloseServices**, **ctaDestroyContext**, or **ctaDestroyQueue**. Both the service name (***svcname***) and service ID (***svcid***) are supplied as well as the manager context (***mgrcontext***) provided by the service manager in the call to **xxxOpenServiceManager**.

Services that do not require asynchronous close service processing must queue CTAEVN_DISP_CLOSE_SERVICE_DONE using **dispMakeAndQueueEvent** so that **ctaCloseServices** does not hang.

Resources allocated in **xxxOpenService** should be deallocated with this function call. Also, if RTC notifications were set up using **dispAddRTC** when the service was initialized, **xxxCloseService** should call **dispRemoveRTC** to remove its RTC notifications from the RTC database. If aliases were added at initialization time, **xxxCloseService** should also call **dispUnregisterRTCAlias** to remove the aliases from the RTC database.

The value field of the **xxxCloseService** completion event is set to CTA_REASON_FINISHED if the service was closed successfully or to any Natural Access or service-specific error message if there are problems closing the service.

In case of a synchronous error or an asynchronous error, Natural Access reports it in the Natural Access trace log and continues to close services. After the last service is closed, **ctaCloseServices** returns the first error to occur in the value field of CTAEVN_CLOSE_SERVICES_DONE. If all services are successfully closed, the value field contains CTA_REASON_FINISHED.

For more information about this function, refer to *Shutting down services* on page 95.

See also

xxxCloseServiceManager

xxxCloseServiceManager

Called when a service manager is closed on a context.

Prototype

DWORD **xxxCloseServiceManager** (CTAHD ***ctahd***, void ****mgrcontext***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to data structure containing device information for a single client returned by **xxxOpenServiceManager**.

Return values

SUCCESS or any Natural Access or service-specific error code.

Details

xxxCloseServiceManager is invoked immediately after the last service provided by this service manager is closed on a context using **ctaCloseServices**.

This function should de-allocate any resources allocated in **xxxOpenServiceManager** and unregister any wait objects.

For more information about this function, refer to *Shutting down services* on page 95.

See also

dispUnregisterWaitObject, **xxxAttachServiceManager**, **xxxCloseService**

xxxDefineService

Defines a service at initialization time.

Prototype

DWORD **xxxDefineService** (char ***svcname**)

Argument	Description
svcname	Pointer to the name of a service to be registered with Natural Access.

Return values

SUCCESS or error returned by **dispRegisterService**

Details

xxxDefineService is called at initialization time for each service to be registered with the dispatcher. **xxxDefineService** calls **dispRegisterService** to perform the following functions:

- Declare the parameter descriptor tables.
- Declare the version numbers for the service.
- Request SPI compatibility level checking for other services whose SPI is called from this service.
- Declare compatibility levels for service.

For more information about this function, refer to *Registering services* on page 76.

xxxDetachServiceManager

Detaches a service manager from a Natural Access queue.

Prototype

DWORD **xxxDetachServiceManager** (CTAQUEUEHD ***ctaqueuehd***, void ****queuecontext***)

Argument	Description
ctaqueuehd	Handle created by the dispatcher in the call to **ctaCreateQueue**.
queuecontext	Pointer to the data structure allocated in **xxxAttachServiceManager**.

Return values

SUCCESS or any Natural Access or service-specific error code.

Details

xxxDetachServiceManager is invoked when the Natural Access queue is destroyed using **ctaDestroyQueue**.

This function should de-allocate any resources allocated in **xxxAttachServiceManager** and unregister any wait objects.

For more information about this function, refer to *Shutting down event handling* on page 96.

See also

dispUnregisterWaitObject

xxxFormatMessage

Formats a command or event generated by a service managed by this service manager.

Prototype

DWORD **xxxFormatMessage** (DISP_MESSAGE ***pmsg***, char ***msgtext***, unsigned ***size***, char ***indent***)

Argument	Description

pmsg	Pointer to the DISP_MESSAGE structure:
	`typedef struct`
	`{`
	`  DWORD id;  /* Event code (and source service ID ) */`
	`  DWORD ctahd;  /* Natural Accesso context handle  */`
	`  DWORD specific[6];  /* Fields specific to events/command  */`
	`  DWORD reserved;  /* Must be zero (0).  */`
	`  MESSAGE_ADDR addr;  /* Source/destination service ID  */`
	`} DISP_MESSAGE;`
	
	`typedef struct`
	`{`
	`  WORD source  /* source service ID  */`
	`  WORD destination  /* destination service ID  */`
	`} MESSAGE_ADDR;`

msgtext	Pointer to a character string to receive the formatted message.
size	Size of ***msgtext***.
indent	Pointer to a character string to be placed at the beginning of each newly formatted line.

Return values

SUCCESS

Details

xxxFormatMessage formats a dispatcher command or event into a user-readable text format. The DISP_MESSAGE structure contains the fields necessary to determine whether the message is a command or an event. Once this is known, the function can cast the pointer to the appropriate DISP_COMMAND or DISP_EVENT structure and format it appropriately.

msgtext is passed in by the dispatcher and its ***size*** should be checked so that the service only copies as much text as can fit into the string without overwriting the end and causing a protection violation. Typically, services format their messages into an internal buffer big enough to hold the text of any message produced by the service. It then copies as much as it can to ***msgtext***.

indent should be prepended to the beginning of each new text line by the formatting routine. This gives the application some control over how events are printed if this function is called from **ctaFormatEvent**.

For more information about this function, refer to *Runtime binding functions* on page 154.

See also

xxxGetText

xxxFormatTraceBuffer

Formats a service-specific trace buffer.

Prototype

DWORD **xxxFormatTraceBuffer** (unsigned ***tracetag***, void ***inbuffer***, unsigned ***insize***, char ***outbuffer***, unsigned ***outsize***)

Argument	Description
tracetag	Service specific tracetag.
inbuffer	Pointer to a byte array containing the trace record to format.
insize	Size of ***inbuffer***.
outbuffer	Pointer to the character array for the formatted output.
outsize	Size of ***outbuffer***.

Return values

Return value	Description
SUCCESS	
CTAERR_NOT_IMPLEMENTED	The function was not implemented.

Details

xxxFormatTraceBuffer is called only by *ctdaemon* when a trace buffer is written to the Natural Access trace log and is referred to by the service specific ***tracetag***. Trace tags that are defined by Natural Access can be formatted directly by dispatcher formatting routines. Trace tags created by a service contain the service ID in the high word and a service specific ID in the low word.

inbuffer is the exact array of bytes that was passed to **dispLogTrace** by the service that ultimately formats the buffer. This mechanism allows services to define their own trace structures and have them printed by *ctdaemon*.

outbuffer should be checked to make sure that it can hold the entire formatted trace output before anything is written to it. Refer to **xxxFormatMessage** for more details.

For more information about this function, refer to *Runtime binding functions* on page 154.

xxxGetText

Retrieves the service-specific textual representation of an error, reason, event, or command code.

Prototype

const char * **xxxGetText** (unsigned ***code***)

Argument	Description
code	Error, reason, event, or command code specified by a service managed by this service manager.

Return values

Pointer to returned character string.

Details

xxxGetText returns a pointer to the text string corresponding to the service specific error, reason, event, or command in ***code***. If the service does not know how to translate the code, NULL is returned to the dispatcher.

For more information about this function, refer to *Runtime binding functions* on page 154.

See also

xxxFormatMessage

xxxInitializeManager

Initializes a service manager by registering it with Natural Access.

Prototype

DWORD **xxxInitializeManager** ( )

Return values

SUCCESS or error returned by **dispRegisterServiceManager**

Details

xxxInitializeManager is the only required exported function from a service manager DLL or shared library. It is called when the process is initialized using **ctaInitialize**. It is the only entry that must be visible to Natural Access in the service manager library.

The three letter prefix of this function must reflect the specific three letter prefix assigned to each service manager by NMS. For example, the ADI manager is found in *adimgr.dll* and this function is called **adiInitializeManager**.

xxxInitializeManager does the following:

- Acts as an entry point into the service for service registration
- Declares version numbers for the service manager
- Declares compatibility levels for the service manager
- Declares service manager binding functions
- Acquires the global tracemask pointer managed by *ctdaemon* (if active)

For more information about this function, refer to *Registering services* on page 76.

xxxOpenService

Called when a service is opened.

Prototype

DWORD **xxxOpenService** (CTAHD ***ctahd***, void ****mgrcontext***, char ****svcname***, unsigned ***svcid***, CTA_MVIP_ADDR ****mvipaddr***, CTA_SERVICE_ARGS ****svcargs***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to the data structure containing information about the managed resource for a specific context returned by **xxxOpenServiceManager**.
svcname	Pointer to the name of the service to open.
svcid	Identifier of the service to open.

mvipaddr	Pointer to the MVIP address structure:
	`typedef struct`
	`{`
	`  DWORD board;`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`  DWORD mode;`
	`} CTA_MVIP_ADDR;`

svcargs	Pointer to service-specific arguments:
	`typedef struct`
	`{`
	`  char strarg [40];`
	`  DWORD args [10];`
	`} CTA_SERVICE_ARGS;`

Return values

SUCCESS or any service-specific error code.

Events

CTAEVN_DISP_OPEN_SERVICE_DONE

The event value field contains the open services completion status. The value should be set to CTA_REASON_FINISHED if the service was successfully opened.

Any other value means the service failed to open.

Details

xxxOpenService is invoked when a service is opened on a context using **ctaOpenServices**. ***mgrcontext*** is used to reference the context- specific memory created when the service manager was opened with **xxxOpenServiceManager**.

The MVIP_ADDR structure conveys the address of a resource to the device service. This is typically some form of DSP resource associated with the service being opened. For AG boards, it is the board number and local or MVIP stream and timeslot of the DSP port associated with the ADI service.

Other resources that may be required by the service can be specified in the service specific arguments. It is the responsibility of the service to define the meaning of the fields of the CTA_SERVICE_ARGS structure.

If the service makes use of RTC notifications, **xxxOpenService** can set up the notifications using **dispAddRTC** and **dispRegisterRTCAlias**.

xxxOpenService is asynchronous (it should return immediately), but **ctaOpenServices** waits for the completion event to be returned by the service before continuing to open services or return a completion event to the application. The value field of the completion event is set to CTA_REASON_FINISHED if the function completed successfully. If it is set to any other reason or error code, **ctaOpenServices** closes all services it was in the process of opening and returns the error to the application in the completion event.

Services that do not require asynchronous open service processing must queue CTAEVN_DISP_OPEN_SERVICE_DONE using **dispQueueEvent**. Otherwise, **ctaOpenServices** hangs.

Note: If this function returns an error, there is no asynchronous event expected and **xxxCloseService** is not called. If an error is returned asynchronously in the completion event, **xxxCloseService** is called. The service must clean up any resources it allocated in the process of opening the service.

For more information about this function, refer to *Starting services* on page 89.

xxxOpenServiceManager

Called when a service manager is opened on a context.

Prototype

DWORD **xxxOpenServiceManager** (CTAHD ***ctahd***, void ****queuecontext***, void *****mgrcontext***)

Argument	Description
ctahd	Context handle.
queuecontext	Pointer to the data structure returned by **xxxAttachServiceManager**.
mgrcontext	Pointer to the returned manager context.

Return values

SUCCESS

Details

When **xxxOpenServiceManager** is called, any resources that the service manager needs to manage are allocated and wait objects are registered with the dispatcher.

For more information about this function, refer to *Starting services* on page 89.

See also

dispRegisterWaitObject, xxxOpenService

xxxProcessCommand

Called when a command is addressed to a service.

Prototype

DWORD **xxxProcessCommand** (CTAHD ***ctahd***, void ****mgrcontext***, DISP_COMMAND ****ctacmd***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to data structure returned by **xxxOpenServiceManager**.

ctacmd	Pointer to a command structure to process. The DISP_COMMAND structure contains the following fields:
	`typedef struct`
	`{`
	`  DWORD id;  /* Command ID (and dest. service ID)  */`
	`  CTAHD ctahd;  /* Natural Accesso context handle  */`
	`  DWORD size1;  /* If dataptr1 !=NULL => size1 is buffer */`
	`  /* size - else message specific data  */`
	`  void *dataptr1;  /* Data pointer  */`
	`  DWORD size2;  /* If dataptr2 !=NULL => size2 is buffer */`
	`  /* size - else message specific data  */`
	`  void *dataptr2;  /* Data pointer  */`
	`  DWORD size3;  /* If dataptr3 !=NULL => size3 is buffer */`
	`  /* size - else message specific data  */`
	`  void *dataptr3;  /* Data pointer  */`
	`  DWORD reserved;  /* Must be zero (0).  */`
	`  MESSAGE_ADDR addr;  /* Source/Destination service ID  */`
	`} DISP_COMMAND;`
	
	`typedef struct`
	`{`
	`  WORD source  /* source service ID  */`
	`  WORD destination  /* destination service ID  */`
	`} MESSAGE_ADDR;`

Return values

Return value	Description
SUCCESS	There are no out or inout parameters being returned.
SUCCESS_RESPONSE	There are out or inout parameters containing data to be returned to the caller.
Any Natural Access or service-specific error code.	

Details

xxxProcessCommand is invoked when a command is sent to a service managed by this service manager. The ***mgrcontext*** allocated when the service manager was opened (with **xxxOpenServiceManager**) and a pointer to the command to process are passed in as arguments.

This function:

- Allocates a table of function pointers to API implementation functions
- Performs a lookup based on an incoming command code within the DISP_COMMAND message buffer
- Performs an upcall to the appropriate function

Commands must not block in **xxxProcessCommand**. They must return SUCCESS, SUCCESS_RESPONSE, or an error code to the dispatcher.

The dispatcher returns either SUCCESS or an error code to the calling SPI function that called **dispSendCommand**.

If **xxxProcessCommand** returns SUCCESS, either the function completed successfully without return information or an asynchronous function was started successfully.

If **xxxProcessCommand** returns SUCCESS_RESPONSE, either the function completed successfully with return information or an asynchronous function started successfully.

For more information about this function, refer to *Runtime binding functions* on page 154.

xxxProcessEvent

Called when an event is addressed to a service.

Prototype

DWORD **xxxProcessEvent** (CTAHD ***ctahd***, void ****mgrcontext***, DISP_EVENT ****ctaevt***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to data structure returned by **xxxOpenServiceManager**.

ctaevt	Pointer to event structure:
	`typedef struct`
	`{`
	`  DWORD id;  /* Event code (and source service ID)  */`
	`  CTAHD ctahd;  /* Natural Accesso context handle  */`
	`  DWORD timestamp; /* timestamp  */`
	`  DWORD userid;  /* User ID (defined by ctaCreateContext  */`
	`  DWORD size;  /* Size of buffer if buffer != NULL  */`
	`  void *buffer;  /* Buffer pointer  */`
	`  DWORD value;  /* Event status or event-specific data  */`
	`  DWORD extra;  /* Extra internal event field  */`
	`  unsigned objHd; /*Service object handle (formerly "reserved")*/`
	`  MESSAGE_ADDR addr; /* Source/Destination service ID  */`
	`} DISP_EVENT;`
	
	`typedef struct`
	`{`
	`  WORD source  /* source service ID  */`
	`  WORD destination /* destination service ID  */`
	`} MESSAGE_ADDR;`

Return values

SUCCESS

Details

xxxProcessEvent is invoked when an event is sent to this service from another Natural Access service. The manager context allocated when the service manager was opened and a pointer to the event to process are passed in as arguments.

This function is typically called when a device service sends an event to an integration service. Usually a device service processes its events in the event handling function (**xxxFetchAndProcess**) registered with the wait object by **dispRegisterWaitObject**. If it has an event to send to the higher-level service, it sends it to the dispatcher event queue using **dispQueueEvent**. The dispatcher then calls **xxxProcessEvent** of the integration service for further processing.

For more information about this function, refer to Runtime binding functions.

See also

dispMakeAndQueueEvent, **dispRegisterServiceManager**

xxxRemoveRTC

Invoked by the dispatcher into a provider service when an RTC condition and action involving events from the service is removed from the RTC database.

Prototype

DWORD **xxxRemoveRTC** (CTAHD ***ctahd***, void ****mgrcontext***, RTC_CONDITION ****pcondition***, RTC_ACTION ****paction***)

Argument	Description
ctahd	Handle to a Natural Access context.
mgrcontext	Context pointer allocated when the service manager opened.

pcondition	Pointer to an RTC_CONDITION structure describing the RTC condition being removed from the database:
	`typedef struct _RTC_CONDITION`
	`{`
	`  RTC_ATTRIBUTES Attribute; /* Characteristics of RTC condition */`
	`  DWORD  EventID;  /* Service's event ID for condition */`
	`  DWORD  EventValue;  /* Specific value sent with the event.*/`
	`  /* Used with Attribute RTC_WITH_VALUE */`
	`  char  Alias[ RTC_NAME_LEN ]; /* ASCII name of RTC condition */`
	`} RTC_CONDITION;`

paction	Pointer to an RTC_ACTION structure describing the RTC action being removed from the database:
	`typedef struct _RTC_ACTION`
	`{`
	`  DWORD  Consumer;  /* Service ID of RTC consumer */`
	`  DWORD  ActionCmd;  /* Special command sent to service on */`
	`  /* recognition of condition */`
	`} RTC_ACTION;`

Return values

SUCCESS

Details

The dispatcher invokes **xxxRemoveRTC** into a provider service when an RTC condition and action involving events from the service is removed from the RTC database. This function is invoked in the following cases:

- A consumer service successfully removed a condition and action pair using **dispRemoveRTC**, where the condition involves one of the provider service's events.
- The provider service added an alias for one or more of its events to the database using **dispRegisterRTCAlias**, and a consumer service successfully removed a condition and action pair using **dispRemoveRTC**, where the condition involves the alias.

xxxRemoveRTC is an optional binding function. The service writer defines the actions taken by the service when this function is called.

See also

dispAddRTC, **xxxAddRTC**

xxxSetTraceLevel

Called to set the trace mask for a specific client (context).

Prototype

DWORD **xxxSetTraceLevel** (CTAHD ***ctahd***, void ****mgrcontext***, unsigned ***svcid***, unsigned ***tracemask***)

Argument	Description
ctahd	Context handle.
mgrcontext	Pointer to data structure returned by **xxxOpenServiceManager**.
svcid	ID of the service.
tracemask	Trace mask to be used by the service.

Return values

SUCCESS

Details

The dispatcher invokes **xxxSetTraceLevel** when the application calls **ctaSetTraceLevel** on a given context. The service responds to this call by storing the tracemask in its internal context structure (***mgrcontext***). When deciding whether to log trace messages to the Natural Access trace log, the service looks at both the global trace pointer and this context-specific trace mask. The trace mask contains bits for use by the dispatcher as well as service-specific bits for exclusive use of services.

For more information about this function, refer to *Runtime binding functions* on page 154.

See also

xxxFormatMessage, **xxxFormatTraceBuffer**, **xxxGetText**

14 Dispatcher service functions

Dispatcher service function summary

This topic presents a summary of the dispatcher service functions:

- Service manager and service registration
- Event queue processing control
- Parameter modification
- Service command queuing and sending
- Handle management
- Wait object
- Error and tracing log
- Service information retrieval
- Buffer management
- Runtime control
- Application initialization information
- Server command

Service manager and service registration

The following functions enable you to register service managers and services with the dispatcher. Both of these functions are synchronous:

Function	Description
dispRegisterService	Registers a service with the dispatcher at initialization time.
dispRegisterServiceManager	Registers a service manager with the dispatcher at initialization time.

Event queue processing control

The following functions are used to access the event queue. All of these functions are synchronous:

Function	Description
dispGetQueueHandle	Retrieves the queue handle of the specified context.
dispMakeAndQueueEvent	Creates an event and sends it to the application event queue or to the dispatcher event queue.
dispQueueEvent	Queues events to other services or applications.

Parameter modification

By modifying associated parameters, you can alter the characteristics of many Natural Access services. The following functions enable you to obtain or change parameter information. All of these functions are synchronous:

Function	Description
dispCheckParmByName	Verifies the existence and returns the value of a parameter field or structure.
dispGetParmByName	Retrieves the value of parameter field or structure specified by name.
dispGetParms	Returns parameter values for a specified parameter structure.
dispSetParmByName	Permanently modifies the value of a default parameter.

Service command queuing and sending

Service SPI functions call **dispSendCommand** to send commands to the service implementation. This function is synchronous.

Handle management

The following functions enable you to create, modify, and destroy service API handles. All of these functions are synchronous:

Function	Description
dispAttachHandle	Associates a handle with a Natural Access context or object descriptor.
dispCreateHandle	Creates a client-side API handle and associates it with a server-side object handle.
dispDestroyHandle	Destroys the API handle and clears its associated value.
dispFindHandle	Returns the client-side handle associated with a server-side handle.
dispFindHandleByName	Returns the client-side handle associated with the name of a service object.
dispGetHandleValue	Retrieves information associated with a client-side API handle: context handle, server-side service object handle, and service identifier.
dispGetHandleValueEx	Retrieves all information associated with a client-side API handle: context handle, server-side service object handle, service object name, and service identifier.
dispSetHandleValue	Associates a server-side object handle with a client-side API handle.
dispSetHandleValueEx	Associates a service object name with a client-side API handle.
dispQueueCommand	Adds a command message to the context's dispatcher queue.

Wait object

The following functions enable service managers or services to associate wait objects with Natural Access queues. All of these functions are synchronous:

Function	Description
dispFindWaitObject	Returns an internal wait object representing the operating system-specific wait object.
dispRegisterWaitObject	Registers wait objects and their processing functions with a Natural Access queue.
dispUnregisterWaitObject	Unregisters wait objects and their processing functions from a Natural Access queue.

Error and tracing log

The following functions enable you to control service error logging and tracing. All of these functions are synchronous:

Function	Description
dispApiError	Invokes a user defined error handler, logs a trace message, and returns the error code.
dispError	Logs an error to the Natural Access trace log.
dispGetTracePointer	Retrieves the address of the global trace mask at process initialization.
dispLogTrace	Logs a trace message and buffer to the Natural Access trace log.

Service information retrieval

dispGetText returns the text representation of errors, reasons, events and commands. This function is synchronous.

Buffer management

The following functions aid service developers in the use of data buffers within the Natural Access client-server development environment. All of these functions are synchronous:

Function	Description
dispAllocBuffer	Allocates buffer of memory managed by the Natural Access dispatcher.
dispFreeBuffer	Releases a buffer of memory managed by the Natural Access dispatcher.
dispSetAsyncIn	Designates a command buffer as being used for asynchronous input.
dispSetAsyncOut	Designates a command buffer as being used for asynchronous output.

Runtime control

The following functions enable service managers to use the dispatcher runtime control mechanisms. All of these functions are synchronous:

Function	Description
dispAddRTC	Requests that a service action be invoked when a runtime control condition occurs.
dispGetActiveRTC	Returns the runtime control conditions and actions that are registered on the Natural Access context.
dispGetNumRTC	Returns the number of runtime control conditions that are registered on the Natural Access context.
dispGetNumRTCAliases	Returns the number of runtime control aliases added to the RTC database by services on the context.
dispGetRTCAliases	Returns the runtime control aliases added to the RTC database by services on the context.
dispRegisterRTCAlias	Registers an alias with the dispatcher to be associated with a service specific event.
dispRemoveRTC	Requests that the dispatcher's notification about a runtime control condition be removed.
dispUnregisterRTCAlias	Unregisters an event alias previously registered with the dispatcher.

Application initialization information

dispGetAppCtl returns information about default Natural Access usage by the application. This function is synchronous:

Server command

dispShutdown shutdowns or restarts either the Natural Access server (*ctdaemon*) or the system hosting the server. This function is synchronous.

Using the dispatcher service function reference

This section provides an alphabetically ordered reference to the Natural Access dispatcher functions. A prototype of each function is shown with the function description and details of all arguments and return values. A typical function description includes:

Prototype	The prototype is shown followed by a listing of the function's arguments. NMS Communications data types include: • WORD16-bit unsigned • DWORD32-bit unsigned • INT1616-bit signed • INT3232-bit signed • BYTE8-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.
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.

dispAddRTC

Registers a condition and corresponding action in the dispatcher's RTC database.

Prototype

DWORD **dispAddRTC** (CTAHD ***ctahd***, RTC_CONDITION ***pcondition***, RTC_ACTION ***paction***)

Argument	Description
ctahd	Handle to a Natural Access context.

pcondition	Pointer to an RTC_CONDITION structure describing an RTC condition:
	`typedef struct _RTC_CONDITION`
	`{`
	`  RTC_ATTRIBUTES Attribute; /* Characteristics of RTC condition */`
	`  DWORD  EventID;      /* Service's event ID for condition      */`
	`  DWORD  EventValue;   /* Specific value sent with the event    */`
	`                       /*  - used with Attribute RTC_WITH_VALUE */`
	`  char   Alias[ RTC_NAME_LEN ];`
	`                       /* ASCII name of RTC condition           */`
	`} RTC_CONDITION;`
	See the Details section for field descriptions.

paction	Pointer to an RTC_ACTION structure describing an action for the calling service to take when RTC condition ***pcondition*** is met:
	`typedef struct _RTC_ACTION`
	`{`
	`  DWORD  Consumer;     /* Service ID of RTC consumer            */`
	`  DWORD  ActionCmd;    /* Special command sent to service on    */`
	`                       /* recognition of condition              */`
	`} RTC_ACTION;`
	See the Details section for field descriptions.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Indicates any of the following: • The consumer service is not opened on the context • The RTC condition EventId is for a service not attached to the context's CTA queue (see **ctaCreateQueue**). • The Alias parameter is a zero length string, or is a pointer. • The ActionCmd is not a Natural Access command value.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for RTC object.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispAddRTC adds an RTC condition/action pair to the dispatcher's database. RTC_CONDITION describes one or more events from a specific service. RTC_ACTION specifies a command for the dispatcher to send to the calling service when an event is generated that is described by RTC_CONDITION.

The following table describes the fields in RTC_CONDITION:

Field	Description
Attribute	Set to one of the following: • RTC_NONE: Specifies that the RTC condition is met when the dispatcher receives event EventID with any reason code, or an event with alias Alias. • RTC_EVT_WITH_ONE_VALUE: Specifies that the RTC condition is met when the dispatcher receives event EventID with a specific reason code. • RTC_ALL_PROVIDER_EVENTS: Specifies that the RTC condition is met when the dispatcher receives any event from a specific service.
EventID	Set as follows, depending upon the Attribute setting: • If Attribute is set to RTC_NONE and Alias is a zero-length string, set to the event ID of the trigger event. • If Attribute is set to RTC_EVT_WITH_ONE_VALUE, set to the event ID of the trigger event and specify an event value in the EventValue field. • If Attribute is set to RTC_ALL_PROVIDER_EVENTS, set to the event ID of an event from the provider service. The dispatcher infers the service name from the event. Any event from the service triggers the action. • If Attribute is set to RTC_NONE and Alias is a non-zero-length string, EventID is ignored (Alias specifies the condition).
EventValue	If Attribute is set to RTC_EVT_WITH_ONE_VALUE, specifies the value to look for. Otherwise, EventValue is ignored.
Alias	Specifies an alias name. If the dispatcher detects an event registered to the alias name (using **dispRegisterRTCAlias**), it triggers the action. To use aliases, set Attribute to RTC_NONE.

The following table describes the fields in RTC_ACTION:

Field	Description
Consumer	Service ID of the service to direct to perform the action.
ActionCmd	Action command to send to the service manager, in the form of a generic DISP_COMMAND structure.

When **dispAddRTC** is invoked, the dispatcher calls the provider's **xxxAddRTC** function if the **xxxAddRTC** function is present in the provider's set of binding functions. The service writer defines the actions taken by the service when this function is called.

Note: If **dispAddRTC** is invoked with an alias, **xxxAddRTC** is not invoked until the alias is registered with the dispatcher (using **dispRegisterRTCAlias**).

dispAddRTC returns SUCCESS even if the RTC provider service is not opened at the time that **dispAddRTC** is invoked, or if the Alias is not yet registered with the dispatcher. Neither of these situations is an error condition.

See also

dispRemoveRTC

dispAllocBuffer

Allocates a buffer owned and managed by the Natural Access dispatcher.

Prototype

DWORD **dispAllocBuffer** (void ****buffer***, DWORD ***size***)

Argument	Description
buffer	Pointer to the returned address of the allocated buffer.
size	Size of buffer to allocate (in bytes).

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL or ***size*** is 0.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory.

Details

dispAllocBuffer allocates a buffer that is managed by the Natural Access dispatcher. If you are passing **dispAllocBuffer** in an event message, set the CTA_INTERNAL_BUFFER in the buffer's associated ***size*** field. When the buffer is sent in an event, either the dispatcher frees the buffer when event delivery is complete or the application frees the buffer.

See also

dispFreeBuffer

dispApiError

Called by service API functions on errors to invoke a user-defined error handler, log a trace message, and return the error code.

Prototype

DWORD **dispApiError** (CTAHD ***ctahd***, DWORD ***errcode***, const char ****func***, const char ****info***)

Argument	Description
ctahd	Context handle.
errcode	Error code returned by API function.
func	Pointer to the name of the API function reporting the error.
info	Pointer to the character string with additional information about the error.

Return values

This function returns the return value of the application-supplied error handler, or ***errcode*** if there is no handler.

Details

dispApiError is usually called when returning an error from an API function call by using the CTAAPIERROR macro or the CTAAPIERRORINFO macro. It returns the error code passed in as an argument so it can be called in a function return statement.

If tracing is enabled and either the application specified CTA_EXIT_ON_ERRORS or CTA_REPORT_ERRORS as the error handler, or the global or context specific trace mask has the CTA_TRACEMASK_API_ERRORS bit set, **dispApiError** logs a trace record to *ctdaemon*.

The Natural Access error macros, CTAAPIERRORINFO and CTAAPIERROR, are defined in *ctadisp.h*.

See also

dispError, **dispLogTrace**

dispAttachHandle

Creates a client handle to the object referenced by the descriptor and attaches to the associated object.

Prototype

DWORD **dispAttachHandle** (char ***descriptor**, unsigned ***hd**)

Argument	Description
descriptor	Pointer to an object handle descriptor.
hd	Pointer to the returned handle for a specified descriptor.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid descriptor or NULL handle pointer.
CTAERR_NOT_FOUND	Unable to locate object.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispAttachHandle enables services opened on shared contexts to share service objects. To use this function, the service must be provided with an object descriptor by the application.

This function creates a new entry in the dispatcher's handle mapping table and returns a client-side service object handle referencing the shared object.

dispCheckParmByName

Verifies the existence and returns the value of a parameter field or structure.

Prototype

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

Argument	Description
ctahd	Context handle.
parmname	Pointer to the parameter name or field.
buffer	Pointer to a buffer to receive the parameter value.
size	Size of the memory area.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Required pointer argument is NULL.
CTAERR_BAD_SIZE	***size*** is too small to receive the data structure.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	The specified parameter does not exist or the owning service was not initialized.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispCheckParmByName verifies the existence and returns the value of a parameter field or structure. This function is the same as **dispGetParmByName** except that it does not log an error if one occurs. The service is indicated as the first part of the parameter name string, for example, ADI in adi.play.gain.

See also

dispGetParms

dispCreateHandle

Creates a handle and associates it with a service specific value.

Prototype

DWORD **dispCreateHandle** (CTAHD ***ctahd***, unsigned ***svcid***, unsigned ***value***, unsigned ****hd***)

Argument	Description
ctahd	Context handle.
svcid	Identifier of the requesting service.
value	Single 32-bit value associated with the handle.
hd	Pointer to the returned handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***hd*** pointer is NULL.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for the handle.
CTAERR_OUT_OF_RESOURCES	Unable to find free space in internal tables (maximum number of handles is 64 K).
CTAERR_SERVICE_NOT_AVAILABLE	The specified service is not initialized.

Details

dispCreateHandle is typically used by services to create API (client) side service handles that can then be used by API functions that implicitly reference the ***ctahd***. Services that allow multiple opens of objects that they manage can use this function to create a handle and associate with it the ***ctahd*** and a server side handle that references the real object. The Voice Message service uses this function to create ***vcehd***s for voice objects. The Switching service uses this function to create ***swihd***s for open switches.

This extra handle translation is necessary since the API must call **dispSendCommand** with a valid ***ctahd*** as well as its own server side object handle. Since the API functions do not have any connection to their server side contexts in future client-server versions of Natural Access, a mapping needs to be maintained on the client side.

If only the server-side object handle is passed back to the application, then every client side API would need to perform a mapping from object handle to ***ctahd*** including any locks necessary on the data structures to make the code thread safe. Instead, this functionality is incorporated into the dispatcher and made available to service writers.

An additional problem with using server-side handles is that servers on different nodes could return identical handles. Although these handles would be on different contexts, they could be in the same client process. The client-side object handles must be different from the server object handles. This is guaranteed to be the case by using the unique handle returned by this function.

The dispatcher goes through its handle table when a service is closed and removes all handle associations for the specified context and service ID. This works because all the information that the API manages is in the dispatcher handle table. The server-side handle and its information is managed by the service and must be closed and cleaned up on the invocation of the close service binding function **xxxCloseService**.

dispSetHandleValue and **dispGetHandleValue** are used to modify and retrieve the server side handle associated with the API handle. **dispDestroyHandle** is used by the API to destroy a handle when it is being closed.

See also

dispAttachHandle, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValueEx**, **dispSetHandleValueEx**

dispDestroyHandle

Destroys the API handle and clears its associated value.

Prototype

DWORD **dispDestroyHandle** (unsigned ***hd***)

Argument	Description
hd	Handle returned by **dispCreateHandle**.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	An invalid handle was passed as an argument to this function.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.

Details

dispDestroyHandle is used when an API closes or destroys a server side object to remove the association between the API handle and the server-side handle.

API handles are a limited resource so they must be destroyed when the object they reference is destroyed. Otherwise, a handle leak occurs and future API calls can return CTAERR_OUT_OF_RESOURCES.

API handles associated with open server-side objects are automatically cleaned up when a service is closed. Services do not need to worry about handle leaks. Services must ensure that the server-side objects are cleaned up properly, or other resource leaks will occur.

See also

dispAttachHandle, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValue**, **dispGetHandleValueEx**, **dispSetHandleValue**, **dispSetHandleValueEx**

dispError

Logs an error to the Natural Access trace log.

Prototype

DWORD **dispError** (CTAHD ***ctahd***, DWORD ***errcode***, unsigned ***svcid***, char ****func***, char ****file***, unsigned ***lineno***, char ****info***, unsigned ***logerror***)

Argument	Description
ctahd	Context handle.
errcode	Error code.
svcid	ID of the service that is reporting the error.
func	Pointer to the name of the function reporting the error.
file	Pointer to the name of the file containing the function.
lineno	Line number on which the error is reported.
info	Pointer to additional information about the error.
logerror	Boolean to determine whether error should be logged to Natural Access trace log.

Return values

errcode

Details

If ***logerror*** is TRUE (non-zero), **dispError** logs an error to the trace log. The error code is then returned as the result of this function.

dispError is typically called using one of the following macros: CTAERROR, CTALOGERROR, or CTALOGERRORINFO. These macros return the error value passed in. They can also be used directly to return the error value from a command function.

```
#define CTAERROR (crhd, err) \
    dispError((crhd),(err),CTA_SYS_SVCID,\
    funcname,__FILE__, __LINE__,NULL,0)

#define CTALOGERROR (crhd, err, svcid) \
    dispError((crhd),(err),(svcid),\
    funcname, __FILE__, __LINE__,NULL, 1)
#define CTALOGERRORINFO (crhd, err, svcid, info) \
    dispError((crhd),(err),(svcid),funcname,\
    __FILE__, __LINE__, info, 1)
```

The following table describes each of these macros:

Macro	Description
CTAERROR	CTAERROR (***ctahd, errorcode***) is used to return an error without logging information to *ctdaemon*. The dispatcher passes the error back to the caller and does nothing with it. Use this macro when minimum reporting is required and the error will be returned and logged by the caller. If no reporting is desired (for internal service functions), return the error code directly to the calling function.
CTALOGERROR	CTALOGERROR (***ctahd, errcode, svcid***) is used when logging a trace message associated with the service ID, error code, function name, file, and line number. This macro expands to a call to **dispError** with the arguments as informational elements. If tracing is enabled, an error is logged to *ctdaemon* and an error appears in the trace log. This method allows services to report errors. Developers can diagnose exactly which service and function caused the error even if the error is reported by multiple services.
CTALOGERRORINFO	CTALOGERRORINFO (***ctahd, errcode, svcid, info***) does the same thing as CTALOGERROR except it allows an extra character string argument to be passed. This string gets logged in the trace buffer as additional information associated with the error. If used, this string should contain a clear explanation why the error occurred so that diagnosis of error conditions is easy and efficient. This is important when generic errors are returned from multiple functions in the same service. CTALOGERROR and CTALOGERRORINFO can be used to report error conditions without returning an error or they can be used in a return statement. Both macros return the error code argument as a result.

These macros expand to pass the function name (***func***), file name (***file***), and line number (***lineno***) as arguments to **dispError**. Use the macro CTABEGIN (***func***) at the beginning of the function calling these macros to establish the function name as a local variable to be referenced in the macro expansion.

Call **dispError** through the appropriate macros whenever an error condition is encountered. It should be done unconditionally since all errors should be logged if logging is enabled. Services wanting to filter certain internal errors can check the CTA_TRACEMASK_SVC_ERRORS bit in the service or global trace mask before calling **dispError**. The CTA_TRACEMASK_SVC_ERRORS bit is not checked by **dispError**.

See also

dispApiError, **dispLogTrace**

dispFindHandle

Returns the client-side handle associated with a server-side handle.

Prototype

DWORD **dispFindHandle** (CTAHD ***ctahd***, unsigned ***svcid,*** unsigned ***value***, CTAHD ****ctaClntHd***)

Argument	Description
ctahd	Context handle.
svcid	Identifier of the service associated with the handle.
value	Value associated with the handle created by **dispCreateHandle**.
ctaClntHd	Pointer to the returned object handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	No object handle is associated with the specified arguments.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.

Details

dispFindHandle compares the service ID, the Natural Access context handle, and the value (for example: service object handle) to the values in the entries maintained in the dispatcher's internal handle mapping table. If an entry is found matching all these parameters, the entry's associated client-side handle is returned to the caller.

See also

dispAttachHandle, **dispDestroyHandle**, **dispFindHandleByName**, **dispGetHandleValue**, **dispGetHandleValueEx**, **dispSetHandleValue**, **dispSetHandleValueEx**

dispFindHandleByName

Returns the client-side object handle associated with the named of service object.

Prototype

DWORD **dispFindHandleByName**(CTAHD ***ctahd***, unsigned ***svcid***, char ****name***, CTAHD ****ctaClntHd***)

Argument	Description
ctahd	Context handle.
svcid	Identifier of the service associated with the handle.
name	Pointer to the name of the service object.
ctaClntHd	Pointer to the returned client-side object handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_CTAHD	Object cannot be found.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispFindHandleByName compares the service ID, the context handle, and the service object name to the entries in the dispatcher's internal handle mapping table. If an entry is found matching all these values, the entry's associate client-side handle is returned to the caller.

See also

dispAttachHandle, **dispCreateHandle**, **dispDestroyHandle**, **dispFindHandle**, **dispGetHandleValue**, **dispGetHandleValueEx**, **dispSetHandleValue**, **dispSetHandleValueEx**

dispFindWaitObject

Returns a pointer to the internal dispatcher array entry containing the specified wait object.

Prototype

DWORD **dispFindWaitObject** (CTAQUEUEHD ***ctaQueueHd***, CTA_WAITOBJ* ***pctaWaitObj***, CTA_WAITOBJ** ***ppctaWaitObjFound***)

Argument	Description
ctaQueueHd	Queue handle.
pctaWaitObj	Pointer to a wait object.
ppctaWaitObjFound	Pointer to the returned entry containing the wait object.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Wait object is not associated with a queue.
CTAERR_INVALID_CTAQUEUEHD	***ctaQueueHd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access was not initialized.

Details

dispFindWaitObject returns the address of the internal array entry where the matching wait object is located. This function is currently needed for NMS's abstracted IPC communications layer (TSI library).

dispFreeBuffer

Releases a Natural Access buffer that was originally allocated using **dispAllocBuffer**.

Prototype

DWORD **dispFreeBuffer** (void ***buffer***)

Argument	Description
buffer	Buffer pointer returned by **dispAllocBuffer**.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL

Details

dispFreeBuffer releases a Natural Access buffer that was originally allocated using **dispAllocBuffer**.

dispGetActiveRTC

Returns a list of RTC conditions and actions that were added to the Natural Access context.

Prototype

DWORD **dispGetActiveRTC** (CTAHD ***ctahd***, unsigned ***maxRTCs***, RTC_CONDITION ***pconditions[]***, RTC_ACTION ***pactions[]***, unsigned ****pnumRTCs***)

Argument	Description
ctahd	Handle to a Natural Access context.
maxRTCs	Maximum number of entries in ***pconditions***/***pactions*** arrays.
pconditions	Pointer to the RTC_CONDITION array.
pactions	Pointer to the RTC_ACTION array.
pnumRTCs	Returned number of entries in ***pconditions***/***pactions*** arrays.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***pnumRTCs***, ***pconditions***, or ***pactions*** is NULL, or ***maxRTCs*** is 0.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetActiveRTC returns a list of RTC conditions and associated actions that were added to the specified context. These arrays are parallel in that the same indexed entry is associated to one another. Call **dispGetNumRTC** to determine the amount of space that needs to be allocated for the return buffers (***pconditions, pactions***).

This function returns all ***pconditions***/***pactions*** arrays in the database, up to ***maxRTCs***. This function cannot be used to iterate through the database.

See also

dispAddRTC

dispGetAppCtl

Returns information about default Natural Access usage by the application.

Prototype

DWORD **dispGetAppCtl** (unsigned ***ctltype***, void ****buffer***, unsigned ***size***)

Argument	Description
clttype	Type of application information desired. Valid value is CTAAPP_CTAFLAGS.
buffer	Pointer to buffer (sized to DWORD) into which to place the application flags.
size	Size of ***buffer***.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** is NULL, ***size*** is 0, or invalid ***clttype*** parameter value.
CTAERR_BAD_SIZE	***buffer*** size is too small.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispGetAppCtl returns information about how the application and the default system settings control the behavior of Natural Access.

The return value the following set of flags:

Flag	Description
CTA_NOTIFY_UPDATE_WAITOBJS	Natural Access sends CTAEVN_UPDATE_WAITOBJS when there is a change to the internal array of Natural Access wait objects.
CTA_MODE_SERVER	The Natural Access Server functionality runs in a Natural Access Server *(ctdaemon)* that is separate from the application, but on the same host.
CTA_MODE_LIBRARY	The Natural Access Server functionality runs within the process address space.
CTA_COMMS_SERIALIZE	When set, commands from all queues sent to the server are serialized (this is the default). This mode requires fewer system resources, allowing more event queues to be created.
CTA_COMMS_NOTSERIALIZE	When set, 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.
CTA_CONTEXTNAME_REQUIRE_UNIQUE	When set, Natural Access cannot modify the context name to ensure uniqueness. If two applications specify the same context name, the second application receives an error.
CTA_CONTEXTNAME_CREATE_UNIQUE	When set, allows the local server to modify context names specified by client applications so that context names are not duplicated (default).

See **ctaInitialize** in the *Natural Access Developer's Reference Manual* for more details on these flags.

dispGetHandleValue

Retrieves the value associated with the given API handle.

Prototype

DWORD **dispGetHandleValue** (unsigned ***hd***, CTAHD ****ctahd***, unsigned ****value***)

Argument	Description
hd	Handle returned by **dispCreateHandle**.
ctahd	Pointer to a handle.
value	Pointer to the returned value.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	The specified ***hd*** is not valid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispGetHandleValue retrieves the value that was associated with the given API handle either when it was created using **dispCreateHandle** or when it was explicitly set using **dispSetHandleValue**. This value can be any unsigned integer. This value is typically a number that represents a server-side object and can be used as an index into a context-specific array by the service.

See also

dispAttachHandle, **dispDestroyHandle**, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValueEx**, **dispSetHandleValueEx**

dispGetHandleValueEx

Retrieves all information associated with a client-side API handle, including context handle, server-side service object handle, service object name, and service identifier.

Prototype

DWORD NMSAPI **dispGetHandleValueEx** (unsigned ***hd***, CTAHD ****ctahd***, unsigned ****svcid***, unsigned ****value***, char ****name***)

Argument	Description
hd	Handle returned by **dispCreateHandle**.
ctahd	Pointer to a handle.
svcid	Pointer to the identifier of the service associated with the handle.
value	Pointer to receive the returned value.
name	Pointer to a name associated with the object (with **dispSetHandleValueEx**).

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	The specified ***hd*** is not valid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispGetHandleValueEx retrieves the information in the dispatcher's internal handle mapping table that is associated with the passed-in client-side handle. The information returned is:

- The associated context handle
- The service ID of the service that created the handle
- The server-side object handle
- The service object name

value can be any unsigned integer. ***value*** is typically a number that represents a server-side object and can be used as an index into a context-specific array by the service. ***name*** is used for service object sharing.

See also

dispAttachHandle, **dispCreateHandle**, **dispDestroyHandle**, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValue**, **dispSetHandleValue**

dispGetNumRTC

Returns the number of RTC conditions that were added to the Natural Access context.

Prototype

DWORD **dispGetNumRTC** (CTAHD ***ctahd***, unsigned ****pnumCondtions***)

Argument	Description
ctahd	Handle to a Natural Access context.
pnumConditions	Pointer to the returned number of RTCs associated with context.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***pnumConditions*** is NULL.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetNumRTC returns the number of RTC conditions that were added to the Natural Access context.

See also

dispGetActiveRTC

dispGetNumRTCAliases

Returns the number of RTC aliases that are registered on the Natural Access context.

Prototype

DWORD **dispGetNumRTCAliases** (CTAHD ***ctahd***, unsigned ****pnumAliases***)

Argument	Description
ctahd	Handle to a Natural Access context.
pnumAliases	Pointer to returned number of aliases registered with the dispatcher.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***pnumAliases*** is NULL.
CTAERR_INVALID_CTAHD	The specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetNumRTCAliases returns the number of RTC aliases that are registered.

See also

dispGetRTCAliases

dispGetParmByName

Retrieves the value of a parameter field or structure specified by name.

Prototype

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

Argument	Description
ctahd	Handle to a Natural Access context.
parmname	Pointer to a parameter name or field.
buffer	Pointer to buffer to receive the parameter value.
size	Size of the memory area.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Required pointer argument is NULL.
CTAERR_BAD_SIZE	***buffer*** is too small to receive the data structure.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	Specified parameter does not exist or owning service is not initialized.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetParmByName retrieves the value for the parameter identified as ***parmname***.

ctahd must refer to a valid context since this function only retrieves the parameters from the context defaults. Services cannot retrieve or modify global defaults.

Natural Access writes the values to ***buffer***. The ***size*** argument specifies the amount of available memory at ***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. 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.

See also

dispCheckParmByName, **dispGetParms**, **dispSetParmByName**

dispGetParms

Returns parameter values for a specified parameter structure.

Prototype

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

Argument	Description
ctahd	Context handle.
parmid	Pointer to the parameter identifier.
buffer	Pointer to a buffer to receive the parameter value.
size	Size of the memory area.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Required pointer argument is NULL.
CTAERR_BAD_SIZE	***size*** is too small to receive the data structure.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	Specified parameter does not exist or the owning service is not initialized.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetParms retrieves the parameter values for the entire parameter structure identified by the ***parmid***.

ctahd must refer to a valid context since this function only retrieves the parameters from the context defaults. Services cannot retrieve or modify global defaults.

size specifies the amount of available memory at ***buffer*** and must be large enough to contain the returned parameter structure.

See also

dispCheckParmByName, dispGetParmByName, **dispSetParmByName**

dispGetQueueHandle

Retrieves the queue handle of the specified context.

Prototype

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

Argument	Description
ctahd	Context handle.
ctaqueuehd	Pointer to the returned queue handle.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***ctaqueuehd*** is NULL.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispGetQueueHandle retrieves the queue handle associated with the context.

dispGetRTCAliases

Returns a list of RTC aliases that are registered on the Natural Access context.

Prototype

DWORD **dispGetRTCAliases** (CTAHD ***ctahd***, unsigned ***maxAliases***, RTC_ALIAS ***aliases[]***, unsigned ****pnumAliases***)

Argument	Description
ctahd	Handle to a Natural Access context.
maxAliases	Maximum number of entries in the ***aliases*** array.

aliases	Pointer to the RTC_ALIAS array, defined as:
	`typedef struct`
	`{`
	`  DWORD EventID;  /* RTC condition event */`
	`  char  Alias[ RTC_NAME_LEN ];  /* ASCII name  */`
	`}  RTC_ALIAS;`

pnumAliases	Returned number of entries in the ***aliases*** array

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***aliases*** is NULL.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetRTCAliases returns a list of RTC condition aliases that are registered with RTC on the specified context. Call **dispGetNumRTCAliases** to determine the amount of space that needs to be allocated for the return buffer (***aliases***).

See also

dispRegisterRTCAlias, **dispUnregisterRTCAlias**

dispGetText

Returns the text representation of errors, reasons, events, and commands.

Prototype

DWORD **dispGetText** (DWORD ***code***, char ****buffer***, unsigned ***size***)

Argument	Description
code	Error, reason, event, or command code.
buffer	Pointer to the returned text.
size	Size of ***buffer*** in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***buffer*** passed as NULL.
CTAERR_BAD_SIZE	***size*** passed as zero (0).
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetText returns a text string in ***buffer*** corresponding to the Natural Access error, reason, event, or command code. It finds a service matching the service ID indicated in the upper word of ***code***. All services with the same service ID should be capable of translating ***code*** to its text representation.

If ***buffer*** passed in is not large enough to contain the full string, only as many characters are copied as can fit into ***buffer*** including a null terminator.

See also

xxxFormatMessage, xxxGetText

dispGetTracePointer

Retrieves the address of the global trace mask.

Prototype

DWORD **dispGetTracePointer** (volatile DWORD ****trace_pointer**)

Argument	Description
trace_pointer	Pointer to the returned address of the global trace mask.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***trace_pointer*** is NULL.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispGetTracePointer is typically called at service or service manager initialization time (on the call to **xxxInitializeManager** or **xxxDefineService**) to retrieve the address of the 32 bit global trace mask. This global trace mask is held in shared memory when the *ctdaemon* is running. Otherwise, it is held in a process global. It should be checked, along with the context specific trace mask (refer to **xxxSetTraceLevel**), whenever a service or service manager must make a decision whether to call **dispLogTrace**.

Natural Access defines standard trace mask bits in *ctadef.h*, as well as service specific bits that can be used freely by each service.

dispLogTrace

Logs a trace message and a buffer to the Natural Access trace log.

Prototype

DWORD **dispLogTrace** (CTAHD ***ctahd***, unsigned ***tracesource***, CTA_TRACE_SEVERITY ***traceseverity***, unsigned ***tracetag***, void ***tracevalue***, unsigned ***tracesize***)

Argument	Description
ctahd	Context handle. It is used to retrieve the user-supplied context name and user ID, and is stored in the trace record as the main way of indicating where the trace message originated.
tracesource	Service ID of the service reporting the trace message.
traceseverity	Indicates if the message is informational, a warning, or an error message. See the Details section for acceptable values.
tracetag	Type of the parameter passed by reference in ***tracevalue***. Use either a service defined ***tracetag*** or one of the following values: 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	Specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

If tracing is enabled for the process, **dispLogTrace** writes a trace message and an optional buffer to the shared memory segment owned by *ctdaemon*. It does this in a process and thread-safe manner by acquiring a system-wide trace lock before writing to the shared memory. It then signals an event that causes *ctdaemon* to wake up, read the trace message and buffer, display it on the screen, and, optionally, log it to a file.

If a service wants to log an informational trace message, the service must call **dispLogTrace** directly. It should do this after checking the global and service trace masks to determine if the bits that correspond to the level of tracing necessary to generate the trace record are set.

traceseverity can have the following values specified by the enum CTA_TRACE_SEVERITY in *ctadef.h*:

Value	Use this value for...
CTA_TRACE_SEVERITY_INFO	Command, event, or internal service tracing. These are not error conditions. They are informational tracing information. They do not typically cause a backtrace error dump when backtrace on error mode is specified in the call to **ctaInitialize**.
CTA_TRACE_SEVERITY_WARNING	Warnings. Warnings are not generated by the dispatcher or any core Natural Access service. **Note***:* Warnings are treated as errors by the dispatcher.
CTA_TRACE_SEVERITY_ERROR	Errors.

To define a service-specific trace format, define a trace tag value that contains the service ID in the upper word. When processing trace records containing a service-specific tag, the dispatcher calls **xxxFormatTraceBuffer**.

ctdaemon must be initialized with all the services that could be initialized in applications so that it has access to the binding functions necessary to format service-specific trace records.

Tracetags are defined in the same way as other codes. The high word is reserved for the service ID and the low word must be in the range from 0x4000 to 0x4FFF.

See also

dispApiError, **dispError**, **dispGetTracePointer**, **xxxSetTraceLevel**

dispMakeAndQueueEvent

Creates an event and sends it to the application event queue or to the dispatcher event queue.

Prototype

DWORD **dispMakeAndQueueEvent** (CTAHD ***ctahd***, unsigned ***id***, unsigned ***value***, unsigned ***source***, unsigned ***destination***)

Argument	Description
ctahd	Context handle returned by **ctaCreateContext**.
id	Event ID field of DISP_EVENT.
value	Value field of DISP_EVENT.
source	Source service identifier of the event (addr field of DISP_EVENT).
destination	Destination service identifier of the event (addr field of DISP_EVENT).

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument has an invalid value.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for the queue, the driver, or the context.
CTAERR_SERVICE_NOT_AVAILABLE	Specified destination service is not available.

Details

dispMakeAndQueueEvent is a wrapper function for **dispQueueEvent**. It does the following:

- Creates a DISP_EVENT structure
- Fills in the id, value, and addr fields of the DISP_EVENT structure from the passed arguments
- Calls **dispQueueEvent**, which then places the event on either the application event queue or on the dispatcher event queue corresponding (based on ***destination***) to the ***ctahd***

source is typically the service ID. ***destination*** can be:

- A service ID
- CTA_APP_SVCID
- CTA_SYS_SVCID

dispMakeAndQueueEvent is provided to simplify sending events to other services or to the application when the events only contain an event ID and value.

dispQueueCommand

Adds a command message to a context's dispatcher queue. (This command is available only when using the in-process Natural Access server.)

Prototype

DWORD NMSAPI **dispQueueCommand** (DISP_COMMAND ****ctacmd***);

Argument	Description

ctacmd	Pointer to DISP_COMMAND structure containing the message to add:
	`typedef struct`
	`{`
	`  DWORD id;  /* Command ID (and dest. service ID)  */`
	`  CTAHD ctahd;  /* Natural Access context handle  */`
	`  DWORD size1;  /* If dataptr1 !=NULL => size1 is buffer  */`
	`  /*  size- else message specific data  */`
	`  void *dataptr1; /* Data pointer  */`
	`  DWORD size2;  /* If dataptr2 !=NULL => size2 is buffer  */`
	`  /*  size- else message specific data  */`
	`  void *dataptr2; /* Data pointer  */`
	`  DWORD size3;  /* If dataptr3 !=NULL => size3 is buffer  */`
	`  /*  size- else message specific data  */`
	`  void *dataptr3; /* Data pointer  */`
	`  DWORD objHd;  /*Service object handle (formerly "reserved")*/`
	`  MESSAGE_ADDR addr; /* Source/Destination service ID  */`
	`} DISP_COMMAND;`
	
	`typedef struct`
	`{`
	`  WORD source  /* Source service ID  */`
	`  WORD destination /* Destination service ID  */`
	`} MESSAGE_ADDR`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***ctacmd*** is NULL or id field in the command is invalid.
CTAERR_SERVICE_NOT_AVAILABLE	Service associated with the command is not available on the command context.
CTAERR_NOT_IMPLEMENTED	Command unavailable when not using the in-process Natural Access server.

Details

dispQueueCommand adds a DISP_COMMAND structure to a context's dispatcher queue and signals the default queue wait object.

dispQueueEvent

Queues events to other services or to the application.

Prototype

DWORD **dispQueueEvent** (DISP_EVENT ***ctaevt***)

Argument	Description

ctaevt	Pointer to the event structure to queue. The event structure contains the following fields:
	`typedef struct`
	`{`
	`  DWORD id;  /* Event code (and source service ID)  */`
	`  CTAHD ctahd;  /* Natural Access context handle  */`
	`  DWORD timestamp;/* timestamp  */`
	`  DWORD userid;  /* User ID (defined by ctaCreateContext)  */`
	`  DWORD size;  /* Size and type of buffer if buffer != NULL */`
	`  void *buffer;  /* Buffer pointer  */`
	`  DWORD value;  /* Event status or event-specific data  */`
	`  DWORD extra;  /* Extra internal event field  */`
	`  unsigned objHd; /*Service object handle (formerly "reserved")*/`
	`  MESSAGE_ADDR addr; /* Source/Destination service ID  */`
	`} DISP_EVENT;`
	`typedef struct`
	`{`
	`  WORD source  /* source service ID  */`
	`  WORD destination  /* destination service ID  */`
	`} MESSAGE_ADDR`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***ctaevt*** is NULL.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_IMPLEMENTED	Function was called from the service interface.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for the queue, the driver, or the context.
CTAERR_SERVICE_NOT_AVAILABLE	Specified destination service is not available.

Details

dispQueueEvent can be used by services or service managers to send events to other services or to the application. The event is placed either in the dispatcher internal event queue or in the application queue (based on the destination) associated with the context referenced in the event.

The caller must fill in the id, ctahd, value, and addr fields. The context can be any valid context on any queue. source is typically the service ID. destination can be one of the following values:

- A service ID
- CTA_APP_SVCID
- CTA_SYS_SVCID

The size and buffer fields can be used to pass data to the application. If buffer = NULL, size can contain data.

The size field is also used to indicate to the dispatcher whether the event's buffer is static or dynamic. To do this, set one of the following flags in the field if buffer is not NULL:

Flag	Indicates...
CTA_INTERNAL_BUFFER_STATIC	Static buffer.
CTA_INTERNAL_BUFFER	Dynamic buffer. If the service implementation sets this flag, Natural Access or the application ensures the proper release of this buffer. The service implementation does not need to perform this task. Use **dispAllocBuffer** to allocate this buffer.

For more information about these flags, refer to *DISP_EVENT structure* on page 117.

The dispatcher sets the userid and timestamp fields.

See also

dispMakeAndQueueEvent

dispRegisterRTCAlias

Defines an alias associated with service-specific events. This alias can then be used by other services as an identifier of an RTC condition.

Prototype

DWORD **dispRegisterRTCAlias** (CTAHD ***ctahd***, RTC_ALIAS ****alias***)

Argument	Description
ctahd	Handle to a Natural Access context.

alias	Pointer to the RTC_ALIAS array, defined as:
	`typedef struct`
	`{`
	`  DWORD EventID;  /* RTC condition event */`
	`  char  Alias[ RTC_NAME_LEN ]; /* ASCII name  */`
	`} RTC_ALIAS;`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***alias*** is NULL, ***alias*** contains a zero length Alias string, or the EventID is not a valid service event.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for the RTC_ALIAS structure.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispRegisterRTCAlias creates an association between the RTC condition and the specified ASCII alias name and records it within the dispatcher's RTC database. The same name can be assigned to multiple service events.

To receive RTC notification whenever an event in the alias group occurs, a consumer service uses **dispAddRTC** to register the alias as a condition and associate an action with it. Then when an event associated with an alias occurs, the dispatcher notifies any consumer services registered with the alias.

See also

dispUnregisterRTCAlias

dispRegisterService

Registers a service with the Natural Access dispatcher at initialization time.

Prototype

DWORD **dispRegisterService** (const char ***svcname***, unsigned ***svcid***, const char ***svcmgrname***, const CTAINTREV_INFO ***previnfo***, const CTAREQSVC_INFO ****preqsvc []***, unsigned ***nreqsvcs***, const CTAPARM_DESC ****const *stdparmdesc***, const CTAPARM_DESC ****const *extparmdesc***)

Argument	Description
svcname	Pointer to the name of the service.
svcid	Service identifier used in commands, errors, reasons, and events (assigned by NMS).
svcmgrname	Pointer to the service manager name.

previnfo	Pointer to the service revision information structure containing the following fields:
	`typedef struct`
	`{`
	`  DWORD size;  /* size of the structure */`
	`  DWORD majorrev  /* major revision of service/manager */`
	`  DWORD minorrev;  /* minor revision of service/manager */`
	`  char buildate [12]; /* build date, "mmm dd yyyy\0" */`
	`  DWORD reqdisplevel; /* Required compat level of dispatcher */`
	`  DWORD expapilevel;  /* Exported compat level of Service API */`
	`  DWORD expspilevel;  /* Exported compat level of Service SPI */`
	`} CTAINTREV_INFO;`

preqsvc	Pointer to array of structures containing information on services required by this service:
	`typedef struct`
	`{`
	`  char *svcname;  /* Name of required service. */`
	`  DWORD svcid;  /* Service ID of required service. */`
	`  DWORD reqspilevel; /* Required compat level of Service SPI */`
	`} CTAREQSVC_INFO;`

nreqsvcs	Number of required service structures in ***preqsvc***.

stdparmdesc	Pointer to an array of pointers to standard parameter descriptors that are managed for the service by Natural Access. The array is terminated with a NULL character. Each parameter descriptor structure in the array has the following definition:
	`typedef struct`
	`{`
	`  DWORD parm_id;`
	`  void *parm;  /* Pointer to defaults (binary) */`
	`  const CTAPARM_FIELD *descr; /* Pointer to field descriptor */`
	`  DWORD fieldcnt;  /* Number of fields in descr */`
	`} CTAPARM_DESC;`
	Field descriptors have the following definition:
	`typedef struct`
	`{`
	`  char  *name;  /* Name of the field. */`
	`  DWORD offset; /* Byte offset from base or # of fields */`
	`  /* if received zero (0) in array. */`
	`  DWORD size;  /* Size of the field. */`
	`  DWORD format; /* Format specifier. */`
	`  DWORD units;  /* Units specifier. */`
	`} CTAPARM_FIELD;`
	Note: These structures are automatically generated by *pf2src* from a *.pf* file.

Argument	Description
extparmdesc	Pointer to an array of pointers to non-standard (extension) parameter descriptors that are managed for the service by Natural Access. The array is terminated with a null character. The structure is defined in ***stdparmdesc***.

Return values

Return value	Description
SUCCESS	
CTAERR_ALREADY_INITIALIZED	**ctaInitialize** was already called by the application.
CTAERR_BAD_ARGUMENT	Function argument has an invalid value or a required pointer argument is NULL.
CTAERR_DUPLICATE_EXTPARMS	External parameters must be unique.
CTAERR_INCOMPATIBLE_PARMS	Standard parameters did not match.
CTAERR_INCOMPATIBLE_REVISION	Service or service manager is incompatible with the revision of Natural Access that is being used.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory.
CTAERR_SHAREMEM_ACCESS	Failed to access shared memory.

Details

dispRegisterService registers a service with the Natural Access dispatcher at initialization time. This registration includes establishing the parameters that Natural Access manages for the service. This function must be called for every service that provides an API to a Natural Access application. It is typically called from **xxxDefineService**.

Parameter descriptor tables are used directly by Natural Access and are not copied. They must remain accessible in the process address space throughout the lifetime of the process that initialized Natural Access. Parameter default values are copied to shared memory if this call is made within *ctdaemon*. In application processes, the defaults are taken either from shared memory or directly from the defaults registered with the parameter descriptor tables. Parameter default values must also remain accessible to Natural Access as long as the process is active.

Revision information is used to verify dispatcher compatibility. It is also available for checking the compatibility of the application API and the service SPI. The SPI compatibility level is used to verify that other services that make calls to this service (services that have this service listed in its list of required services) expect the same SPI. The required service list indicates the list of services whose SPIs are used by this service.

dispRegisterServiceManager

Registers a service manager with the Natural Access dispatcher at initialization time.

Prototype

DWORD **dispRegisterServiceManager** (const char ***svcmgrname***, CTASVCMGR_FNS ***svcmgrfns***, const CTAINTREV_INFO ***previnfo***)

Argument	Description
svcmgrname	Pointer to the name of the service manager.
svcmgrfns	Pointer to the service manager binding functions. This structure has the following definition:

```
typedef DWORD (NMSSTDCALLPTR PFUNC)();

typedef struct _CTASVCMGR_FNS
    {
    DWORD size;
    DWORD (NMSSTDCALLPTR xxxDefineService)(
       char *svcname);

    DWORD (NMSSTDCALLPTR xxxAttachServiceManager)(
     CTAQUEUEHD ctaqueuehd, unsigned mode, void **queuecontext);

    DWORD (NMSSTDCALLPTR xxxDetachServiceManager)(
     CTAQUEUEHD ctaqueuehd, void *queuecontext);

    DWORD (NMSSTDCALLPTR xxxOpenServiceManager)(
     CTAHD ctahd, void *queuecontext, void **mgrcontext);

    DWORD (NMSSTDCALLPTR xxxCloseServiceManager)(
     CTAHD ctahd, void *mgrcontext);

    DWORD (NMSSTDCALLPTR xxxOpenService)(
     CTAHD ctahd, void *mgrcontext,
   char *svcname,unsigned svcid,
     CTA_MVIP_ADDR *mvipaddr,CTA_SERVICE_ARGS*svcargs);

    DWORD (NMSSTDCALLPTR xxxCloseService)(
     CTAHD ctahd, void *mgrcontext, char *svcname,
     unsigned svcid);

    DWORD (NMSSTDCALLPTR xxxProcessEvent)(
     CTAHD ctahd, void *mgrcontext, DISP_EVENT *ctaevt);

    DWORD (NMSSTDCALLPTR xxxProcessCommand)(
     CTAHD ctahd, void *mgrcontext, DISP_COMMAND *ctaevt);

    /****** for future use ******/
    DWORD (NMSSTDCALLPTR xxxAddRTC)(
     CTAHD ctahd, void *mgrcontext,
     RTC_CONDITION *rtccondition, RTC_ACTION *rtcaction);

    /****** for future use ******/
    DWORD (NMSSTDCALLPTR xxxRemoveRTC)(
     CTAHD ctahd, void *mgrcontext,
     RTC_CONDITION *rtccondition, RTC_ACTION *rtcaction);

    const char * (NMSSTDCALLPTR xxxGetText)(
     unsigned code);

    DWORD (NMSSTDCALLPTR xxxFormatMessage)(
     DISP_MESSAGE *pmsg, char *msgtext, unsigned size,
     char *indent);

    DWORD (NMSSTDCALLPTR xxxSetTraceLevel)(
```

Argument	Description
	CTAHD ctahd, void *mgrcontext, unsigned svcid, unsigned tracemask); DWORD (NMSSTDCALLPTR xxxFormatTraceBuffer)(unsigned tracetag, void *inbuffer, unsigned insize, char *outbuffer, unsigned outsize); /****** for future use ******/ PFUNC (NMSSTDCALLPTR xxxGetFunctionPointer)(unsigned ordinal); } CTASVCMGR_FNS;
previnfo	Pointer to the service manager revision information structure containing the following fields: typedef struct { DWORD size; /* size of the returned structure */ DWORD majorrev; /* major revision of service/manager */ DWORD minorrev; /* minor revision of service/manager */ char builddate [12]; /* build date, "mmm dd yyyy\0" */ DWORD reqdisplevel; /* Required compat level of dispatcher */ DWORD expapilevel; /* Exported compat level of srvce API */ DWORD expspilevel; /* Exported compat level of srvce SPI */ } CTAINTREV_INFO ;

Return values

Return value	Description
SUCCESS	
CTAERR_ALREADY_DEFINED	Service manager that you are trying to register is already defined.
CTAERR_ALREADY_INITIALIZED	**ctaInitialize** was already called by the application.
CTAERR_BAD_ARGUMENT	Required pointer argument is NULL.
CTAERR_INCOMPATIBLE_REVISION	Service manager is incompatible with the revision of Natural Access that is being used.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory for the queue, the driver, or the context.

Details

dispRegisterServiceManager is called by a service manager at initialization time to register a service manager and its binding functions with the Natural Access dispatcher. Once this is done, the service manager is completely specified from the point of view of Natural Access.

The only exported entry that must be provided by the service manager DLL is **xxxInitializeManager** (for example, **vceInitializeManager** for the Voice Message service). **xxxInitializeManager** is called by **ctaInitialize** the first time the service manager is loaded into the process address space. After the service manager is registered with Natural Access, **xxxDefineService** is invoked on all the services managed by the service manager and is listed in the arguments to **ctaInitialize**.

Binding functions can be specified as NULL in the list provided to the dispatcher. In this case, the dispatcher substitutes its own default binding function for the missing service manager binding function. These default binding functions translate to no operations and allow the dispatcher to always call a binding function whether one is specified by the service manager or not.

dispRegisterWaitObject

Used by services to register wait objects and their respective processing functions with a Natural Access queue.

Prototype

DWORD **dispRegisterWaitObject** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ***waitobj***, CTA_WAITFN ***waitfn***, void ***arg***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.
waitobj	Pointer to wait object to register.
waitfn	Function to call when a wait object needs servicing. Typically this is **xxxFetchAndProcess** when handling events from the managed resource.
arg	Pointer to the argument to be passed to a wait function when signaled. Typically this is a ***queuecontext*** or ***mgrcontext***.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument has an invalid value, or a required pointer argument is NULL.
CTAERR_OUT_OF_RESOURCES	Unable to find free space in internal tables.
CTAERR_DUPLICATE_CTAWAITOBJ	Already registered wait object was passed as an argument to **ctaRegisterWaitObject**.
CTAERR_OUT_OF_MEMORY	Unable to allocate memory.

Details

dispRegisterWaitObject is used by services and service managers to register wait objects with the Natural Access queue. Wait objects must be handles to objects that can be waited upon using an operating system specific wait function (**poll** in Unix, **WaitForMultipleObjects** in Windows). They are waited upon either by **ctaWaitEvent** or by the application. Requests for servicing cause the dispatcher to invoke the wait function passed as an argument to **dispRegisterWaitObject**. **dispRegisterWaitObject** is invoked with ***waitobj*** and ***arg*** as arguments. If the function results in the generation of a Natural Access event, it should call **dispQueueEvent** to place the event in the dispatcher event queue or in the application event queue.

The wait object structure is operating system dependent. For UNIX it is a pollfd structure with the fd field identifying the muxable handle. For Windows, the wait object is the manual reset event object.

```
#ifdef UNIX
#include <stropts.h>
#include <poll.h>
typedef struct pollfd CTA_WAITOBJ;
```

The signature for **xxxFetchAndProcess** is:

DWORD **xxxFetchAndProcess** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ***waitobj***, void ***arg***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.
waitobj	Pointer to the newly created wait object.
arg	Pointer to a data structure used to help process the event. Typically this is a ***queuecontext*** or ***mgrcontext***.

See also

dispUnregisterWaitObject, **xxxAttachServiceManager**, **xxxOpenServiceManager**

dispRemoveRTC

Removes an RTC condition and corresponding action from the dispatcher's RTC database.

Prototype

DWORD **dispRemoveRTC** (CTAHD ***ctahd***, RTC_CONDITION ***pcondition***, RTC_ACTION ***paction***)

Argument	Description
ctahd	Handle to a Natural Access context.

pcondition	Pointer to an RTC_CONDITION structure describing an RTC condition to be removed:
	`typedef struct _RTC_CONDITION`
	`{`
	`  RTC_ATTRIBUTES Attribute; /*Characteristics of RTC condition*/`
	`  DWORD  EventID;    /* Service's event ID for condition    */`
	`  DWORD  EventValue; /* Specific value sent with the event. */`
	`                     /* Used with Attribute RTC_WITH_VALUE  */`
	`  char   Alias[ RTC_NAME_LEN ];`
	`                     /* ASCII name of RTC condition         */`
	`} RTC_CONDITION;`

paction	Pointer to an RTC_ACTION structure describing an action. Together with RTC_CONDITION, RTC_ACTION is used to explicitly specify the condition/action pair to be removed:
	`typedef struct _RTC_ACTION`
	`{`
	`  DWORD  Consumer;  /* Service ID of RTC consumer          */`
	`  DWORD  ActionCmd; /* Special command sent to service on  */`
	`                    /* recognition of condition            */`
	`} RTC_ACTION;`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Indicates any of the following: • pcondition or paction is NULL. • Consumer service is not opened on the context. • RTC condition EventId is for a service not attached to the context's CTA queue (see **ctaCreateQueue**). • Alias is a zero length string. • ActionCmd is not a Natural Access command value.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	Specified condition/action pair was not found in the Natural Access dispatcher's RTC database.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispRemoveRTC removes the RTC_CONDITION from the dispatcher's RTC database on the specified context. The dispatcher also invokes the provider service's binding function **xxxRemoveRTC**.

RTC_CONDITION and RTC_ACTION specify a command/action pair (or pairs) to remove from the database. The following table describes the fields in RTC_CONDITION:

Field	Description
Attribute	Set to one of the following to determine which condition/action pairs to remove: • RTC_EVT_WITH_ONE_VALUE: A specific pair where the event matches the EventID field and the event value matches the EventValue field. • RTC_EVT_WITH_ANY_VALUE: All pairs where the event matches the EventID field. The event can have any value. • RTC_NONE: All pairs where the event matches the EventID field or the alias matches the Alias field, and the action matches the ActionCmd field (in RTC_ACTION). The event can have any value. • RTC_ALL_PROVIDER_EVENTS: All pairs where the event is from a specific provider service. (The service is inferred from the event ID provided in the EventID field.) • RTC_ALL_CONSUMER_RTCS: All condition/action pairs relevant to the calling service.
EventID	Set as follows, depending upon the Attribute setting: • If Attribute is set to RTC_NONE and Alias is a zero-length string, set to the event ID. • If Attribute is set to RTC_EVT_WITH_ANY_VALUE, set to the event ID. • If Attribute is set to RTC_EVT_WITH_ONE_VALUE, set to the event ID. and specify the event value in the EventValue field. • If Attribute is set to RTC_ALL_PROVIDER_EVENTS, set to the event ID of any event from the provider service. The dispatcher infers the service name from the event. • If Attribute is set to RTC_ALL_CONSUMER_RTCS, EventID is ignored.
EventValue	If Attribute is set to RTC_EVT_WITH_ONE_VALUE, specifies the value to look for. Otherwise, EventValue is ignored.
Alias	Specifies a condition/action pair to remove, where the condition is an alias. To specify a pair this way, also set Attribute to RTC_NONE

The following table describes the fields in RTC_ACTION:

Field	Description
Consumer	Specifies the service ID of the consumer service.
ActionCmd	If Attribute is set to RTC_NONE and you are not specifying an alias (in the Alias field in RTC_CONDITION) set to the action command of the condition/action pair to be removed. If Attribute is set any other way, this field is ignored.

See also

dispAddRTC, **dispUnregisterRTCAlias**

dispSendCommand

Used by the service SPI to send commands to the API service implementation function.

Prototype

DWORD **dispSendCommand** (DISP_COMMAND ***ctacmd***)

Argument	Description

ctacmd	Pointer to command structure to send containing the following fields:
	`typedef struct`
	`{`
	`  DWORD id;  /* Command ID (and dest. service ID)  */`
	`  CTAHD ctahd;  /* Natural Accesso context handle  */`
	`  DWORD size1;  /* If dataptr1 !=NULL => size1 is buffer  */`
	`  /* size - else message specific data  */`
	`  void *dataptr1; /* Data pointer  */`
	`  DWORD size2;  /* If dataptr2 !=NULL => size2 is buffer  */`
	`  /* size - else message specific data  */`
	`  void *dataptr2; /* Data pointer  */`
	`  DWORD size3;  /* If dataptr3 !=NULL => size3 is buffer  */`
	`  /* size - else message specific data  */`
	`  void *dataptr3; /* Data pointer  */`
	`  unsigned objHd; /*Service object handle (formerly "reserved")*/`
	`  MESSAGE_ADDR addr;/* Source/Destination service ID  */`
	`} DISP_COMMAND;`
	`typedef struct`
	`{`
	`  WORD source  /* source service ID  */`
	`  WORD destination /* destination service ID  */`
	`} MESSAGE_ADDR`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***ctacmd*** is NULL.
CTAERR_INVALID_ADDRESS	Specified command address is invalid.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_LOCK_TIMEOUT	Thread lock timed out.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SERVICE_NOT_AVAILABLE	Specified service is not available.

Details

dispSendCommand can be used by the service SPI module to synchronously send commands to the associated API service implementation function. Upon return, the service SPI can dispose of buffers or structures as needed.

Note: You cannot invoke this command from within the service implementation of another service. Services must call other service's SPI functions.

For details on marshaling information in and out of the DISP_COMMAND structure, refer to *DISP_COMMAND structure* on page 61.

See also

xxxProcessCommand

dispSetAsyncIn

Designates a command buffer as being used for asynchronous input.

Prototype

DWORD NMSAPI **dispSetAsyncIn** (DWORD ***size***)

Argument	Description
size	Pointer to the buffer size field of the DISP_COMMAND structure.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***size*** is NULL.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispSetAsyncIn is used in service SPIs to control asynchronous inputs transferred with **dispSendCommand**. When Natural Access is running in client/server mode, you must specify that buffers are valid on the server side after service API calls return. The service SPI passes the buffer to the service manager with the function call. This buffer is used in the service manager after service API calls return. The service manager sends the execution results with the Natural Access event.

Note: Only one buffer in the DISP_COMMAND structure can include an asynchronous flag.

For more information about this function, refer to *DISP_COMMAND structure* on page 61.

See also

dispSetAsyncOut

dispSetAsyncOut

Designates a command buffer as being used for asynchronous output.

Prototype

DWORD NMSAPI **dispSetAsyncOut** (DWORD ***size**)

Argument	Description
size	Pointer to the buffer size field of the DISP_COMMAND structure.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***size*** is NULL.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispSetAsyncOut is used in service SPIs to control asynchronous output transferred with **dispSendCommand**. The service SPI passes the buffer to the service manager during a function call, and the service manager uses the buffer after the service API call returns. The service manager then sends the execution results with the Natural Access event.

When Natural Access is running in client/server mode, you must specify that buffers are valid on the server side after the service API calls return.

Note: Only one buffer in the DISP_COMMAND structure can include an asynchronous flag.

For more information about this function, refer to *DISP_COMMAND structure* on page 61.

See also

dispSetAsyncIn

dispSetHandleValue

Associates a value with a dispatcher-created handle.

Prototype

DWORD **dispSetHandleValue** (unsigned ***hd***, unsigned ***value***)

Argument	Description
hd	Handle returned by **dispCreateHandle**.
value	Value to associate with a given handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	Specified ***hd*** is not valid.

Details

dispSetHandleValue associates the specified ***value*** with the dispatcher-created handle. The value can be any value that fits into an unsigned integer (pointer or index). **dispSetHandleValue** is typically used for managing the mapping of API (client) side user handles to context handles and service (server) side handles. This is useful for services, such as the Voice Message service, that create another set of objects that are also associated with the context.

The value can be retrieved by using **dispGetHandleValue** and can also be set directly in the call to **dispCreateHandle**.

See also

dispAttachHandle, **dispDestroyHandle**, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValueEx**, **dispSetHandleValueEx**

dispSetHandleValueEx

Associates a service object name with a client-side API handle.

Prototype

DWORD NMSAPI **dispSetHandleValueEx** (unsigned ***hd***, unsigned ***value***, char ****name***)

Argument	Description
hd	Handle returned by **dispCreateHandle**.
value	Value to associate with the given handle.
name	Pointer to a name to associate with the object.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	Specified ***hd*** is not valid.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispSetHandleValueEx associates a ***name*** and server-side ***value*** with an object handle created by the dispatcher. ***value*** can be any value that fits into an unsigned integer (pointer or index). **dispSetHandleValueEx** is typically used to map API (client) side object handles to Natural Access context handles and service (server) side object handles. This is useful for services, such as the Voice Message service, that create multiple sets of objects associated with the Natural Access context. ***name*** is used for service object sharing.

The value can be retrieved by using **dispGetHandleValueEx** and can also be set directly in the call to **dispCreateHandle**.

See also

dispAttachHandle, **dispDestroyHandle**, **dispFindHandle**, **dispFindHandleByName**, **dispGetHandleValue**, **dispSetHandleValue**

dispSetParmByName

Permanently modifies the value of a default parameter.

Prototype

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

Argument	Description
ctahd	Handle to a Natural Access context.
parmname	Pointer to a parameter name or field.
buffer	Pointer to a buffer to receive the parameter value.
size	Size of ***buffer***.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Required pointer argument is NULL.
CTAERR_BAD_SIZE	***size*** is set to zero.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	Specified parameter does not exist or the owning service was not initialized.
CTAERR_NOT_INITIALIZED	Natural Access was not yet initialized.

Details

dispSetParmByName permanently modifies the default parameter values specified by ***parmname*** on the specified context. If the parameter is a single field, only that value is updated. If the field name is missing, this function updates the entire parameter structure with the defaults contained in ***buffer***. It is assumed that ***size*** matches exactly for either case.

Note: This function should be used very carefully since it modifies the per-context values without the knowledge of the application. It is typically the application's responsibility to maintain and modify parameter values.

See also

dispCheckParmByName, **dispGetParmByName**, **dispGetParms**

dispShutdown

Initiates a system or *ctdaemon* shutdown or restart

Prototype

DWORD **dispShutdown** (unsigned ***action***)

Argument	Description
action	How shutdown should be performed. Valid values for ***action*** are: CTA_SERVER_RESTART CTA_SERVER_SHUTDOWN CTA_SYSTEM_REBOOT CTA_SYSTEM_SHUTDOWN

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Invalid action specified.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

dispUnregisterRTCAlias

Removes an RTC condition event alias from the RTC database.

Prototype

DWORD **dispUnregisterRTCAlias** (CTAHD ***ctahd***, RTC_ALIAS ****alias***)

Argument	Description
ctahd	Handle to a Natural Access context.

alias	Pointer to the RTC_ALIAS array, defined as:
	`typedef struct`
	`{`
	`  DWORD EventID;  /* RTC condition event */`
	`  char  Alias[ RTC_NAME_LEN ]; /* ASCII name  */`
	`} RTC_ALIAS;`

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	***alias*** is NULL, ***alias*** contains a zero length Alias string, or the EventID is not a valid service event.
CTAERR_INVALID_CTAHD	Specified ***ctahd*** is invalid.
CTAERR_NOT_FOUND	Specified ***alias*** could not be found in the RTC database.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.
CTAERR_SVR_COMM	Natural Access Server is not running.

Details

dispUnregisterRTCAlias removes an alias from the RTC database. **dispUnregisterRTCAlias** should be called by the **xxxCloseService** binding function if a corresponding **dispRegisterRTCAlias** function was performed at the time of the **xxxOpenService** binding function call.

See also

dispRemoveRTC

dispUnregisterWaitObject

Used by services to remove the registration of wait objects and their respective processing functions with a Natural Access queue.

Prototype

DWORD **dispUnregisterWaitObject** (CTAQUEUEHD ***ctaqueuehd***, CTA_WAITOBJ ***waitobj***)

Argument	Description
ctaqueuehd	Handle returned by **ctaCreateQueue**.
waitobj	Pointer to the wait object to unregister.

Return values

Return value	Description
SUCCESS	
CTAERR_BAD_ARGUMENT	Function argument has an invalid value or a required pointer argument is NULL
CTAERR_INVALID_CTAQUEUEHD	Invalid ***ctaqueuehd*** was passed.
CTAERR_NOT_FOUND	Specified wait object does not exist.
CTAERR_NOT_INITIALIZED	Natural Access is not initialized.

Details

dispUnregisterWaitObject is used by services and service managers to remove the registration of wait objects with the Natural Access queue. It should be called from the detach/close binding function corresponding to the attach/open binding function from which the wait object was registered.

See also

dispRegisterWaitObject, **xxxCloseServiceManager**, **xxxDetachServiceManager**

15 Demonstration programs and utilities

Summary of the demonstration programs and utilities

The Natural Access Service Writer's Developer's Kit provides the following demonstration programs and utilities:

Program	Description
tiktst	Tests and demonstrates the tik service.
evttest	Tests and demonstrates the Event Tracking (EVT) service.
pf2src	Converts a *.pf* file to the corresponding C source and header files containing required Natural Access compliant parameter data structures.

tik service test program: tiktst

Name

tiktst

Purpose

Demonstrates the features of the tik service.

Usage

`tiktst` [***options***]

where ***options*** are:

Option	Specifies...
-m ***n***	Test mode. The supported test modes are: 1 = One Natural Access event queue; ten contexts registered with that queue. 2 = Two Natural Access events queues; one context registered per queue.
-t	Tracing on a per-context basis. Set the per-context tracemask to trace DRIVER_EVENTS, DRIVER_COMMANDS, and DEBUG_BIT_0.
-n ***numTicks***	Number of tick strings to request from the tik server. This value overrides the default TIK_START_PARM ***numTicks*** parameter.
-f ***frequency***	Frequency of tick strings from the tik server. This value overrides the default TIK_START_PARM ***frequency*** parameter.

Description

tiktst is a tik service demonstration program that uses the tik service to interact with the tik server. *tiktst* uses Natural Access and tik APIs to start a timer on the tik server and receive tick strings at a specified frequency. As each tick string is received, it is printed to *stdout*.

Build procedure

The tik service, tik server, and test program are shipped as source only. To build the components:

1. Build the tik server by entering the following commands:

Operating system	Commands
Windows	`cd nms\ctaccess\demos\tiksvr` `nmake`
UNIX	`cd /nms/ctaccess/demos/tiksvr` `make`

2. Build the tik service by entering the following commands:

Operating system	Commands
Windows	`cd nms\ctaccess\demos\tiksvc` `nmake`
UNIX	`cd /nms/ctaccess/demos/tiksvc` `make`

3. Build the tik test program by entering the following commands:

Operating system	Commands
Windows	`cd nms\ctaccess\demos\tiktst` `nmake`
UNIX	`cd /nms/ctaccess/demos/tiktst` `make`

Test procedure

1. Ensure that the tik server, the tik service, and the tik test program are built.
2. Start the tik server by entering:

```
tiksvr
```

3. Ensure that *ctdaemon* is running and was initialized with the TIK,TIMGR service/service manager pair.
4. Start *tiktst* by entering the following at the prompt:

```
tiktst [-m n] [-n numTicks] [-f frequency]
```

EVT service test program: evttest

Name

evttest

Purpose

Sample event tracking application for testing and demonstrating the Event Tracking (EVT) service. *evttest* uses the tik service (TIK) as the runtime control (RTC) event provider and the Event Tracking service (EVT) as the RTC event consumer to demonstrate the features of RTC.

Usage

`evttest` [***options***]

where ***options*** are:

Option	Specifies...
-m ***n***	Test scenario. The supported test scenarios are: 1 = One Natural Access event queue; ten contexts registered with that queue, all tik event codes added to event-tracking list. 2 = Two Natural Access events queues; one context registered per queue, specific tik event codes added to event-tracking list. 3 = One Natural Access events queue; one context registered on the queue, all tik event codes added to event-tracking list. For more information about these scenarios, see *evttest scenarios* on page 241.
-t	Tracing on a per-context basis. Set the per-context tracemask to trace DRIVER_EVENTS, DRIVER_COMMANDS, and DEBUG_BIT_0.
-n ***numTicks***	Number of tick strings to request from the tik server. This value overrides the default TIK_START_PARM ***numTicks*** parameter.
-f ***frequency***	Frequency of tick strings from the tik server. This value overrides the default TIK_START_PARM ***frequency*** parameter.
-l ***logfile***	Path and name of the event log file. If -l is not specified, the standard output file is used.

Description

The Event Tracking (EVT) service is a sample service shipped with the Natural Access Service Writer's Developer's Kit. EVT does not manage any particular resource of its own. It internally uses Natural Access runtime control (RTC) features to log events generated by other services opened under the same context as EVT. No RTC symbol or type definitions are exposed to the user application through EVT API specifications. (RTC is intended for inter-service operations only.)

A user application can specify event codes to track from a service opened under the same context as EVT. These event codes are specified on a per context basis, both for tracking and logging. For each context, the user application can track:

- All event codes from a certain service
- A particular event code with a particular event value from a certain service
- A particular event code with any event value from a certain service

A user application can remove event codes from the existing tracking list, on a per context basis.

A user application can also turn on and off event logging on a per context basis, and specify the event log file for the whole application.

evttest scenarios

This section describes the three test scenarios that *evttest* can run. The test scenario to run is specified with the `-m` command line option.

Scenario 1

1. One queue is created.
2. Ten contexts are created under the queue.
3. One instance of TIK and one instance of EVT are opened under each context.
4. The EVT log file is set for the whole application, based on the filename specified with the `-l` option.
5. Event logging is turned on.
6. All TIK event codes are added to the event-tracking list.
7. The tick timer is started for each instance of TIK.
8. A TIKEVN_TIMER_TICK event is issued per tick until the maximum tick count specified with the `-n` option is reached. If no maximum tick count is specified, it defaults to 2 ticks.
9. When the maximum tick count is reached, a TIKEVN_TIMER_DONE event with event value (reason code) CTA_REASON_FINISHED is issued for each tick timer.
10. The same event and reason code is issued when the tick timer is stopped.
11. The event codes are removed from the event tracking list.
12. Event logging is turned off.
13. For each context, the instances of TIK and EVT are closed.
14. Each context is destroyed.
15. Each queue is destroyed.

Scenario 2

1. Two queues are created.
2. One context is created under each queue.
3. One instance of TIK and one instance of EVT are opened under each context.
4. The EVT log file is set for the whole application, based on the filename specified with the `-l` option.
5. Event logging is turned on.
6. For the context under the first queue, the event code TIKEVN_TIMER_DONE with event value CTA_REASON_FINISHED is added to the event tracking list.
7. For the context under the second queue, the event code TIKEVN_TIMER_DONE (no specific event value) is added the event tracking list.
8. The tick timer is started for each instance of TIK.

9. A TIKEVN_TIMER_TICK event is issued per tick until the maximum tick count specified with the -n option is reached. If no maximum tick count is specified, it defaults to 2 ticks.
10. When the maximum tick count is reached, a TIKEVN_TIMER_DONE event with event value (reason code) CTA_REASON_FINISHED is issued for each tick timer.
11. The same event and reason code is issued when the tick timer is stopped.
12. The event codes are removed from the event tracking list.
13. Event logging is turned off.
14. For each context, the instances of TIK and EVT are closed.
15. Each context is destroyed.
16. Each queue is destroyed.

Scenario 3

1. One queue is created.
2. One context is created under the queue.
3. One instance of TIK and one instance of EVT are opened under each context.
4. The EVT log file is set for the whole application, based on the filename specified with the -l option.
5. Event logging is turned on.
6. Add all TIK event codes to the event-tracking list.
7. The tick timer is started for each instance of TIK.
8. A TIKEVN_TIMER_TICK event is issued per tick until the maximum tick count specified with the -n option is reached. If no maximum tick count is specified, it defaults to 2 ticks.
9. When the maximum tick count is reached, a TIKEVN_TIMER_DONE event with event value (reason code) CTA_REASON_FINISHED is issued for each tick timer.
10. The same event and reason code is issued when the tick timer is stopped.
11. The event codes are removed from the event tracking list.
12. Event logging is turned off.
13. For each context, the instances of TIK and EVT are closed.
14. Each context is destroyed.
15. Each queue is destroyed.

Build procedure

The Event Tracking service and its demonstration program are provided in source only. To build the components:

1. Build the tik server and the tik service as described in *tik service test program: tiktst* on page 238.
2. Build the EVT service by entering the following commands:

Operating system	Commands
Windows	`cd nms\ctaccess\demos\evtsvc` `nmake`
UNIX	`cd /nms/ctaccess/demos/evtsvc` `make`

3. Build the EVT test program by entering the following commands:

Operating system	Commands
Windows	`cd nms\ctaccess\demos\evttest` `nmake`
UNIX	`cd /nms/ctaccess/demos/evttest` `make`

Test procedure

1. Ensure that the tik server and the tik service are built.
2. Ensure that the EVT service and the EVT test program are built.
3. Configure *cta.cfg* to include the new services and service managers. To do so, add the following lines in the `[ctasys]` section of the file:

```
Service = tik, tikmgr
Service = evt, evtmgr
```

4. Start the tik server:

```
tiksvr
```

Note: To enable tracing, also start the Natural Access Server in interactive mode. To do so, enter: `ctdaemon -i`. If the Natural Access Server is already running as a service, access it in console mode by entering `ctdaemon -c`.

5. Run the first EVT test scenario:

```
evttest -m1 -l evt1.log
```

6. Run the second EVT test scenario:

```
evttest -m2 -l evt2.log
```

7. Run the third EVT test scenario:

```
evttest -m3 -l evt3.log
```

Parameter file generation utility - pf2src

Name

pf2src

Purpose

Converts a *.pf* file to the corresponding C source and header files containing required Natural Access compliant parameter data structures.

Usage

`pf2src` [***options***] ***ParameterFileName.pf***

where ***options*** are:

Option	Specifies...
-w	Not to overwrite existing source and header files.
-x	An extension parameter table should be generated (rather than a standard parameter table).
-s ***filename***	File name for both the generated C source file and header file. The default is to use the same file name as the *.pf* file.

Description

pf2src converts a *.pf* parameter description file to the corresponding C source and header files. These files contain data structures needed by the call to **dispRegisterService**. The data structures must be registered as either a standard parameter table or an extension parameter table.

For a standard parameter table, the generated C label is _xxxParmDescTable.

For an extension parameter table, the generated C label is _xxxExtParmDescTable.

Procedure

1. Create a *.pf* file (for example, *tikparm.pf*). Refer to the *Natural Access Developer's Reference Manual*.
2. Run *pf2src* to create the associated source and header file. For example:

```
pf2src tikparm.pf
```

This example creates the files *tikparm.c* and *tikparm.h*.

In the call to **dispRegisterService** in **xxxDefineService**, use the appropriate parameter table argument. For example, in **tikDefineService**, the call to **dispRegisterService** contains a _tikParmDescTable as the standard parameter table argument.

Note: Typically, the call to *pf2src* is embedded in the makefile for your service. Refer to the makefile for the tik service.

16 Errors and events

Error codes

The Natural Access error codes are presented in two tables:

- Alphabetically by error code name, including a description of the problem and a possible solution.
- Numerically by hex number, decimal number, and error name.

All Natural Access functions return SUCCESS (0) or an error code indicating that the function failed and the reason for the failure. Natural Access error codes are defined in the *ctaerr.h* include file. The error codes are prefixed with CTAERR.

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 only applicable to DONE 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.

Error name	Hex	Decimal	Description
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.
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.

Error name	Hex	Decimal	Description
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.
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.

Error name	Hex	Decimal	Description
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.
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.

Error name	Hex	Decimal	Description
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.
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.

Error name	Hex	Decimal	Description
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.
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.

Error name	Hex	Decimal	Description
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.
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

Hex	Decimal	Error name
0x18	24	CTAERR_DRIVER_ERROR
0x19	25	CTAERR_TOO_MANY_OPEN_FILES
0x1A	26	CTAERR_INVALID_BOARD
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

Hex	Decimal	Error name
0x10000001	268435457	CTAERR_INCOMPATIBLE_PARMS
0x10000002	268435458	CTAERR_DUPLICATE_EXTPARMS
0x10000003	268435459	CTAERR_WAIT_PENDING
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

Events

Natural Access events are defined in the *ctadef.h* include file. Events are prefixed with CTAEVN_. The following table alphabetically lists the Natural Access events:

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.

17 Buffer management summary

Buffer passing rules

When passing buffers, adhere to the following rules:

- A buffer pointer passed in the DISP_COMMAND structure cannot be used as both an input buffer and as an output buffer.
- A DISP_COMMAND structure can contain at most one asynchronous buffer pointer.
- A service must always inform Natural Access of how a buffer pointer is being used by setting flags in the size field related to the buffer pointer.

DISP_COMMAND buffer flags

The following table describes which flag needs to be set in the size field in the DISP_COMMAND structure based upon how the buffer pointer is being used:

Buffer pointer used as...	Usage mode of buffer is...	Flag set in size# field is...
Input	Synchronous	none (default)
Input	Asynchronous	CTA_ASYNC_BUF_IN
Output	Synchronous	CTA_VIRTUAL_BUF
Output	Asynchronous	CTA_ASYNC_BUF_OUT

Asynchronous buffers dynamically allocated by either the service interface or by the application cannot be freed until the service manager generates an associated DONE event.

The Natural Access dispatcher provides two functions for setting the asynchronous command buffer flags: **dispSetAsyncIn** and **dispSetAsyncOut**. The Natural Access dispatcher clears these buffer flags prior to delivering a DISP_COMMAND structure to a service manager.

DISP_EVENT buffer flags

The following table describes which flag needs to be set in the size field in the DISP_EVENT structure based upon the buffer owner, how the buffer is being used, and how the buffer is being allocated:

Buffer owner	Used as asynchronous...	Method of allocation	Flag set in size field
Client-side	Input	Static or dynamic	CTA_EVT_ASYNC_BUF_IN
Client-side	Output	Static or dynamic	None (default)
Server-side	Input	Static	CTA_INTERNAL_BUFFER_STATIC
Server-side	Input	Dynamic	CTA_INTERNAL_BUFFER

To allocate and free buffers, you can use two Natural Access dispatcher functions: **dispAllocBuffer** and **dispFreeBuffer**. It is strongly recommended that the service implementation use **dispAllocBuffer** to allocate non-static buffers for delivery in event messages. Use of this function relieves the service implementation from having to free the buffer (Natural Access or the application performs the necessary **dispFreeBuffer** invocation). This method avoids complications since there is no way for the service implementation to determine when the dispatcher has completed delivery of an event.

Index

A

aliases 142

alphabetical error summary 245

API 19

  definition 40

  header files creation 49

  implementation overview 49

  parameter declaration 55

  writing the API function code 52

arguments coding conventions 33

B

binary files 26

binding functions 65, 69, 78, 98

buffer arguments 255, 255, 256

buffers 24

built-in dispatcher services 21

C

client 15

  binding functions 65

client/server model 15

client-side object handle 123

coding conventions 32

command processing 22

communications 21

context handle 40

CTAAPIERROR macro 53

ctaDestroyQueue 96

ctadisp.h 150

ctaInitialize 76

ctaOpenServices 89

D

definition phase 31, 40

DISP_COMMAND 61, 255

DISP_EVENT 114, 256

dispAddRTC 182

dispAllocBuffer 185

dispApiError 186

dispatcher 20, 21

dispatcher service functions 177

  application initialization 199

  buffer management 185, 197, 229, 230

  error and tracing log 186, 192, 210, 211

  event queue processing 207, 213, 215

  handle management 187, 189, 191, 194, 195, 201, 202, 214, 231, 232

  parameter modification 188, 205, 206, 233

  runtime control 182, 198, 203, 204, 208, 217, 225, 235

  server command 234

  service command queuing and sending 227

  service information 209

  service manager and service registration 218, 220

  wait object 196, 223, 236

dispAttachHandle 187

dispCheckParmByName 188

dispCreateHandle 189

dispDestroyHandle 191

dispError 192

dispFindHandle 194

dispFindHandleByName 195

dispFindWaitObject 196

dispFreeBuffer 197

dispGetActiveRTC 198

dispGetAppCtl 199

dispGetHandleValue 201
dispGetHandleValueEx 202
dispGetNumRTC 203
dispGetNumRTCAliases 204
dispGetParmByName 205
dispGetParms 206
dispGetQueueHandle 207
dispGetRTCAliases 208
dispGetText 209
dispGetTracePointer 210
dispLogTrace 211
dispMakeAndQueueEvent 213
dispQueueCommand 214
dispQueueEvent 215
dispRegisterRTCAlias 217
dispRegisterService 218
dispRegisterServiceManager 220
dispRegisterWaitObject 223
dispRemoveRTC 225
dispSendCommand 227
dispSetAsyncIn 229
dispSetAsyncOut 230
dispSetHandleValue 231
dispSetHandleValueEx 232
dispSetParmByName 233
dispShutdown 234
dispUnregisterRTCAlias 235
dispUnregisterWaitObject 236
distributed computing 15
DLLs 26

E

enumerations coding conventions 36
errors 245
  coding conventions 35
  definition 43
  handling 23
  macros 53
events 254
  coding conventions 34
  defining 41
  handling 22, 85, 96
  managed resources 114
EVT service test program 240

F

function arguments coding conventions 33
function names coding conventions 33
functions 153, 154, 177, 177, 178, 178, 178, 179, 179, 179, 179, 180, 180, 180

H

header files 49

I

implementation phase 31, 49, 59
in arguments 33

L

libraries 26
life cycle binding functions 69

M

managed resources 18, 37, 85
manifest constants coding conventions 36

N

Natural Access architecture 18, 25
Natural Access service 26
numerical error summary 251

O

out arguments 33

P

parameter file generation utility 244
parameters 44
  coding conventions 36
  declaration 55
  management 23
pf2src utility 244

R

reason codes conventions 35

reason names coding conventions 34, 41
remote procedure call (RPC) 16
return values coding conventions 34
RPC (remote procedure call) 16
RTC (runtime control) 135
  base services RTC aliases 152
  reading the database 149
  registering for notification 147
  sample code 150
  using aliases 142
runtime binding functions 98
runtime control 135
  base services RTC aliases 152
  reading the database 149
  registering for notification 147
  sample code 150
  using aliases 142

S

server 15
server-side object handle 123
service 16
  API 19
  functions 110
  handles 123
  ID 26, 40
  implementation 109
  name 26
  object 37, 40, 121, 129
  registration 76
  setup 22
  shutdown 95
  SPI 19
  starting 89
  writing 31
service manager 20
service manager binding functions 153
  life cycle - event handling 158, 163
  life cycle - registration 162, 167
  life cycle - runtime control 156, 174
  life cycle - service startup and shutdown 159, 161, 168, 170
  runtime - command processing 171
  runtime - error handling 164, 166
  runtime - event processing 173
  runtime - trace handling 165, 175
service test program 238
software structure 26
source files 27
SPI 19
  function 52
  implementation 59, 60
  SPI function prototypes 59
structures coding conventions 36
stubs 16

T

template source files 29
TIK sample service 29
  API definition 40
  API function example 54
  error definition 43
  event handling shutdown 96
  events and reasons definition 41
  example overview 29
  example SPI function 65
  header files creation 49
  managed resource 37
  managed resource event handling 85
  parameter declaration 55
  parameter definition 44, 55
  registering services 76
  service function implementation 110
  service ID 40
  service shut down 95
  service starting 89
  SPI function prototypes 59

tracing definition 45
tracing records 119
tikst service test program 238
trace records 119
trace tags 46
tracemask 45
tracing 24, 45
V
Voice Message (VCE) service 121
W
writing a service 31
X
xxxAddRTC 156
xxxapi.c file 52
xxxAttachServiceManager 158
xxxCloseService 159
xxxCloseServiceManager 161
xxxcmd.c 109
xxxDefineService 162
xxxDetachServiceManager 163
xxxFormatMessage 164
xxxFormatTraceBuffer 165
xxxGetText 166
xxxInitializeManager 167
xxxOpenService 168
xxxOpenServiceManager 170
xxxparm.c 55
xxxparm.h 55
xxxparm.pf 55
xxxProcessCommand 171
xxxProcessEvent 173
xxxRemoveRTC 174
xxxSetTraceLevel 175
xxxspi.c 60
xxxspi.h 59
xxxsys.h 109