DCE 1.2.2 Application Development Reference

OSF® DCE Product Documentation

The Open Group

OSF® DCE Product Documentation:

DCE 1.2.2 Application Development Reference, (Volume 1)
ISBN 1–85912–103–9
Document Number F205A

DCE 1.2.2 Application Development Reference, (Volume 2)
ISBN 1–85912–108–X
Document Number F205B

DCE 1.2.2 Application Development Reference, (Volume 3)
ISBN 1–85912–159–4
Document Number F205C

Published in the U.K. by The Open Group, 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:
OGPubs@opengroup.org

OTHER NOTICES

Contents

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the IT DialTone. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the "X" mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of IT DialTone to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product

Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The “X” mark is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

CAE Specifications

CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

Preliminary Specifications

Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as

stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

Preliminary Specifications are analogous to the trial-use standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

Consortium and Technology Specifications

The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

Product Documentation

This includes product documentation—programmer's guides, user manuals, and so on—relating to the Prestructured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

Guides

These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

Technical Studies

Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They

are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all live documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

This Book

The *DCE 1.2.2 Application Development Reference* provides complete and detailed reference information to help application programmers use the correct syntax for Distributed Computing Environment (DCE) calls when writing UNIX applications for a distributed computing environment.

Audience

This document is written for application programmers who want to write Distributed Computing Environment applications for a UNIX environment.

Applicability

This document applies to the OSF ® DCE Version 1.2.2 offering and related updates. See your software license for details.

Purpose

The purpose of this document is to assist application programmers when writing UNIX applications for a distributed computing environment. After reading this manual, application programmers should be able to use the correct syntax for DCE calls.

Document Usage

This document consists of six chapters and is organized into three volumes.

- Volume 1 (Document Number 205A, ISBN 1–85912–103–9) includes:
  - — DCE Routines (Chapter 1)

  - — DCE Threads (Chapter 2)
  - — DCE Remote Procedure Call (beginning of Chapter 3)
- Volume 2 (Document Number 205B, ISBN 1–85912–108–X) includes:
  - — DCE Remote Procedure Call (Chapter 3, continued)
  - — DCE Directory Service (Chapter 4)
  - — DCE Distributed Time Service (Chapter 5)
  - — DCE Security Service (beginning of Chapter 6)
- Volume 3 (Document Number 205C, ISBN 1–85912–159–4) includes:
  - — DCE Security Service (Chapter 6, continued)

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *DCE 1.2.2 Introduction to OSF DCE*
  Document Number F201, ISBN 1–85912–182–9
- *DCE 1.2.2 Command Reference*
  Document Number F212, ISBN 1–85912–138–1
- *DCE 1.2.2 Application Development—Introduction and Style Guide*
  Document Number F202, ISBN 1–85912– 187–X
- *DCE 1.2.2 Application Development Guide—Core Components*
  Document Number F203A, ISBN 1–85912–192–6 (Volume 1)
  Document Number F203B, ISBN 1–85912–154–3 (Volume 2)
- *DCE 1.2.2 Application Development Guide—Directory Services*
  Document Number F204, ISBN 1–85912–197–7
- *DCE 1.2.2 Administration Guide—Introduction*
  Document Number F207, ISBN 1–85912–113–6

- *DCE 1.2.2 Administration Guide—Core Components*
  Document Number F208, ISBN 1–85912–118–7
- *DCE 1.2.2 DFS Administration Guide and Reference*
  Document Number F209A, ISBN 1–85912–123–3 (Volume 1)
  Document Number F209B, ISBN 1–85912–128–4 (Volume 2)
- *DCE 1.2.2 GDS Administration Guide and Reference*
  Document Number F211, ISBN 1–85912–133–0
- *DCE 1.2.2 File-Access Administration Guide and Reference*
  Document Number F216, ISBN 1–85912–158–6
- *DCE 1.2.2 File-Access User's Guide*
  Document Number F217, ISBN 1–85912–163–3
- *DCE 1.2.2 Problem Determination Guide*
  Document Number F213A, ISBN 1–85912–143–8 (Volume 1)
  Document Number F213B, ISBN 1–85912–148–9 (Volume 2)
- *DCE 1.2.2 Testing Guide*
  Document Number F215, ISBN 1–85912–153–5
- *DCE 1.2.2 File-Access FVT User's Guide*
  Document Number F210, ISBN 1–85912–189–6
- *DCE 1.2.2 Release Notes*
  Document Number F218, ISBN 1–85912–168–3

Typographic and Keying Conventions

This guide uses the following typographic conventions:

Bold — **Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.

Italic — *Italic* words or characters represent variable values that you must supply. *Italic* type is also used to introduce a new DCE term.

`Constant width` — Examples and information that the system displays appear in `constant width` typeface.

[ ] — Brackets enclose optional items in format and syntax descriptions.

{ } Braces enclose a list from which y ou must choose an item in format and syntax descriptions.

| A vertical bar separates items in a list of choices.

< > Angle brackets enclose the name of a key on the keyboard.

... Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

This guide uses the following keying conventions:

< Ctrl-*x* **>** or ^ *x*
The notation **< Ctrl-** *x* **>** or ^ *x* followed by the name of a key indicates a control character sequence. For example, **< Ctrl-C>** means that you hold down the control key while pressing **< C>**.

< Return> The notation **< Return>** refers to the key on your terminal or workstation that is labeled with the word Return or Enter, or with a left arrow.

Pathnames of Directories and Files in DCE Documentation

For a list of the pathnames for directories and files referred to in this guide, see the *DCE 1.2.2 Administration Guide—Introduction* and *DCE 1.2.2 Testing Guide*.

Problem Reporting

If you have any problems with the software or vendor-supplied documentation, contact your software vendor’s customer service department. Comments relating to this Open Group document, however, should be sent to the addresses provided on the copyright page.

Trademarks

Motif®, OSF/1®, and UNIX® are registered trademarks and the IT DialTone™, The Open Group™, and the "X Device"™ are trademarks of The Open Group.

DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation.

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.

Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.

AFS, Episode, and Transarc are registered trademarks of the Transarc Corporation.

DFS is a trademark of the Transarc Corporation.

Episode is a registered trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

AIX and RISC System/6000 are registered trademarks of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG.

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

NFS, Network File System, SunOS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

PostScript is a trademark of Adobe Systems Incorporated.

Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corp.

NetWare is a registered trademark of Novell, Inc.

rpc_mgmt_ep_elt_inq_begin

Purpose

Creates an inquiry context for viewing the elements in an endpoint map; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_begin(
        rpc_binding_handle_t ep_binding,
        unsigned32 inquiry_type,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        uuid_t *object_uuid,
        rpc_ep_inq_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input

ep_binding Specifies the host whose local endpoint map elements you receive. To receive elements from the same host as the calling application, specify NULL.

To receive local endpoint map elements from another host, specify a server binding handle for that host. You can specify the same binding handle you are using to make other remote procedure calls. The object UUID associated with this parameter must be a nil UUID. If you specify a nonnil UUID, the routine fails with the status code **ept_s_cant_perform_op**. Other than the host information and object UUID, all information in this parameter is ignored.

inquiry_type Specifies an integer value that indicates the type of inquiry to perform on the local endpoint map. The following table shows the valid inquiry types:

Valid Inquiries on Local Endpoint Maps	
Value	**Description**
rpc_c_ep_all_elts	Returns every element from the local endpoint map. The *if_id*, *vers_option*, and *object_uuid* parameters are ignored.
rpc_c_ep_match_by_if	Searches the local endpoint map for those elements that contain the interface identifier specified by the *if_id* and *vers_option* values. The *object_uuid* parameter is ignored.
rpc_c_ep_match_by_obj	Searches the local endpoint map for those elements that contain the object UUID specified by the *object_uuid* parameter. The *if_id* and *vers_option* parameters are ignored.
rpc_c_ep_match_by_both	Searches the local endpoint map for those elements that contain the interface identifier and object UUID specified by the *if_id*, *vers_option*, and *object_uuid* parameters.

if_id Specifies the interface identifier of the local endpoint map elements to be returned by the **rpc_mgmt_ep_elt_inq_next()** routine.

Use this parameter only when specifying a value of **rpc_c_ep_match_by_if** or **rpc_c_ep_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and the value NULL can be specified.

vers_option Specifies how the **rpc_mgmt_ep_elt_inq_next()** routine uses the *if_id* parameter. Use this parameter only when specifying a value of **rpc_c_ep_match_by_if** or **rpc_c_ep_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and a 0 (zero) value can be specified.

The following table presents the valid values for this parameter:

Valid values of vers_option	
Value	**Description**
rpc_c_vers_all	Returns local endpoint map elements that offer the specified interface UUID, regardless of the version numbers. For this value, specify 0 (zero) for both the major and minor versions in *if_id*.
rpc_c_vers_compatible	Returns local endpoint map elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
rpc_c_vers_exact	Returns local endpoint map elements that offer the specified version of the specified interface UUID.
rpc_c_vers_major_only	Returns local endpoint map elements that offer the same major version of the specified interface UUID (ignores the minor version). For this value, specify 0 (zero) for the minor version in *if_id*.
rpc_c_vers_upto	Returns local endpoint map elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, suppose *if_id* contains V2.0 and the local endpoint map contained elements with the following versions: V1.3, V2.0, and V2.1. The **rpc_mgmt_ep_elt_inq_next()** routine returns the elements with V1.3 and V2.0.)

object_uuid Specifies the object UUID that **rpc_mgmt_ep_elt_inq_next()** looks for in local endpoint map elements.

This parameter is used only when you specify a value of **rpc_c_ep_match_by_obj** or **rpc_c_ep_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you can supply NULL to specify a nil UUID.

Output

inquiry_context
: Returns an inquiry context for use with the **rpc_mgmt_ep_elt_inq_next()** and **rpc_mgmt_ep_elt_inq_done()** routines.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_ep_elt_inq_begin()** routine creates an inquiry context for viewing server address information stored in the local endpoint map.

Using the *inquiry_type* and *vers_option* parameters, an application specifies which of the following local endpoint map elements are returned from calls to the **rpc_mgmt_ep_elt_inq_next()** routine:

- All elements.
- Those elements with the specified interface identifier.
- Those elements with the specified object UUID.
- Those elements with both the specified interface identifier and object UUID.

Before calling the **rpc_mgmt_ep_elt_inq_next()** routine, the application must first call this routine to create an inquiry context.

After viewing the local endpoint map elements, the application calls the **rpc_mgmt_ep_elt_inq_done()** routine to delete the inquiry context.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_inquiry_context
Invalid inquiry context.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ep_unregister(3rpc)**, **rpc_mgmt_ep_elt_inq_done(3rpc)**, **rpc_mgmt_ep_elt_inq_next(3rpc)**, **rpc_mgmt_ep_unregister(3rpc)**.

rpc_mgmt_ep_elt_inq_done

Purpose

Deletes the inquiry context for viewing the elements in an endpoint map; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_done(
        rpc_ep_inq_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input/Output

inquiry_context

Specifies the inquiry context to delete. (An inquiry context is created by calling **rpc_mgmt_ep_elt_inq_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_ep_elt_inq_done()** routine deletes an inquiry context. The **rpc_mgmt_ep_elt_inq_begin()** routine created the inquiry context.

An application calls this routine after viewing local endpoint map elements using the **rpc_mgmt_ep_elt_inq_next()** routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_inquiry_context
Invalid inquiry context.

Related Information

Functions: **rpc_mgmt_ep_elt_inq_begin(3rpc)**, **rpc_mgmt_ep_elt_inq_next(3rpc)**.

rpc_mgmt_ep_elt_inq_next

Purpose

Returns one element from an endpoint map; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_ep_elt_inq_next(
        rpc_ep_inq_handle_t inquiry_context,
        rpc_if_id_t *if_id,
        rpc_binding_handle_t *binding,
        uuid_t *object_uuid,
        unsigned_char_t **annotation,
        unsigned32 *status);
```

Parameters

Input

inquiry_context
: Specifies an inquiry context. This inquiry context is returned from the **rpc_mgmt_ep_elt_inq_begin()** routine.

Output

if_id
: Returns the interface identifier of the local endpoint map element.

binding
: Returns the binding handle from the local endpoint map element.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call the **rpc_binding_free()** routine.

object_uuid
: Returns the object UUID from the local endpoint map element.

  Specify NULL to prevent the routine from returning this parameter.

annotation
: Returns the annotation string for the local endpoint map element. If there is no annotation string in the local endpoint map element, the string **\0** is returned.

Specify NULL to prevent the routine from returning this argument. In this case the application does not call the **rpc_string_free()** routine.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_ep_elt_inq_next()** routine returns one element from the local endpoint map. Regardless of the selector value specified for the *inquiry_type* parameter in **rpc_mgmt_ep_elt_inq_begin()**, this routine returns all the components of a selected local endpoint map element. The **rpc_ep_register()** routine's reference page summarizes the contents of an element in the local endpoint map.

An application can view all the selected local endpoint map elements by repeatedly calling the **rpc_mgmt_ep_elt_inq_next()** routine. When all the elements have been viewed, this routine returns an **rpc_s_no_more_elements** status. The returned elements are unordered.

If a remote endpoint map contains elements that include a protocol sequence that your system does not support, this routine does not return the elements. (A protocol sequence is part of the binding information component of an endpoint map element.) To receive all possible elements from a remote endpoint map, your application must run on a system that supports the protocol sequences included in the elements.

For example, if your system does not support protocol sequence **ncacn_ip_tcp** and a remote endpoint map contains elements that include this protocol sequence, this routine does not return these elements to your application. If your application ran on a system that supported protocol sequence **ncacn_ip_tcp**, this routine would return the elements.

The RPC runtime allocates memory for the returned *binding* and the *annotation* string on each call to this routine. The application calls the **rpc_binding_free()** routine for each returned *binding* and the **rpc_string_free()** routine for each returned *annotation* string.

After viewing the local endpoint map's elements, the application must call the **rpc_mgmt_ep_elt_inq_done()** routine to delete the inquiry context.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_comm_failure
Communications failure.

ept_s_database_invalid
Endpoint map database invalid.

rpc_s_fault_context_mismatch
Fault context mismatch.

ept_s_invalid_context
Invalid inquiry type for this context.

ept_s_invalid_entry
Invalid database entry.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_inquiry_context
Invalid inquiry context.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_no_more_elements
No more elements.

Related Information

Functions: **rpc_binding_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_mgmt_ep_elt_begin(3rpc)**, **rpc_mgmt_ep_elt_done(3rpc)**, **rpc_string_free(3rpc)**.

rpc_mgmt_ep_unregister

Purpose

Removes server address information from an endpoint map; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_ep_unregister(
        rpc_binding_handle_t ep_binding,
        rpc_if_id_t *if_id,
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

Parameters

Input

ep_binding Specifies the host whose local endpoint map elements you unregister (that is, remove). To remove elements from the same host as the calling application, specify NULL.

To remove local endpoint map elements from another host, specify a server binding handle for that host. You can specify the same binding handle you are using to make other remote procedure calls. The object UUID associated with this parameter must be a nil UUID. If you specify a nonnil UUID, the routine fails with the status code **ept_s_cant_perform_op**. Other than the host information and object UUID, all information in this parameter is ignored.

if_id Specifies the interface identifier to remove from the local endpoint map.

binding Specifies the binding handle to remove.

object_uuid
Specifies an optional object UUID to remove.

The value NULL indicates there is no object UUID to consider in the removal.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_ep_unregister()** routine unregisters (that is, removes) an element from a local endpoint map. A management program calls this routine to remove addresses of servers that are no longer available, or to remove addresses of servers that support objects that are no longer offered.

Use this routine cautiously; removing elements from the local endpoint map may make servers unavailable to client applications that do not already have a fully bound binding handle to the server.

A management application calls the **rpc_mgmt_ep_inq_next()** routine to view local endpoint map elements. The application can then remove the elements using the **rpc_mgmt_ep_unregister()** routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

ept_s_cant_access
Error reading the endpoint database.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_comm_failure
Communications failure.

ept_s_database_invalid
Endpoint map database is invalid.

ept_s_invalid_entry
Invalid database entry.

ept_s_not_registered
No entries found.

ept_s_update_failed
Update failed.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_no_interfaces
No interfaces registered.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_mgmt_ep_elt_inq_begin(3rpc)**, **rpc_mgmt_ep_elt_inq_done(3rpc)**, **rpc_mgmt_ep_elt_inq_next(3rpc)**, **rpc_ns_binding_unexport(3rpc)**.

rpc_mgmt_inq_com_timeout

Purpose

Returns the communications timeout value in a binding handle; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_inq_com_timeout(
        rpc_binding_handle_t binding,
        unsigned32 *timeout,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a server binding handle.

Output

timeout Returns the communications timeout value from the *binding* parameter.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_inq_com_timeout()** routine returns the communications timeout value in a server binding handle. The timeout value specifies the relative amount of time to spend trying to communicate with the server. Depending on the protocol sequence for the specified binding handle, the value in *timeout* acts only as advice to the RPC runtime.

The **rpc_mgmt_set_com_timeout(3rpc)** reference page explains the timeout values returned in *timeout*.

To change the timeout value, a client calls **rpc_mgmt_set_com_timeout()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_mgmt_set_com_timeout(3rpc)**.

rpc_mgmt_inq_dflt_protect_level

Purpose

Returns the default protection level for an authentication service; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_inq_dflt_protect_level(
        unsigned32 authn_svc,
        unsigned32 *protect_level,
        unsigned32 *status);
```

Parameters

Input

authn_svc Specifies the authentication service for which to return the default protection level.

The supported authentication services are as follows:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

rpc_c_authn_default
DCE default authentication service.

Output

protect_level Returns the default protection level for the specified authentication service. The protection level determines the degree to which

authenticated communications between the client and the server are protected.

The possible protection levels are as follows:

rpc_c_protect_level_default
: Uses the default protection level for the specified authentication service.

rpc_c_protect_level_none
: Performs no protection.

rpc_c_protect_level_connect
: Performs protection only when the client establishes a relationship with the server.

rpc_c_protect_level_call
: Performs protection only at the beginning of each remote procedure call when the server receives the request.

rpc_c_protect_level_pkt
: Ensures that all data received is from the expected client.

rpc_c_protect_level_pkt_integ
: Ensures and verifies that none of the data transferred between client and server has been modified.

rpc_c_protect_level_pkt_privacy
: Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_inq_dflt_protect_level()** routine returns the default protection level for the specified authentication service.

A client can call this routine to learn the default protection level before specifying **rpc_c_protect_level_default** for the *protect_level* parameter in the **rpc_binding_set_auth_info()** routine. If the default level is inappropriate, the client can specify a different, explicit level.

A called remote procedure within a server application can call this routine to obtain the default protection level for a given authentication service. By calling routine **rpc_binding_inq_auth_client()** in the remote procedure, the server can obtain the protection level set up by the calling client. The server can then compare the client-specified protection level with the default level to determine whether to allow the remote procedure to execute.

Alternatively, a remote procedure can compare the client's protection level against a level other than the default level. In this case there is no need for the server's remote procedure to call this routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_unknown_authn_service
Unknown authentication service.

Related Information

Functions: **rpc_binding_inq_auth_client(3rpc)**, **rpc_binding_set_auth_info(3rpc)**.

rpc_mgmt_inq_if_ids

Purpose

Returns a vector of interface identifiers of interfaces a server offers; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_inq_if_ids(
        rpc_binding_handle_t binding,
        rpc_if_id_vector_t **if_id_vector,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a binding handle. To receive interface identifiers from a remote application, specify a server binding handle for that application. To receive interface information about your own (local) application, specify NULL.

If the binding handle you supply refers to partially bound binding information and the binding information contains a nil object UUID, this routine returns the **rpc_s_binding_incomplete** status code. In this case, the DCE host daemon (**dced**) does not know which server instance to select from the local endpoint map because the RPC management interface is automatically registered (by the RPC runtime) for all RPC servers.

To avoid this situation, you can obtain a fully bound server binding handle by calling the **rpc_ep_resolve_binding()** routine.

Output

if_id_vector Returns the address of an interface identifier vector.

status Returns the status code from this routine, which indicates whether the routine completed successfully or, if not, why not. *status* can also return the value of parameter *status* from the application-defined authorization function (**rpc_mgmt_authorization_fn_t**). The prototype for such a function is defined in the *authorization_fn* parameter listed in the reference page for the **rpc_mgmt_set_authorization_fn(3rpc)** routine.

Description

An application calls the **rpc_mgmt_inq_if_ids()** routine to obtain a vector of interface identifiers listing the interfaces registered by a server with the RPC runtime.

If a server has not registered any interfaces with the runtime, this routine returns a **rpc_s_no_interfaces** status code and an *if_id_vector* parameter value of NULL.

The application calls the **rpc_if_id_vector_free()** routine to release the memory used by the vector.

By default, the RPC runtime allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application supplies an authorization function using the **rpc_mgmt_set_authorization_fn()** routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_no_interfaces
No interfaces registered.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_resolve_binding(3rpc)**, **rpc_if_id_vector_free(3rpc)**, **rpc_mgmt_set_authorization_fn(3rpc)**, **rpc_server_register_if(3rpc)**.

rpc_mgmt_inq_server_princ_name

Purpose

Returns a server's principal name; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_inq_server_princ_name(
        rpc_binding_handle_t binding,
        unsigned32 authn_svc,
        unsigned_char_t **server_princ_name,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a binding handle. If a client application wants the principal name from a server application, supply a server binding handle for that server. For a server application to receive a principal name of itself, supply the value NULL.

If the binding handle you supply refers to partially bound binding information and the binding information contains a nil object UUID, this routine returns the **rpc_s_binding_incomplete** status code. In this case the DCE host daemon does not know which server instance to select from the local endpoint map because the RPC runtime automatically registers the RPC management interface for all RPC servers.

You can avoid this situation by calling **rpc_ep_resolve_binding()** to obtain a fully bound server binding handle.

authn_svc Specifies the authentication service for which a principal name is returned. The **rpc_binding_set_auth_info(3rpc)** reference page, in its explanation of the *authn_svc* parameter, contains a list of supported authentication services.

Output

server_princ_name
: Returns a principal name. This name is registered for the authentication service in parameter *authn_svc* by the server referenced in parameter *binding*. If the server registered multiple principal names, only one of them is returned.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

  In addition to the above values, *status* can return the value of parameter *status* from the application-defined authorization function (**rpc_mgmt_authorization_fn_t**). The prototype for such a function is defined in the *authorization_fn* parameter in the reference page for **rpc_mgmt_set_authorization_fn(3rpc)**.

Description

An application calls the **rpc_mgmt_inq_server_princ_name()** routine to obtain the principal name of a server registered for a specified authentication service.

A client (or management) application uses this routine when it wants to allow one-way authentication with the server specified by *binding*. This means that the client does not care which server principal receives the remote procedure call request. However, the server verifies that the client is who the client claims to be. For one-way authentication, a client calls this routine before calling **rpc_binding_set_auth_info()**.

A server application uses this routine to obtain the principal name it registered by calling **rpc_server_register_auth_info()**.

The RPC runtime allocates memory for the string returned in *server_princ_name*. The application calls **rpc_string_free()** to deallocate that memory.

By default, the RPC runtime allows all clients to call this routine remotely. To restrict these calls, a server application supplies an authorization function by calling **rpc_mgmt_set_authorization_fn()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_unknown_authn_service
Unknown authentication service.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_binding_inq_object(3rpc)**, **rpc_binding_set_auth_info(3rpc)**, **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_set_authorization_fn(3rpc)**, **rpc_server_register_auth_info(3rpc)**, **rpc_string_free(3rpc)**, **uuid_is_nil(3rpc)**.

rpc_mgmt_inq_stats

Purpose

Returns RPC runtime statistics; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_inq_stats(
        rpc_binding_handle_t binding,
        rpc_stats_vector_t **statistics,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a binding handle. To receive statistics about a remote application, specify a server binding handle for that application. To receive statistics about your own (local) application, specify NULL.

If the binding handle you supply refers to partially bound binding information and the binding information contains a nil object UUID, this routine returns the **rpc_s_binding_incomplete** status code. In this case, the DCE host daemon does not know which server instance to select from the local endpoint map because the RPC management interface is automatically registered (by the RPC runtime) for all RPC servers.

To avoid this situation, you can obtain a fully bound server binding handle by calling the **rpc_ep_resolve_binding()** routine.

Output

statistics Returns the statistics vector for the server specified by the *binding* parameter. Each statistic is a value of the type **unsigned32**.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. *status* can also return the value of parameter *status* from

rpc_mgmt_authorization_fn_t, which is the application-defined authorization function. The prototype for such a function is defined in the *authorization_fn* parameter in the reference page for **rpc_mgmt_set_authorization_fn(3rpc)**.

Description

The **rpc_mgmt_inq_stats()** routine returns statistics from the RPC runtime about a specified server.

The explanation of a statistics vector in the **rpc_intro(3rpc)** reference page lists the elements of the vector.

The RPC runtime allocates memory for the statistics vector. The application calls the **rpc_mgmt_stats_vector_free()** routine to release the memory that the statistics vector used.

By default, the RPC runtime allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application supplies an authorization function using the **rpc_mgmt_set_authorization_fn()** routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_set_authorization_fn(3rpc)**, **rpc_mgmt_stats_vector_free(3rpc)**.

rpc_mgmt_is_server_listening

Purpose

Tells whether a server is listening for remote procedure calls; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

boolean32 rpc_mgmt_is_server_listening(
        rpc_binding_handle_t binding,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a server binding handle. To determine if a remote application is listening for remote procedure calls, specify a server binding handle for that application. To determine if your own (local) application is listening for remote procedure calls, specify NULL.

If the binding handle you supply refers to partially bound binding information and the binding information contains a nil object UUID, this routine returns the **rpc_s_binding_incomplete** status code. In this case, the DCE host daemon does not know which server instance to select from the local endpoint map because the RPC management interface is automatically registered (by the RPC runtime) for all RPC servers.

To avoid this situation, you can obtain a fully bound server binding handle by calling the **rpc_ep_resolve_binding()** routine.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. *status* can also return the value of parameter *status* from **rpc_mgmt_authorization_fn_t**, which is the application-defined

authorization function. The prototype for such a function is defined in the *authorization_fn* parameter in the reference page for **rpc_mgmt_set_authorization_fn(3rpc)**.

Description

The **rpc_mgmt_is_server_listening()** routine determines whether the server specified in the *binding* parameter is listening for remote procedure calls.

This routine returns a value of TRUE if the server is blocked in the **rpc_server_listen()** routine.

By default, the RPC runtime allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application supplies an authorization function using the **rpc_mgmt_set_authorization_fn()** routine.

Return Values

Your program must examine the return value of the *status* parameter and the return value of the routine to understand the meaning of the routine value. The following table summarizes the values that this routine can return.

Values Returned by rpc_mgmt_is_server_listening()		
Value Returned	**Status Code**	**Explanation**
TRUE	**rpc_s_ok**	The specified server is listening for remote procedure calls.
FALSE	One of the status codes returned by the *status* parameter	The specified server is not listening for remote procedure calls, or the server cannot be reached.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_set_authorization_fn(3rpc)**, **rpc_server_listen(3rpc)**.

rpc_mgmt_set_authorization_fn

Purpose

Establishes an authorization function for processing remote calls to a server's management routines; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_set_authorization_fn(
        rpc_mgmt_authorization_fn_t authorization_fn,
        unsigned32 *status);
```

Parameters

Input

authorization_fn

Specifies a pointer to an authorization function. The RPC server runtime automatically calls this function whenever the server runtime receives a client request to execute one of the RPC management routines.

Specify NULL to unregister a previously registered authorization function. In this case, the default authorizations (as described later) are used.

The following C definition for **rpc_mgmt_authorization_fn_t** illustrates the prototype for the authorization function:

```
typedef boolean32 (*rpc_mgmt_authorization_fn_t)
  (
   rpc_binding_handle_t client_binding, /* in  */
   unsigned32 requested_mgmt_operation, /* in  */
   unsigned32 *status   /* out */
  );
```

The following table shows the *requested_mgmt_operation* values passed by the RPC runtime to the authorization function.

Operation Values Passed to Authorization Function	
Called Remote Routine	*requested_mgmt_operation* **Value**
rpc_mgmt_inq_if_ids()	**rpc_c_mgmt_inq_if_ids**
rpc_mgmt_inq_server_princ_name()	**rpc_c_mgmt_inq_princ_name**
rpc_mgmt_inq_stats()	**rpc_c_mgmt_inq_stats**
rpc_mgmt_is_server_listening()	**rpc_c_mgmt_is_server_listen**
rpc_mgmt_stop_server_listening()	**rpc_c_mgmt_stop_server_listen**

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_set_authorization_fn()** routine sets up an authorization function to control remote access to the calling server's remote management routines.

If a server does not provide an authorization function, the RPC runtime controls client application access to the server's remote management routines as shown in the next table. In the table, an *enabled* authorization allows all clients to execute the remote routine and a *disabled* authorization prevents all clients from executing the remote routine.

Default Controls for Remote Management Routines	
Remote Routine	**Default Authorization**
rpc_mgmt_inq_if_ids()	Enabled
rpc_mgmt_inq_server_princ_name()	Enabled
rpc_mgmt_inq_stats()	Enabled
rpc_mgmt_is_server_listening()	Enabled
rpc_mgmt_stop_server_listening()	Disabled

A server can modify the default authorizations by calling **rpc_mgmt_set_authorization_fn()** to specify an authorization function. When an authorization function is provided, the RPC runtime automatically calls that function to control the execution of all remote management routines called by clients.

The specified function must provide access control for all of the remote management routines.

If the authorization function returns TRUE, the management routine is allowed to execute. If the authorization function returns FALSE, the management routine does not execute, and the called routine returns to the client the status code returned from the **rpc_mgmt_authorization_fn_t** function. However, if the status code that the **rpc_mgmt_authorization_fn_t** function returns is 0 (zero) or **rpc_s_ok**, then the status code **rpc_s_mgmt_op_disallowed** is returned to the client.

The RPC runtime calls the server-provided authorization function with the following two input arguments:

- The binding handle of the calling client.
- An integer value denoting which management routine the client has called.

Using these arguments, the authorization function determines whether the calling client is allowed to execute the requested management routine. For example, the authorization function can call **rpc_binding_inq_auth_client()** to obtain authentication and authorization information about the calling client and determine if that client is authorized to execute the requested management routine.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_mgmt_ep_unregister(3rpc)**, **rpc_mgmt_inq_if_ids(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**, **rpc_mgmt_inq_stats(3rpc)**, **rpc_mgmt_is_server_listening(3rpc)**, **rpc_mgmt_stop_server_listening(3rpc)**.

rpc_mgmt_set_cancel_timeout

Purpose

Sets the lower bound on the time to wait before timing out after forwarding a cancel; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_set_cancel_timeout(
        signed32 seconds,
        unsigned32 *status);
```

Parameters

Input

seconds An integer specifying the number of seconds to wait for a server to acknowledge a cancel. To specify that a client waits an infinite amount of time, supply the value **rpc_c_cancel_infinite_timeout**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_set_cancel_timeout()** routine resets the amount of time the RPC runtime waits for a server to acknowledge a cancel before orphaning the call.

The application specifies either to wait forever or to wait a length of time specified in seconds. If the value of *seconds* is 0 (zero), the remote procedure call is immediately orphaned when the RPC runtime detects and forwards a pending cancel; control returns immediately to the client application. The default value, **rpc_c_cancel_infinite_timeout**, specifies waiting forever for the call to complete.

The value for the cancel timeout applies to all remote procedure calls made in the current thread. A multithreaded client that wishes to change the timeout value must call this routine in each thread of execution.

For more information about canceled threads and orphaned remote procedure calls, see the *DCE 1.2.2 Application Development Guide—Directory Services.*

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **pthread_cancel(3thr)**, **pthread_setcancel(3thr)**.

rpc_mgmt_set_com_timeout

Purpose

Sets the communications timeout value in a binding handle; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_set_com_timeout(
        rpc_binding_handle_t binding,
        unsigned32 timeout,
        unsigned32 *status);
```

Parameters

Input

binding Specifies the server binding handle whose timeout value is set.

timeout Specifies a communications timeout value.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_set_com_timeout()** routine resets the communications timeout value in a server binding handle. The timeout value specifies the relative amount of time to spend trying to communicate with the server. Depending on the protocol sequence for the specified binding handle, the *timeout* value acts only as advice to the RPC runtime.

After the initial relationship is established, subsequent communications for the binding handle cannot revert to less than the default timeouts for the protocol service. This

means that after setting a short initial timeout and establishing a connection, calls in progress are not timed out any sooner than the default.

Note: Because of differences in underlying transport layers, only the **rpc_c_infinite_binding_timeout** constant changes binding behavior when **rpc_mgmt_set_com_timeout()** is used with connection-oriented RPC.

The timeout value can be any integer value from 0 (zero) to 10. Note that these values do *not* represent seconds. They represent a relative amount of time to spend to establish a client/server relationship (a binding).

Constants are provided for certain values in the timeout range. The following table lists the binding timeout values, describing the DCE RPC predefined values that an application can use for the *timeout* parameter.

Predefined Time-Out Values		
Name	**Value**	**Description**
rpc_c_binding_min_timeout	0	Attempts to communicate for the minimum amount of time for the network protocol being used. This value favors response time over correctness in determining whether the server is running.
rpc_c_binding_default_timeout	5	Attempts to communicate for an average amount of time for the network protocol being used. This value gives equal consideration to response time and correctness in determining whether a server is running. This is the default value.
rpc_c_binding_max_timeout	9	Attempts to communicate for the longest finite amount of time for the network protocol being used. This value favors correctness in determining whether a server is running over response time.
rpc_c_binding_infinite_timeout	10	Attempts to communicate forever.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_timeout
Invalid timeout value.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_mgmt_inq_com_timeout(3rpc)**.

rpc_mgmt_set_server_stack_size

Purpose

Specifies the stack size for each server thread; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_set_server_stack_size(
        unsigned32 thread_stack_size,
        unsigned32 *status);
```

Parameters

Input

thread_stack_size
: Specifies, in bytes, the stack size allocated for each thread created by **rpc_server_listen()**. This value is applied to all threads created for the server. Select this value based on the stack requirements of the remote procedures offered by the server.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_mgmt_set_server_stack_size()** routine specifies the thread stack size to use when the RPC runtime creates call threads for executing remote procedure calls. The *max_calls_exec* parameter in **rpc_server_listen()** specifies the number of call execution threads created.

A server, provided it knows the stack requirements of all the manager routines in the interfaces it offers, can call **rpc_mgmt_set_server_stack_size()** to ensure that each call thread has the necessary stack size.

This routine is optional. When it is used, it must be called before the server calls **rpc_server_listen()**. If a server does not call this routine, the default per thread stack size from the underlying threads package is used.

Some thread packages do not support the specification or modification of thread stack sizes. The packages cannot perform such operations or the concept of a thread stack size is meaningless to them.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_arg
Invalid argument.

rpc_s_not_supported
Not supported.

Return Values

No value is returned.

Related Information

Functions: **rpc_server_listen(3rpc)**.

rpc_mgmt_stats_vector_free

Purpose

Frees a statistics vector; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_stats_vector_free(
        rpc_stats_vector_t **stats_vector,
        unsigned32 *status);
```

Parameters

Input/Output

stats_vector Specifies the address of a pointer to a statistics vector. On return, *stats_vector* contains the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application calls **rpc_mgmt_stats_vector_free()** to release the memory used to store a vector of statistics.

An application calls **rpc_mgmt_inq_stats()** to obtain a vector of statistics. Follow a call to **rpc_mgmt_inq_stats()** with a call to **rpc_mgmt_stats_vector_free()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_mgmt_inq_stats(3rpc)**.

rpc_mgmt_stop_server_listening

Purpose

Tells a server to stop listening for remote procedure calls; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_mgmt_stop_server_listening(
        rpc_binding_handle_t binding,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a server binding handle. To direct a remote server to stop listening for remote procedure calls, specify a server binding handle to that server. To direct your own (local) server to stop listening for remote procedure calls, specify NULL.

If the binding handle you supply refers to partially bound binding information and the binding information contains a nil object UUID, this routine returns the **rpc_s_binding_incomplete** status code. In this case, the DCE host daemon does not know which server instance to select from the local endpoint map because the RPC management interface is automatically registered (by the RPC runtime) for all RPC servers.

To avoid this situation, you can obtain a fully bound server binding handle by calling **rpc_ep_resolve_binding()**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. *status* can also return the value of parameter *status* from **rpc_mgmt_authorization_fn_t()**, which is the application-

defined authorization function. The prototype for such a function is defined in the *authorization_fn* parameter in the reference page for **rpc_mgmt_set_authorization_fn(3rpc)**.

Description

The **rpc_mgmt_stop_server_listening()** routine directs a server to stop listening for remote procedure calls.

On receiving such a request, the DCE RPC runtime stops accepting new remote procedure calls. Executing calls are allowed to complete.

After all calls complete, **rpc_server_listen()** returns to the caller.

By default, the RPC runtime does not allow any client to remotely call this routine. To allow clients to execute this routine, a server application supplies an authorization function using **rpc_mgmt_set_authorization_fn()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_binding_incomplete
Binding incomplete (no object ID and no endpoint).

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_unknown_if
Unknown interface.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_set_authorization_fn(3rpc)**, **rpc_server_listen(3rpc)**.

rpc_network_inq_protseqs

Purpose

Returns all protocol sequences supported by both the RPC runtime and the operating system; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_network_inq_protseqs(
        rpc_protseq_vector_t **protseq_vector,
        unsigned32 *status);
```

Parameters

Input

None.

Output

protseq_vector
: Returns the address of a protocol sequence vector.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_network_inq_protseqs()** routine obtains a vector containing the protocol sequences supported by the RPC runtime and the operating system. A server chooses to accept remote procedure calls over some or all of the supported protocol sequences. If there are no supported protocol sequences, this routine returns the **rpc_s_no_protseqs** status code and the value NULL in the *protseq_vector* parameter.

The application calls **rpc_protseq_vector_free()** to release the memory used by the vector.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

Functions: **rpc_network_is_protseq_valid(3rpc)**, **rpc_protseq_vector_free(3rpc)**.

rpc_network_is_protseq_valid

Purpose

Tells whether the specified protocol sequence is supported by both the RPC runtime and the operating system; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

boolean32 rpc_network_is_protseq_valid(
        unsigned_char_t *protseq,
        unsigned32 *status);
```

Parameters

Input

protseq Specifies a string identifier for a protocol sequence. (See the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page for a list of acceptable values.)

The **rpc_network_is_protseq_valid()** routine determines whether this parameter contains a valid protocol sequence. If not, the routine returns FALSE and the *status* parameter contains the **rpc_s_invalid_rpc_protseq** status code.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_network_is_protseq_valid()** routine determines whether a specified protocol sequence is available for making remote procedure calls. A server chooses to accept remote procedure calls over some or all of the supported protocol sequences.

A protocol sequence is valid if the RPC runtime and the operating system support the protocol sequence. DCE RPC supports the protocol sequences pointed to by the explanation of the *protseq* parameter.

An application calls **rpc_network_inq_protseqs()** to obtain all the supported protocol sequences.

Return Values

This routine can return the following values:

TRUE — The RPC runtime supports the protocol sequence specified in the *protseq* parameter. The routine returns the status code **rpc_s_ok** in the *status* parameter.

FALSE — The RPC runtime does not support the protocol sequence specified in the *protseq* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_network_inq_protseqs(3rpc)**, **rpc_string_binding_parse(3rpc)**.

rpc_ns_binding_export

Purpose

Establishes a name service database entry with binding handles or object UUIDs for a server; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_export(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter.

  To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the value **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry name to which binding handles and object UUIDs are exported. This can be either the global or cell-relative name.

if_handle
: Specifies a stub-generated data structure that identifies the interface to export. Specifying the value NULL indicates there are no binding handles to export (only object UUIDs are exported) and the *binding_vec* parameter is ignored.

binding_vec Specifies a vector of server bindings to export. Specify the value NULL for this parameter in cases where there are no binding handles to export (only object UUIDs are exported).

object_uuid_vec
Identifies a vector of object UUIDs offered by the server. The server application constructs this vector. NULL indicates there are no object UUIDs to export (only binding handles are exported).

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_export()** routine allows a server application to publicly offer, in the name service database, an interface that any client application can use. A server application can also use this routine to publicly offer the object UUIDs of the application's resources.

To export an interface, the server application calls the routine with an interface and the server binding handles that a client can use to access the server.

A server can export interfaces and objects in a single call to this routine, or it can export them separately.

If the entry in the name service database specified by the *entry_name* parameter does not exist, **rpc_ns_binding_export()** tries to create it. In this case a server must have the correct permissions to create the entry. Otherwise, a management application with the necessary permissions creates the entry by calling **rpc_ns_mgmt_entry_create()** before the server runs.

A server is not required to export its interfaces to the name service database. When a server does not export any interfaces, only clients that privately know of that server's binding information can access its interfaces. For example, a client that has the information needed to construct a string binding can call **rpc_binding_from_string_binding()** to create a binding handle for making remote procedure calls to a server.

Before calling **rpc_ns_binding_export()** to export interfaces (but not to export object UUIDs), a server must do the following:

- Register one or more protocol sequences with the local RPC runtime by calling one of the following routines:
  - **rpc_server_use_protseq()**
  - **rpc_server_use_protseq_ep()**
  - **rpc_server_use_protseq_if()**
  - **rpc_server_use_all_protseqs()**
  - **rpc_server_use_all_protseqs_if()**
- Obtain a list of server bindings by calling **rpc_server_inq_bindings()**.

  The vector returned from **rpc_server_inq_bindings()** becomes the *binding_vec* parameter for this routine. To prevent a binding from being exported, set the selected vector element to the value NULL. (See the section on RPC data types and structures in the **rpc_intro(3rpc)** reference page.)

If a server exports an interface to the same entry in the name service database more than once, the second and subsequent calls to this routine add the binding information and object UUIDs only if they differ from the ones in the server entry. Existing data is not removed from the entry.

To remove binding handles and object UUIDs from the name service database, a server application calls **rpc_ns_binding_unexport()** and a management application calls **rpc_ns_mgmt_binding_unexport()**.

For an explanation of how a server can establish a client/server relationship without using the name service database, see the explanation of a string binding in the **rpc_intro(3rpc)** reference page.

In addition to calling this routine, a server that called either **rpc_server_use_all_protseqs()** or **rpc_server_use_protseq()** must also register with the local endpoint map by calling either **rpc_ep_register()** or **rpc_ep_register_no_replace()**.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target name service entry). If the entry does not exist, you also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_nothing_to_export
Nothing to export.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_unexport(3rpc)**, **rpc_ns_mgmt_binding_unexport(3rpc)**, **rpc_ns_mgmt_entry_create(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_ns_binding_import_begin

Purpose

Creates an import context for an interface and an object in the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_import_begin(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_t *obj_uuid,
        rpc_ns_handle_t *import_context,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of parameter *entry_name*. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the value **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry name with which the search for compatible binding handles begins. This can be either the global or the cell-relative name.

  To use the entry name found in the **RPC_DEFAULT_ENTRY** environment variable, supply NULL or a null string (**\0**) for this parameter. When this entry name is used, the RPC runtime automatically uses the default name syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

if_handle
: A stub-generated data structure specifying the interface to import. If the interface specification has not been exported or is of no concern to

the caller, specify NULL for this parameter. In this case the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and, depending on the value of parameter *obj_uuid*, contain the specified object Universal Unique Identifier (UUID). The desired interface may not be supported by the contacted server.

obj_uuid Specifies an optional object UUID.

If you specify NULL or a nil UUID for this parameter, the returned binding handles contain one of the object UUIDs that the compatible server exported. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

If you specify a nonnil UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID. Each returned binding handle contains the specified nonnil object UUID.

Output

import_context

Returns the name service handle for use with the following routines:

- **rpc_ns_binding_import_next()**
- **rpc_ns_binding_import_done()**

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_object
Invalid object.

rpc_s_no_env_setup
Environment variable not set up.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Description

The **rpc_ns_binding_import_begin()** routine creates an import context for importing compatible server binding handles for servers. These servers offer the specified interface and object UUID in the respective *if_handle* and *obj_uuid* parameters.

Before calling **rpc_ns_binding_import_next()**, the client must first call this routine to create an import context. The arguments to this routine control the operation of **rpc_ns_binding_import_next()**.

After importing binding handles, the client calls **rpc_ns_binding_import_done()** to delete the import context.

Return Values

No value is returned.

Related Information

Functions: **rpc_ns_binding_import_done(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**.

rpc_ns_binding_import_done

Purpose

Deletes the import context for searching the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_import_done(
        rpc_ns_handle_t* import_context,
        unsigned32 *status);
```

Parameters

Input/Output

import_context

Specifies the name service handle to delete. (A name service handle is created by calling **rpc_ns_binding_import_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_import_done()** routine deletes an import context created by calling **rpc_ns_binding_import_begin()**. This deletion does not affect any previously imported bindings.

Typically, a client calls this routine after completing remote procedure calls to a server using a binding handle returned from **rpc_ns_binding_import_next()**. A client program calls this routine for each created import context, regardless of the status

returned from **rpc_ns_binding_import_next()**, or the success in making remote procedure calls.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_binding_import_begin(3rpc)**, **rpc_ns_binding_import_next(3rpc)**.

rpc_ns_binding_import_next

Purpose Returns a binding handle of a compatible server (if found) from the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_import_next(
        rpc_ns_handle_t import_context,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

Parameters

Input

import_context
: Specifies a name service handle. This handle is returned from the **rpc_ns_binding_import_begin()** routine.

Output

binding
: Returns a compatible server binding handle.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_import_next()** routine returns one compatible (to the client) server binding handle selected at random from the name service database. The server offers the interface and object UUID specified by the respective *if_handle* and *obj_uuid* parameters in **rpc_ns_binding_import_begin()**.

A similar routine is **rpc_ns_binding_lookup_next()**, which returns a vector of compatible server binding handles for one or more servers.

Note: The routine **rpc_ns_binding_import_next()** calls the routine **rpc_ns_binding_lookup_next()** which, in turn, obtains a vector of server binding handles from the name service database. Next, routine **rpc_ns_binding_import_next()** randomly selects one of the elements from the vector.

The **rpc_ns_binding_import_next()** routine communicates only with the name service database, not directly with servers.

The returned compatible binding handle always contains an object UUID. Its value depends on the value specified in the *obj_uuid* parameter of the **rpc_ns_binding_import_begin()** routine, as follows:

- If *obj_uuid* contains a nonnil object UUID, the returned binding handle contains that object UUID.
- If *obj_uuid* contains a nil object UUID or NULL, the object UUID returned in the binding handle depends on how the server exported object UUIDs:
  - — If the server did not export any object UUIDs, the returned binding handle contains a nil object UUID.
  - — If the server exported one object UUID, the returned binding handle contains that object UUID.
  - — If the server exported multiple object UUIDs, the returned binding handle contains one of the object UUIDs, selected in an unspecified way.

    Applications should not count on multiple calls to **rpc_ns_binding_import_next()** returning different object UUIDs. In particular, note that each name service entry stores server address information separately from exported object UUIDs. Successive calls to **rpc_ns_binding_import_next()** using the same import context will return exactly one binding for each compatible server address, not the cross product of all compatible server addresses with all exported UUIDs. Each returned binding will contain one of the exported object UUIDs, but applications should not count on any specific selection mechanism for these object UUIDs

The client application can use the returned binding handle to make a remote procedure call to the server. If the client fails to communicate with the server, it can call the **rpc_ns_binding_import_next()** routine again.

Each time the client calls **rpc_ns_binding_import_next()**, the routine returns another server binding handle. The binding handles returned are unordered. Multiple binding handles can refer to different protocol sequences from the same server.

When the search finishes, the routine returns a status code of **rpc_s_no_more_bindings** and returns the value NULL in *binding*.

A client application calls **rpc_ns_binding_inq_entry_name()** to obtain the name of the entry in the name service database where the binding handle came from.

The **rpc_ns_binding_import_next()** routine allocates memory for the returned *binding* parameter. When a client application finishes with the binding handle, it must call **rpc_binding_free()** to deallocate the memory. Each call to **rpc_ns_binding_import_next()** requires a corresponding call to **rpc_binding_free()**.

The client calls the **rpc_ns_binding_import_done()** routine after it has satisfactorily used one or more returned server binding handles. The **rpc_ns_binding_import_done()** routine deletes the import context. The client also calls **rpc_ns_binding_import_done()** if the application wants to start a new search for compatible servers (by calling **rpc_ns_binding_import_begin()**). The order of binding handles returned can be different for each new search. This means that the order in which binding handles are returned to an application can be different each time the application is run.

Permissions Required

You need read permission to the specified CDS object entry (the starting name service entry) and to any CDS object entry in the resulting search path.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_ns_binding_import_begin(3rpc)**, **rpc_ns_binding_import_done(3rpc)**, **rpc_ns_binding_inq_entry_name(3rpc)**, **rpc_ns_binding_lookup_begin(3rpc)**, **rpc_ns_binding_lookup_done(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_ns_binding_select(3rpc)**.

rpc_ns_binding_inq_entry_name

Purpose

Returns the name of an entry in the name service database from which the server binding handle came; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_inq_entry_name(
        rpc_binding_handle_t binding,
        unsigned32 entry_name_syntax,
        unsigned_char_t **entry_name,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a server binding handle whose entry name in the name service database is returned.

entry_name_syntax
An integer value that specifies the syntax of returned parameter *entry_name*. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the value **rpc_c_ns_syntax_default**.

Output

entry_name Returns the name of the entry in the name service database in which *binding* was found. The returned name is a global name.

Specify NULL to prevent the routine from returning this parameter. When you specify this value, the client does not need to call **rpc_string_free()**.

status Returns the status code from this routine, which indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_inq_entry_name()** routine returns the global name of the entry in the name service database from which a binding handle for a compatible server came.

The RPC runtime allocates memory for the string returned in the *entry_name* parameter. Your application calls **rpc_string_free()** to deallocate that memory.

An entry name is associated only with binding handles returned from the following routines:

- **rpc_ns_binding_import_next()**
- **rpc_ns_binding_lookup_next()**
- **rpc_ns_binding_select()**

If the binding handle specified in the *binding* parameter is not returned from an entry in the name service database (for example, the binding handle is created by calling **rpc_binding_from_string_binding()**), this routine returns the **rpc_s_no_entry_name** status code.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_no_entry_name
No entry name for binding.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_ns_binding_select(3rpc)**, **rpc_string_free(3rpc)**.

rpc_ns_binding_lookup_begin

Purpose

Creates a lookup context for an interface and an object in the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_begin(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_t *object_uuid,
        unsigned32 binding_max_count,
        rpc_ns_handle_t *lookup_context,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the value **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry name at which the search for compatible binding handles begins. This can be either the global or cell-relative name.

  To use the entry name found in the **RPC_DEFAULT_ENTRY** environment variable, supply NULL or a null string (**\0**) for this parameter. When this entry name is used, the RPC runtime automatically uses the default name syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable.

if_handle A stub-generated data structure specifying the interface to look up. If the interface specification has not been exported or is of no concern to the caller, specify NULL for this parameter. In this case the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and contain the specified object UUID. The desired interface might not be supported by the contacted server.

object_uuid Specifies an optional object UUID.

If you specify NULL or a nil UUID for this parameter, the returned binding handles contain one of the object UUIDs exported by the compatible server. If the server did not export any object UUIDs, the returned compatible binding handles contain a nil object UUID.

For a nonnil UUID, compatible binding handles are returned from an entry only if the server has exported the specified object UUID. Each returned binding handle contains the specified nonnil object UUID.

binding_max_count
Sets the maximum number of bindings to return in the *binding_vector* parameter of **rpc_ns_binding_lookup_next()**. Specify **rpc_c_binding_max_count_default** to use the default count.

Output

lookup_context
Returns the name service handle for use with the following routines:

- **rpc_ns_binding_lookup_next()**
- **rpc_ns_binding_lookup_done()**

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_lookup_begin()** routine creates a lookup context for locating compatible server binding handles for servers. These servers offer the specified interface and object UUID in the respective *if_handle* and *object_uuid* parameters.

Before calling **rpc_ns_binding_lookup_next()**, the client application must first create a lookup context by calling **rpc_ns_binding_lookup_begin()**. The parameters to this routine control the operation of the routine **rpc_ns_binding_lookup_next()**.

When finished locating binding handles, the client application calls the **rpc_ns_binding_lookup_done()** routine to delete the lookup context.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_object
Invalid object.

rpc_s_no_env_setup
Environment variable not set up.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_binding_lookup_done(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**.

rpc_ns_binding_lookup_done

Purpose

Deletes the lookup context for searching the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_done(
        rpc_ns_handle_t *lookup_context,
        unsigned32 *status);
```

Parameters

Input/Output

lookup_context

Specifies the name service handle to delete. (A name service handle is created by calling **rpc_ns_binding_lookup_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_lookup_done()** routine deletes a lookup context created by calling **rpc_ns_binding_lookup_begin()**.

Typically, a client calls this routine after completing remote procedure calls to a server using a binding handle returned from **rpc_ns_binding_lookup_next()**. A client program calls this routine for each created lookup context, regardless of the status

returned from **rpc_ns_binding_lookup_next()**, or success in making remote procedure calls.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_binding_lookup_begin(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**.

rpc_ns_binding_lookup_next

Purpose

Returns a list of binding handles of one or more compatible servers (if found) from the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_lookup_next(
        rpc_ns_handle_t lookup_context,
        rpc_binding_vector_t **binding_vec,
        unsigned32 *status);
```

Parameters

Input

lookup_context
: Specifies a name service handle. This handle is returned from the **rpc_ns_binding_lookup_begin()** routine.

Output

binding_vec
: Returns a vector of compatible server binding handles.

status
: Returns the status code from this routine, which indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_lookup_next()** routine returns a vector of compatible (to the client) server binding handles. The servers offer the interface and object UUID specified by the respective *if_handle* and *object_uuid* parameters in **rpc_ns_binding_lookup_begin()**. The number of binding handles that **rpc_ns_binding_lookup_next()** attempts to return is the value of *binding_max_count* in the **rpc_ns_binding_lookup_begin()** routine.

A similar routine is **rpc_ns_binding_import_next()**, which returns *one* compatible server binding handle.

The **rpc_ns_binding_lookup_next()** routine communicates only with the name service database, not directly with servers.

This routine traverses entries in the name service database, returning compatible server binding handles from each entry. The routine can return multiple binding handles from each entry. The search operation obeys the following rules for traversing the entries:

- At each entry visited, the search operation randomly processes binding information, then group members, then profile members. Profile members with different priorities are returned according to their priorities, highest priority first.
- The search operation returns members of a group in random order.
- The search operation returns members of a profile with the same priority in random order.

If the entry where the search begins (see the *entry_name* parameter in **rpc_ns_binding_lookup_begin()**) contains binding handles as well as an RPC group and/or a profile, **rpc_ns_binding_lookup_next()** returns the binding handles from *entry_name* before searching the group or profile. This means that **rpc_ns_binding_lookup_next()** can return a partially full vector before processing the members of the group or profile.

Each binding handle in the returned vector always contains an object UUID. Its value depends on the value specified in the *object_uuid* parameter of **rpc_ns_binding_lookup_begin()** as follows:

- If *object_uuid* contains a nonnil object UUID, each returned binding handle contains that object UUID.
- If *object_uuid* contains a nil object UUID or NULL, the object UUID returned in each binding handle depends on how the server exported object UUIDS:
  - — If the server did not export any object UUIDs, each returned binding handle contains a nil object UUID.
  - — If the server exported one object UUID, each returned binding handle contains that object UUID.
  - — If the server exported multiple object UUIDs, the returned binding handle contains one of the object UUIDs, selected in an unspecified way.

> Applications should not count on the binding handles returned from a given entry to contain different object UUIDs. In particular, note that each name service entry stores server address information separately from exported object UUIDs. One or more calls to **rpc_ns_binding_lookup_next()** will return exactly one binding for each compatible server address, not the cross product of all compatible server addresses with all exported UUIDs. Each returned binding will contain one of the exported object UUIDs, but applications should not count on any specific selection mechanism for these object UUIDs.

From the returned vector of server binding handles, the client application can employ its own criteria for selecting individual binding handles, or the application can call **rpc_ns_binding_select()** to select a binding handle. The **rpc_binding_to_string_binding()** and **rpc_string_binding_parse()** routines are useful for a client creating its own selection criteria.

The client application can use the selected binding handle to attempt a remote procedure call to the server. If the client fails to communicate with the server, it can select another binding handle from the vector. When all the binding handles in the vector are used, the client application calls **rpc_ns_binding_lookup_next()** again.

Each time the client calls **rpc_ns_binding_lookup_next()**, the routine returns another vector of binding handles. The binding handles returned in each vector are unordered, as is the order in which the vectors are returned from multiple calls to this routine.

When looking up compatible binding handles from a profile, the binding handles from entries of equal profile priority are unordered in the returned vector. In addition, the vector returned from a call to **rpc_ns_binding_lookup_next()** contains only compatible binding handles from entries of equal profile priority. This means the returned vector may be partially full.

For example, if the *binding_max_count* parameter value in **rpc_ns_binding_lookup_begin()** was **5** and **rpc_ns_binding_lookup_next()** finds only three compatible binding handles from profile entries of priority 0 (zero), **rpc_ns_binding_lookup_next()** returns a partially full binding vector (with three binding handles). The next call to **rpc_ns_binding_lookup_next()** creates a new binding vector and begins looking for compatible binding handles from profile entries of priority 1.

When the search finishes, the routine returns a status code of **rpc_s_no_more_bindings** and returns the value NULL in *binding_vec*.

A client application calls **rpc_ns_binding_inq_entry_name()** to obtain the name of the entry in the name service database where the binding handle came from.

The **rpc_ns_binding_lookup_next()** routine allocates memory for the returned *binding_vec*. When a client application finishes with the vector, it must call **rpc_binding_vector_free()** to deallocate the memory. Each call to **rpc_ns_binding_lookup_next()** requires a corresponding call to **rpc_binding_vector_free()**.

The client calls **rpc_ns_binding_lookup_done()**, which deletes the lookup context. The client also calls **rpc_ns_binding_lookup_done()** if the application wants to start a new search for compatible servers (by calling the routine **rpc_ns_binding_lookup_begin()**). The order of binding handles returned can be different for each new search. This means that the order in which binding handles are returned to an application can be different each time the application is run.

Permissions Required

You need read permission to the specified CDS object entry (the starting name service entry) and to any CDS object entry in the resulting search path.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_binding_to_string_binding(3rpc)**, **rpc_binding_vector_free(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_binding_inq_entry_name(3rpc)**, **rpc_ns_binding_lookup_begin(3rpc)**, **rpc_ns_binding_lookup_done(3rpc)**, **rpc_ns_binding_select(3rpc)**, **rpc_string_binding_parse(3rpc)**.

rpc_ns_binding_select

Purpose

Returns a binding handle from a list of compatible server binding handles; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_select(
        rpc_binding_vector_t *binding_vec,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

Parameters

Input/Output

binding_vec Specifies the vector of compatible server binding handles from which a binding handle is selected. The returned binding vector no longer references the selected binding handle (returned separately in the *binding* parameter).

Output

binding Returns a selected server binding handle.

status Returns the status code from this routine, which indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_select()** routine randomly chooses and returns a server binding handle from a vector of server binding handles.

Each time the client calls **rpc_ns_binding_select()**, the routine returns another binding handle from the vector.

When all of the binding handles are returned from the vector, the routine returns a status code of **rpc_s_no_more_bindings** and returns the value NULL in *binding*.

The select operation allocates storage for the data referenced by the returned *binding* parameter. When a client finishes with the binding handle, it calls **rpc_binding_free()** to deallocate the storage. Each call to the **rpc_ns_binding_select()** routine requires a corresponding call to **rpc_binding_free()**.

Instead of using this routine, client applications can select a binding handle according to their specific needs. In this case the routines **rpc_binding_to_string_binding()** and **rpc_string_binding_parse()** are useful to the applications since the routines work together to extract the individual fields of a binding handle for examination.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_no_more_bindings
No more bindings.

Related Information

Functions: **rpc_binding_free(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_string_binding_parse(3rpc)**.

rpc_ns_binding_unexport

Purpose

Removes the binding handles for an interface, or object UUIDs, from an entry in the name service database; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_binding_unexport(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_handle_t if_handle,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the value **rpc_c_ns_syntax_default**.

entry_name
: Specifies an entry name whose binding handles or object UUIDs are removed. This can be either the global or cell-relative name.

if_handle
: Specifies an interface specification for the binding handles to be removed from the name service database. The value NULL indicates that no binding handles are removed (only object UUIDs are removed).

object_uuid_vec
: Specifies a vector of object UUIDs to be removed from the name service database. The application constructs this vector. The value NULL indicates that no object UUIDs are removed (only binding handles are removed).

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_binding_unexport()** routine allows a server application to unexport (that is, remove) one of the following from an entry in the name service database:

- All the binding handles for an interface.
- One or more object UUIDs for a resource or resources.
- Both binding handles and object UUIDs.

The **rpc_ns_binding_unexport()** routine removes only those binding handles that match the interface UUID and the major and minor interface version numbers found in the *if_handle* parameter. To remove multiple versions of an interface, use **rpc_ns_mgmt_binding_unexport()**.

A server application can remove an interface and objects in a single call to this routine, or it can remove them separately.

If **rpc_ns_binding_unexport()** does not find any binding handles for the specified interface, it returns an **rpc_s_interface_not_found** status code and does not remove the object UUIDs, if any are specified.

If one or more binding handles for the specified interface are found and removed without error, **rpc_ns_binding_unexport()** removes the specified object UUIDs, if any.

If any of the specified object UUIDs are not found, **rpc_ns_binding_unexport()** returns the status code **rpc_s_not_all_objs_unexported**.

A server application, in addition to calling this routine, also calls **rpc_ep_unregister()** to unregister any endpoints that the server previously registered with the local endpoint map.

Use this routine with caution, only when you expect a server to be unavailable for an extended time; for example, when it is permanently removed from service.

Additionally, keep in mind that name service databases are designed to be relatively stable. In replicated name service databases, frequent use of **rpc_ns_binding_export()**

and **rpc_ns_binding_unexport()** causes the name service to remove and replace the same entry repeatedly, and can cause performance problems.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target name service entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_interface_not_found
Interface not found.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_nothing_to_unexport
Nothing to unexport.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ep_unregister(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_ns_mgmt_binding_unexport(3rpc)**.

rpc_ns_entry_expand_name

Purpose

Expands the name of a name service entry; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_entry_expand_name(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned_char_t **expanded_name,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide a value of **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry name to expand. This can be either the global or cell-relative name.

Output

expanded_name
: Returns a pointer to the expanded version of *entry_name*. Do *not* specify NULL since the routine always returns a name string.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application calls **rpc_ns_entry_expand_name()** to obtain a fully expanded entry name.

The RPC runtime allocates memory for the returned *expanded_name* parameter. The application is responsible for calling **rpc_string_free()** for that returned parameter string.

The returned and expanded entry name accounts for local name translations and differences in locally defined naming schemas. For example, suppose the entry in the name service is

/.:/subsys/PrintQ/server1

Upon return from **rpc_ns_entry_expand_name()**, the expanded name could be

/.../abc.com/subsys/PrintQ/server1

For more information about local names and their expansions, see the information on the DCE Directory Service in the *DCE 1.2.2 Administration Guide—Core Components*.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

Related Information

Functions: **rpc_string_free(3rpc)**.

Books: *DCE 1.2.2 Administration Guide—Introduction*.

rpc_ns_entry_inq_resolution

Purpose Resolves the cell namespace components of a name and returns partial results.

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_entry_inq_resolution(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned_char_t **resolved_name,
        unsigned_char_t **unresolved_name,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the argument *entry_name*. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, supply a value of **rpc_c_ns_syntax_default**.

entry_name
: The entry name on which the attempted name resolution is to be done. The name can be specified in either cell-relative or global from.

Input/Output

resolved_name
: Returns a pointer to the resolved portion of the entry name. The *resolved_name* string returned will be null terminated and will not contain trailing component separators (that is, no trailing / (slash) characters).

  If NULL is specified on input for this parameter, nothing will be returned.

unresolved_name
: Returns a pointer to the unresolved portion of the entry name. The *unresolved_name* string returned will be a relative name, containing no leading component separators (that is, it will contain no leading / (slash) characters).

  If NULL is specified on input for this parameter, nothing will be returned.

Output

status
: Returns the status code from this routine. The status code indicates whether the routine completed successfully, or if not, why not.

Description

The **rpc_ns_entry_inq_resolution()** routine attempts to read an entry in the cell namespace. If the entire entry name as specified is successfully read, the full resolution of the entry name (that is, the originally-specified *entry_name*) is returned in *resolved_name* and the status is set to **rpc_s_ok**.

If the read was unsuccessful because the full entry was not found in the cell namespace, then the status code will be set to **rpc_s_partial_results**, and the following will occur:

- The part of the name successfully read will be returned in *resolved_name*
- The remaining (unresolved) part of the name will be returned in *unresolved_name*

Thus, if the status code is **rpc_s_partial_results** and the (nonempty) return parameter *resolved_name* specifies a leaf (not a directory) entry, the contents of *resolved_name* can be used in subsequent calls to the NSI interface to obtain a binding handle for the server that exported to the entry. This behavior allows applications to implement namespace junctions to their own internally-implemented namespaces. Using this routine, clients can attempt to bind to overqualified name entries whose *resolved_name* part is the name of the server entry, and whose *unresolved_name* part is the pathname (meaningful to the server) of some object that is managed by the application. Calling **rpc_ns_entry_inq_resolution()** with the full name allows the client to learn what part of the name denotes the server entry it must import bindings from; it can then bind to the server, passing the rest of the name, which the server interprets as appropriate. The **sec_acl_bind()** routine, for example, works this way.

The RPC runtime allocates memory for the returned *resolved_name* and *unresolved_name* parameters. The application is responsible for calling **rpc_string_free()** to free the allocated memory.

The application requires read permission for the name entries that are resolved within the cell namespace.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_partial_results
The entry name was only partially resolved within the cell namespace and the value of *unresolved_name* points to the residual of the name.

rpc_s_invalid_name_syntax
The requested name syntax is invalid.

rpc_s_unsupported_name_syntax
The requested name syntax is not supported.

Related Information

Functions: **rpc_ns_binding_*()** routines.

rpc_ns_entry_object_inq_begin

Purpose

Creates an inquiry context for viewing the objects of an entry in the name service database; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_begin(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide a value of **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry in the name service database for which object UUIDs are viewed. This can be either the global or cell-relative name.

Output

inquiry_context
: Returns a name service handle for use with the routine **rpc_ns_entry_object_inq_next()**, and with the routine **rpc_ns_entry_object_inq_done()**.

status
: Returns the status code from this routine, indicating whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_entry_object_inq_begin()** routine creates an inquiry context for viewing the object UUIDs exported to *entry_name*.

Before calling **rpc_ns_entry_object_inq_next()**, the application must first call this routine to create an inquiry context.

When finished viewing the object UUIDs, the application calls the **rpc_ns_entry_object_inq_done()** routine to delete the inquiry context.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_binding_export(3rpc)**, **rpc_ns_entry_object_inq_done(3rpc)**, **rpc_ns_entry_object_inq_next(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**.

rpc_ns_entry_object_inq_done

Purpose

Deletes the inquiry context for viewing the objects of an entry in the name service database; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input/Output

inquiry_context

Specifies the name service handle to delete. (A name service handle is created by calling **rpc_ns_entry_object_inq_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_entry_object_inq_done()** routine deletes an inquiry context created by calling **rpc_ns_entry_object_inq_begin()**.

An application calls this routine after viewing exported object UUIDs using the **rpc_ns_entry_object_inq_next()** routine.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_entry_object_inq_begin(3rpc)**, **rpc_ns_entry_object_inq_next(3rpc)**.

rpc_ns_entry_object_inq_next

Purpose

Returns one object at a time from an entry in the name service database; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_entry_object_inq_next(
        rpc_ns_handle_t inquiry_context,
        uuid_t *obj_uuid,
        unsigned32 *status);
```

Parameters

Input

inquiry_context
: Specifies a name service handle. This handle is returned from the **rpc_ns_entry_object_inq_begin()** routine.

Output

obj_uuid
: Returns an exported object UUID.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_entry_object_inq_next()** routine returns one of the object UUIDs exported to an entry in the name service database. The *entry_name* parameter in the **rpc_ns_entry_object_inq_begin()** routine specified the entry.

An application can view all of the exported object UUIDs by repeatedly calling the **rpc_ns_entry_object_inq_next()** routine. When all the object UUIDs are viewed, this

routine returns an **rpc_s_no_more_members** status. The returned object UUIDs are unordered.

The application supplies the memory for the object UUID returned in the *obj_uuid* parameter.

After viewing the object UUIDs, the application must call the **rpc_ns_entry_object_inq_done()** routine to delete the inquiry context.

The order in which **rpc_ns_entry_object_inq_next()** returns object UUIDs can be different for each viewing of an entry. Therefore, the order in which an application receives object UUIDs can be different each time the application is run.

Permissions Required

You need read permission to the CDS object entry (the target name service entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_ns_binding_export(3rpc)**, **rpc_ns_entry_object_inq_begin(3rpc)**, **rpc_ns_entry_object_inq_done(3rpc)**.

rpc_ns_group_delete

Purpose

Deletes a group attribute; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_delete(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 *status);
```

Parameters

Input

group_name_syntax
: An integer value that specifies the syntax of the *group_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide the integer value **rpc_c_ns_syntax_default**.

group_name
: Specifies the RPC group to delete. This can be either the global or cell-relative name.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_delete()** routine deletes the group attribute from the specified entry in the name service database.

Neither the specified entry nor the entries represented by the group members are deleted.

Permissions Required

You need write permission to the CDS object entry (the target group entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_group_member_add(3rpc)**, **rpc_ns_group_member_delete(3rpc)**.

rpc_ns_group_mbr_add

Purpose

Adds an entry name to a group; if necessary, creates the entry; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_add(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

Parameters

Input

group_name_syntax
: An integer value that specifies the syntax of the *group_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

group_name
: Specifies the RPC group that receives a new member. This can be either the global or cell-relative name.

member_name_syntax
: An integer value that specifies the syntax of *member_name*.

  To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Name of the new RPC group member. This can be either the global or cell-relative name.

Output

status Returns the status code from this routine, indicating whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_mbr_add()** routine adds, to the name service database, an entry name as a member to the name service interface (NSI) group attribute of an entry. The *group_name* parameter specifies the entry.

If the specified *group_name* entry does not exist, this routine creates the entry with a group attribute and adds the group member specified by the *member_name* parameter. In this case, the application must have permission to create the entry. Otherwise, a management application with the necessary permissions creates the entry by calling **rpc_ns_mgmt_entry_create()** before the application is run.

An application can add the entry in *member_name* to a group before it creates the entry itself.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target group entry). If the entry does not exist, you also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_group_mbr_remove(3rpc)**, **rpc_ns_mgmt_entry_create(3rpc)**.

rpc_ns_group_mbr_inq_begin

Purpose

Creates an inquiry context for viewing group members; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_begin(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input

group_name_syntax
An integer value that specifies the syntax of the *group_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

group_name Specifies the name of the RPC group to view.

member_name_syntax
An integer value that specifies the syntax of *member_name* in the **rpc_ns_group_mbr_inq_next()** routine.

To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

Output

inquiry_context
Returns a name service handle for use with the following routines:

- **rpc_ns_group_mbr_inq_next()**
- **rpc_ns_group_mbr_inq_done()**

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_mbr_inq_begin()** routine creates an inquiry context for viewing the members of an RPC group.

Before calling **rpc_ns_group_mbr_inq_next()**, the application must first call this routine to create an inquiry context.

When finished viewing the RPC group members, the application calls the **rpc_ns_group_mbr_inq_done()** routine to delete the inquiry context.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_group_mbr_add(3rpc)**, **rpc_ns_group_mbr_inq_done(3rpc)**, **rpc_ns_group_mbr_inq_next(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**.

rpc_ns_group_mbr_inq_done

Purpose

Deletes the inquiry context for a group; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input/Output

inquiry_context

Specifies the name service handle to delete. (A name service handle is created by calling **rpc_ns_group_mbr_inq_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_mbr_inq_done()** routine deletes an inquiry context created by calling **rpc_ns_group_mbr_inq_begin()**.

An application calls this routine after viewing RPC group members using the **rpc_ns_group_mbr_inq_next()** routine.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_group_mbr_inq_begin(3rpc)**, **rpc_ns_group_mbr_inq_next(3rpc)**.

rpc_ns_group_mbr_inq_next

Purpose

Returns one member name at a time from a group; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_inq_next(
        rpc_ns_handle_t inquiry_context,
        unsigned_char_t **member_name,
        unsigned32 *status);
```

Parameters

Input

inquiry_context

Specifies a name service handle. This handle is returned from the **rpc_ns_group_mbr_inq_begin()** routine.

Output

member_name

Returns a pointer to a (global) RPC group member name. The syntax of the returned name is specified by the **rpc_ns_group_mbr_inq_begin()** routine parameter *member_name_syntax*.

Specify NULL to prevent the routine from returning this parameter. In this case, the application does not call **rpc_string_free()**.

status

Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_mbr_inq_next()** routine returns one member of the RPC group specified by the *group_name* parameter in the routine **rpc_ns_group_mbr_inq_begin()**.

An application can view all the members of an RPC group by calling the **rpc_ns_group_mbr_inq_next()** routine repeatedly. When all the group members have been viewed, this routine returns an **rpc_s_no_more_members** status. The returned group members are unordered.

On each call to this routine that returns a member name (as a global name), the RPC runtime allocates memory for the returned *member_name*. The application calls **rpc_string_free()** for each returned *member_name* string.

After viewing the RPC group's members, the application must call the **rpc_ns_group_mbr_inq_done()** routine to delete the inquiry context.

Permissions Required

You need read permission to the CDS object entry (the target group entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_ns_group_mbr_inq_begin(3rpc)**, **rpc_ns_group_mbr_inq_done(3rpc)**, **rpc_string_free(3rpc)**.

rpc_ns_group_mbr_remove

Purpose

Removes an entry name from a group; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_group_mbr_remove(
        unsigned32 group_name_syntax,
        unsigned_char_t *group_name,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

Parameters

Input

group_name_syntax
: An integer value that specifies the syntax of the *group_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

group_name
: Specifies the RPC group from which to remove *member_name*. This can be either the global or cell-relative name.

member_name_syntax
: An integer value that specifies the syntax of *member_name*.

  To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Specifies the member to remove from the name service interface (NSI) group attribute in the *group_name* entry. This member can be either the global or cell-relative name.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_group_mbr_remove()** routine removes a member from the NSI group attribute in the *group_name* entry.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target group entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_group_member_not_found
Group member not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_group_mbr_add(3rpc)**.

rpc_ns_import_ctx_add_eval

Purpose

Adds an evaluation routine to an import context; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_import_ctx_add_eval(
        rpc_ns_handle_t *import_context,
        unsigned32 function_type,
        rpc_ns_handle_t *eval_args,
        void *eval_func,
        void *free_func,
        error_status_t *status);
```

Parameters

Input

import_context
: The name service handle obtained from the **rpc_ns_binding_import_begin()** routine.

func_type
: The type of evaluation function. This value currently must be **rpc_cs_code_eval_func**.

eval_args
: An opaque data type that data used by the evaluation routine.

  Client applications adding a DCE RPC code sets evaluation routine (that is, the routines **rpc_cs_eval_with_universal()** or **rpc_cs_eval_without_universal()**) specify the server's NSI entry name in this parameter.

eval_func
: A function pointer to the evaluation routine to be called from the **rpc_ns_binding_import_next()** routine. The **void** declaration for *eval_func* means that the function does not return a value.

Client applications adding a DCE RPC code sets evaluation routine (that is, the routines **rpc_cs_eval_with_universal()** or **rpc_cs_eval_without_universal()**) specify the routine name in this parameter.

free_func A function pointer to a routine that is invoked from **rpc_ns_binding_import_done()** and which performs application-specific cleanup. Client applications adding a DCE RPC code sets evaluation routine (that is, **rpc_cs_eval_with_universal()** or **rpc_cs_eval_without_universal()**) specify NULL in this parameter.

Output

import_context
Returns the name service handle which contains the following routines:

- **rpc_ns_binding_import_next()**
- **rpc_ns_binding_import_done()**

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_import_ctx_add_eval()** routine adds an evaluation routine to an import context created by the **rpc_ns_binding_import_begin()** routine. The evaluation routine adds additional criteria to that used by **rpc_ns_binding_import_next()** (that is, protocol and interface information) for importing compatible server binding handles. Client applications call the **rpc_ns_import_ctx_add_eval()** routine once for each evaluation routine to be added to an import context (if there are multiple evaluation routines to be set up.)

If the user-specified evaluation routine needs to perform special cleanup functions, such as deleting a temporary file from a disk, use the *free_func* parameter to specify the cleanup routine to be called from **rpc_ns_binding_import_done()**.

For DCE 1.1, client applications that transfer international character data in a heterogeneous character set and code set environment use the **rpc_ns_import_ctx_add_eval()** routine to add one or more code sets evaluation routines to the import context returned by the **rpc_ns_binding_import_begin()** routine. When the client application calls the **rpc_ns_binding_import_next()** routine to import compatible binding handles for servers, this routine calls the

code sets evaluation routine, which applies client-server character set and code sets compatibility checking as another criteria for compatible binding selection.

The code sets compatibility evaluation routine specified can be one of the following:

rpc_cs_eval_with_universal
: A DCE RPC code sets evaluation routine that evaluates character set and code sets compatibility between client and server. If client and server character sets are compatible, but their supported code sets are not, the routine sets code set tags that direct the client and/or server stubs to convert character data to either user-defined intermediate code sets (if they exist) or the DCE intermediate code set, which is the ISO 10646 (or *universal*) code set.

rpc_cs_eval_without_universal
: A DCE RPC code sets evaluation routine that evaluates character set and code sets compatibility between client and server. If client and server character sets are compatible, but their supported code sets are not, the routine attempts to return the message **rpc_s_no_compat_codesets** to **rpc_ns_binding_import_next()**.

application-supplied-routine
: A user-written code sets evaluation routine. Application developers writing internationalized DCE applications can develop their own code sets evaluation routines for client-server code sets evaluation if the DCE-supplied routines do not meet their application's needs.

Restrictions

Client applications that add evaluation routines to server binding import context cannot use the automatic binding method to bind to a server.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_no_memory
The RPC runtime could not allocate heap storage.

rpc_s_invalid_ns_handle
The *import_context* parameter was not valid.

Related Information

Functions: **rpc_cs_eval_with_universal(3rpc)**, **rpc_cs_eval_without_universal(3rpc)**, **rpc_ns_binding_import_begin(3rpc)**, **rpc_ns_binding_import_done(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**.

rpc_ns_mgmt_binding_unexport

Purpose

Removes multiple binding handles, or object UUIDs, from an entry in the name service database; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_binding_unexport(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies an entry name whose binding handles or object UUIDs are removed. This can be either the global or cell-relative name.

if_id
: Specifies an interface identifier for the binding handles to be removed from the name service database. The value NULL indicates that no binding handles are removed (only object UUIDs are removed).

vers_option
: Specifies how the **rpc_ns_mgmt_binding_unexport()** routine uses the *vers_major* and the *vers_minor* fields of the *if_id* parameter.

The following table presents the accepted values for this parameter:

Uses of vers_major and vers_minor fields of if_id	
Value	**Description**
rpc_c_vers_all	Unexports (removes) all bindings for the interface UUID in *if_id*, regardless of the version numbers. For this value, specify 0 (zero) for both the major and minor versions in *if_id*.
rpc_c_vers_compatible	Removes those bindings for the interface UUID in *if_id* with the same major version as in *if_id*, and with a minor version greater than or equal to the minor version in *if_id*.
rpc_c_vers_exact	Removes those bindings for the interface UUID in *if_id* with the same major and minor versions as in *if_id*.
rpc_c_vers_major_only	Removes those bindings for the interface UUID in *if_id* with the same major version as in *if_id* (ignores the minor version). For this value, specify 0 (zero) for the minor version in *if_id*.
rpc_c_vers_upto	Removes those bindings that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if *if_id* contains V2.0 and the name service entry contains binding handles with the versions V1.3, V2.0, and V2.1, the **rpc_ns_mgmt_binding_unexport()** routine removes the binding handles with V1.3 and V2.0.)

object_uuid_vec

Specifies a vector of object UUIDs to be removed from the name service database. The application constructs this vector. The value NULL indicates that no object UUIDs are removed (only binding handles are removed).

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_binding_unexport()** routine allows a management application to unexport (that is, remove) one of the following from an entry in the name service database:

- All the binding handles for a specified interface UUID, qualified by the interface version numbers (major and minor).
- One or more object UUIDs of resources.
- Both binding handles and object UUIDs of resources.

A management application can remove an interface and objects in a single call to this routine, or it can remove them separately.

If the **rpc_ns_mgmt_binding_unexport()** routine does not find any binding handles for the specified interface, the routine returns an **rpc_s_interface_not_found** status and does not remove the object UUIDs, if any are specified.

If one or more binding handles for the specified interface are found and removed without error, **rpc_ns_mgmt_binding_unexport()** removes the specified object UUIDs, if any.

If any of the specified object UUIDs are not found, **rpc_ns_mgmt_binding_unexport()** returns the **rpc_not_all_objs_unexported** status code.

A management application, in addition to calling this routine, also calls the **rpc_mgmt_ep_unregister()** routine to remove any servers that have registered with the local endpoint map.

Use this routine with caution, only when you expect a server to be unavailable for an extended time; for example, when it is permanently removed from service.

Additionally, keep in mind that name service databases are designed to be relatively stable. In replicated name service databases, frequent use of the **rpc_ns_binding_export()** and **rpc_ns_mgmt_binding_unexport()** routines causes the name service to remove and replace the same entry repeatedly, and can cause performance problems.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target name service entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_interface_not_found
Interface not found.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_nothing_to_unexport
Nothing to unexport.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_mgmt_ep_unregister(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_ns_binding_unexport(3rpc)**.

rpc_ns_mgmt_entry_create

Purpose

Creates an entry in the name service database; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_create(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the name of the entry to create. This can be either the global or cell-relative name.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_entry_create()** routine creates an entry in the name service database.

A management application can call **rpc_ns_mgmt_entry_create()** to create an entry in the name service database for use by another application that does not itself have the necessary name service permissions to create an entry.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target name service entry). You also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_already_exists
Name service entry already exists.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_mgmt_entry_delete(3rpc)**.

rpc_ns_mgmt_entry_delete

Purpose

Deletes an entry from the name service database; used by management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_delete(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the name of the entry to delete. This can be either the global or cell-relative name.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_entry_delete()** routine removes an RPC entry from the name service database.

Management applications use this routine only when an entry is no longer needed, such as when a server is permanently removed from service. If the entry is a member of a group or profile, it must also be deleted from the group or profile.

Use this routine cautiously. Since name service databases are designed to be relatively stable, the frequent use of **rpc_ns_mgmt_entry_delete()** can result in the following difficulties:

- Performance problems

  Creating and deleting entries in client or server applications causes the name service to remove and replace the same entry repeatedly in the name service database, which can lead to performance problems.

- Lost entry updates

  When multiple applications access a single entry through different replicas of a name service database, updates to the entry can be lost.

  In this situation, if one application deletes the entry and another application updates the entry before the replicas are synchronized, the delete operation takes precedence over the update operation. When the replicas are synchronized, the update is lost because the entry is deleted from all replicas.

Permissions Required

You need read permission to the CDS object entry (the target name service entry). You also need delete permission to the CDS object entry or to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_mgmt_entry_create(3rpc)**.

rpc_ns_mgmt_entry_inq_if_ids

Purpose

Returns the list of interfaces exported to an entry in the name service database; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_entry_inq_if_ids(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_if_id_vector_t **if_id_vec,
        unsigned32 *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of argument *entry_name*. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the entry in the name service database for which an interface identifier vector is returned. This can be either the global or cell-relative name.

Output

if_id_vec
: Returns the address of the interface identifier vector.

status
: Returns the status code from this routine, indicating whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_entry_inq_if_ids()** routine returns an interface identifier vector containing the interfaces of binding handles exported to argument *entry_name*.

This routine uses an expiration age of 0 (zero) to cause an immediate update of the local copy of name service data. The **rpc_ns_mgmt_inq_exp_age()** routine's reference page contains an explanation of the expiration age.

The application calls **rpc_if_id_vector_free()** to release memory used by the returned vector.

Permissions Required

You need read permission to the CDS object entry (the target name service entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_interfaces_exported
No interfaces were exported to entry.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_if_id_vector_free(3rpc)**, **rpc_if_inq_id(3rpc)**, **rpc_ns_binding_export(3rpc)**.

rpc_ns_mgmt_free_codesets

Purpose

Frees a code sets array that has been allocated by the RPC runtime; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_free_codesets(
        rpc_codeset_mgmt_p_t *code_sets_array,
        error_status_t *status);
```

Parameters

Input/Output

code_sets_array
: A pointer to a code sets array that has been allocated by a call to **rpc_ns_mgmt_read_codesets()** or **rpc_rgy_get_codesets()**.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_free_codesets()** routine belongs to a set of DCE RPC routines for character and code set interoperability. These routines permit client and server applications to transfer international character data in a heterogeneous character set and code sets environment.

The **rpc_ns_mgmt_free_codesets()** routine frees from the client application's memory a code sets array allocated by a client call to the **rpc_ns_mgmt_read_codesets()** or the **rpc_rgy_get_codesets()** routines. The routine frees from a server application's

memory a code sets array allocated by a server call to the **rpc_rgy_get_codesets()** routine.

Client applications use the **rpc_ns_mgmt_read_codesets()** routine to retrieve a server's supported code sets in order to evaluate them against the code sets that the client supports. Clients and servers use the **rpc_rgy_get_codesets()** routine to get their supported code sets from the code set registery. Clients and servers use the **rpc_ns_mgmt_free_codesets()** routine to free the memory allocated to the code sets array as part of their cleanup procedures.

Permissions Required

None.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_rgy_get_codesets(3rpc)**.

rpc_ns_mgmt_handle_set_exp_age

Purpose

Sets a handle's expiration age for local copies of name service data; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_handle_set_exp_age(
        rpc_ns_handle_t ns_handle,
        unsigned32 expiration_age,
        unsigned32 *status);
```

Parameters

Input

ns_handle Specifies the name service handle for which you supply an expiration age. An RPC name service interface (NSI) inquiry begin operation returns a name service handle. An example is the operation that **rpc_ns_entry_object_inq_begin()** performs; it returns a name service handle in its *inquiry_context* parameter.

expiration_age

This integer value specifies the expiration age, in seconds, of local name service data. This data is read by all RPC NSI next routines that use the specified *ns_handle* parameter. An example is the **rpc_ns_entry_object_inq_next()** routine; it accepts a name service handle in its *inquiry_context* parameter.

An expiration age of 0 (zero) causes an immediate update of the local name service data.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_handle_set_exp_age()** routine sets an expiration age for a specified name service handle (in *ns_handle*). The expiration age is the amount of time, in seconds, that a local copy of data from a name service attribute can exist, before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC runtime specifies a random value of between 8 and 12 hours as the default expiration age. The default is global to the application. An expiration age applies only to a specific name service handle and temporarily overrides the current global expiration age.

Normally, avoid using this routine; instead, rely on the application's global expiration age.

A handle's expiration age is used exclusively by RPC NSI next operations (which read data from name service attributes). A next operation normally starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the next operation creates one with fresh attribute data from the name service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application (which in this case is the expiration age set for the name service handle). If the actual age exceeds the handle's expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the next operation fails, returning the **rpc_s_name_service_unavailable** status code.

The scope of a handle's expiration age is a single series of RPC NSI next operations. The **rpc_ns_mgmt_handle_set_exp_age()** routine operates as follows:

1. An RPC NSI begin operation, such as the one performed by **rpc_ns_group_mbr_inq_begin()**, creates a name service handle.
2. A call to **rpc_ns_mgmt_handle_set_exp_age()** creates an expiration age for the handle.
3. A series of corresponding RPC NSI next operations for the name service handle uses the handle's expiration age.
4. A corresponding RPC NSI done operation for the name service handle deletes both the handle and its expiration age.

Permissions Required

No permissions are required.

Cautions

Use this routine with extreme caution.

Setting the handle's expiration age to a small value causes the RPC NSI next operations to frequently update local data for any name service attribute requested by your application. For example, setting the expiration age to 0 (zero) forces the next operation to update local data for the name service attribute requested by your application. Therefore, setting a small expiration age for a name service handle can create performance problems for your application. Also, if your application is using a remote server with the name service database, a small expiration age can adversely affect network performance for all applications.

Limit the use of this routine to the following types of situations:

- When you *must* always get accurate name service data.

  For example, during management operations to update a profile, you may need to always see the profile's current contents. In this case, before beginning to inquire about a profile, your application must call **rpc_ns_mgmt_handle_set_exp_age()** and specify 0 (zero) for the *expiration_age* parameter.

- When a request using the default expiration age fails, and your application needs to retry the operation.

  For example, a client application using import must first try to obtain bindings using the application's default expiration age. However, sometimes the import-next operation returns either no binding handles or an insufficient number of them. In this case, the client can retry the import operation and, after **rpc_ns_binding_import_begin()** terminates, include a **rpc_ns_mgmt_handle_set_exp_age()** routine that specifies 0 (zero) for the *expiration_age* parameter. When the client calls the import-next routine again, the small expiration age for the name service handle causes the import-next operation to update the local attribute data.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_binding_import_begin(3rpc)**, **rpc_ns_binding_lookup_begin(3rpc)**, **rpc_ns_entry_object_inq_begin(3rpc)**, **rpc_ns_group_mbr_inq_begin(3rpc)**, **rpc_ns_mgmt_inq_exp_age(3rpc)**, **rpc_ns_mgmt_set_exp_age(3rpc)**, **rpc_ns_profile_elt_inq_begin(3rpc)**.

rpc_ns_mgmt_inq_exp_age

Purpose

Returns the application's global expiration age for local copies of name service data; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_inq_exp_age(
        unsigned32 *expiration_age,
        unsigned32 *status);
```

Parameters

Input

None.

Output

expiration_age
: Returns the default expiration age (in seconds). All the RPC name service interface (NSI) read operations (all the next operations) use this value.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_inq_exp_age()** routine returns the global expiration age that the application is using. The *expiration_age* parameter represents the amount of time, in seconds, that a local copy of data from a name service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC runtime specifies a random value of between 8 and 12 hours as the default expiration age. The default is global to the application.

The RPC NSI next operations, which read data from name service attributes, use an expiration age. A next operation normally starts by looking for a local copy of the attribute data that an application requests. In the absence of a local copy, the next operation creates one with fresh attribute data from the name service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data from the name service database. If updating is impossible, the old local data remains in place and the next operation fails, returning the **rpc_s_name_service_unavailable** status code.

Applications normally use only the default expiration age. For special cases, an application can substitute a user-supplied global expiration age for the default by calling **rpc_ns_mgmt_set_exp_age()**. The **rpc_ns_mgmt_inq_exp_age()** routine returns the current global expiration age, whether it is a default or a user-supplied value.

An application can also override the global expiration age temporarily by calling **rpc_ns_mgmt_handle_set_exp_age()**.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_ns_mgmt_handle_set_exp_age(3rpc)**, **rpc_ns_mgmt_set_exp_age(3rpc)**.

rpc_ns_mgmt_read_codesets

Purpose

Reads the code sets attribute associated with an RPC server entry in the name service database; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_read_codesets(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        rpc_codeset_mgmt_p_t *code_sets_array,
        error_status_t *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the name of the RPC server entry in the name service database from which to read the code sets attribute. The name can be either the global or cell-relative name.

Output

code_sets_array
: A code sets array that specifies the code sets that the RPC server supports.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_read_codesets()** routine belongs to a set of DCE RPC routines for character and code set interoperability. These routines permit client and server applications to transfer international character data in a heterogeneous character set and code sets environment. The **rpc_ns_mgmt_read_codesets()** routine reads the code sets attribute associated with an RPC server entry in the name service database. The routine takes the name of an RPC server entry and returns a code sets array that corresponds to the code sets that this RPC server supports.

Client applications use the **rpc_ns_mgmt_read_codesets()** routine to retrieve a server's supported code sets in order to evaluate them against the code sets that the client supports. Client applications that use the evaluation routines **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()** do not need to call this routine explicitly, because these code sets evaluation routines call it on the client's behalf. Application developers who are writing their own character and code set evaluation routines may need to include **rpc_ns_mgmt_read_codesets()** in their user-written evaluation routines.

Permissions Required

You need read permission to the target RPC server entry (which is a CDS object).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok

rpc_s_invalid_name_syntax

rpc_s_mgmt_bad_type

rpc_s_name_service_unavailable

rpc_s_no_permission

rpc_s_incomplete_name

rpc_s_no_memory

Related Information

Functions: **dce_cs_rgy_to_loc(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **rpc_ns_mgmt_free_codesets(3rpc)**, **rpc_ns_mgmt_remove_attribute(3rpc)**, **rpc_ns_mgmt_set_attribute(3rpc)**, **rpc_rgy_get_codesets(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**.

rpc_ns_mgmt_remove_attribute

Purpose

Removes an attribute from an RPC server entry in the name service database; used mainly by server applications; can also be used by management applications

Synopsis

```
#include <dce/rpc.h>
#include <dce/nsattrid.h>

void rpc_ns_mgmt_remove_attribute(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        uuid_t *attr_type,
        error_status_t *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the name of the RPC server entry in the name service database from which the attribute will be removed. The name can be either the global or cell-relative name. If you are using this routine to remove a code sets attribute from an RPC server entry in the Cell Directory Service database, then this parameter specifies the CDS name of the server entry that contains the code sets attribute to be removed.

attr_type
: A UUID that specifies the attribute type. For DCE 1.2, this value must be **rpc_c_attr_codesets**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_remove_attribute()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_ns_mgmt_remove_attribute()** routine is designed to be a generic routine for removing an attribute from an RPC server entry in the name service database. The routine removes the attribute from the specified RPC server entry in the name service database. The routine does not remove the RPC server entry.

For DCE 1.2, you use **rpc_ns_mgmt_remove_attribute()** in your application server initialization routine or signal handling routine to remove a code sets attribute from the server's entry in the Cell Directory Service database as part of the server cleanup procedure carried out prior to the server's termination.

A management application can call **rpc_ns_mgmt_remove_attribute()** to remove an attribute from an RPC server entry in the name service database on behalf of an application that does not itself have the necessary name service permissions to remove one.

Permissions Required

You need write permission to the target RPC server entry (which is a CDS object).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
The routine cannot find the RPC server entry specified in the call in the name service database.

rpc_s_incomplete_name
The routine cannot expand the RPC server entry name specified in the call.

rpc_s_invalid_name_syntax
The name syntax specified in the call is not valid.

rpc_s_mgmt_bad_type
The attribute type specified in the call does not match that of the attribute to be removed from the name service database.

rpc_s_name_service_unavailable
The routine was unable to communicate with the name service.

rpc_s_no_ns_permission
The routine's caller does not have the proper permission for an NSI operation.

Related Information

Functions: **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_ns_mgmt_set_attribute(3rpc)**, **rpc_rgy_get_codesets(3rpc)**.

rpc_ns_mgmt_set_attribute

Purpose

Adds an attribute to an RPC server entry in the name service database; used mainly by server applications; can also be used by management applications

Synopsis

```
#include <dce/rpc.h>
#include <dce/nsattrid.h>

void rpc_ns_mgmt_set_attribute(
        unsigned32 entry_name_syntax,
        unsigned_char_t *entry_name,
        uuid_t *attr_type,
        void *attr_value,
        error_status_t *status);
```

Parameters

Input

entry_name_syntax
: An integer value that specifies the syntax of the *entry_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

entry_name
: Specifies the name of the RPC server entry in the name service database with which the attribute will be associated. The name can be either the global or cell-relative name. If you are using this routine to add a code sets attribute to an RPC server entry in the name service database, then this parameter specifies the name of the server entry with which the code sets attribute will be associated.

attr_type
: A UUID that specifies the attribute type. For DCE 1.2, this value must be **rpc_c_attr_codesets**.

attr_val An opaque data structure that specifies the attribute value to be stored in the name service database. If you are using this routine to add a code sets attribute to an RPC server entry, you must cast the representation of the code set data from the data type **rpc_codeset_mgmt_p_t** to the data type **void***.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_set_attribute()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_ns_mgmt_set_attribute()** routine is designed to be a generic routine for adding an attribute to an RPC server entry in the name service database. The routine takes an attribute type and a pointer to the value, and stores the attribute value in the name service database.

For DCE 1.2, you use **rpc_ns_mgmt_set_attribute()** in your application server initialization routine to add a code sets attribute to the server's entry in the Cell Directory Service database (which the initialization routine has created with the **rpc_ns_binding_export()** routine). Because CDS stores integer values in little-endian format, the **rpc_ns_mgmt_set_attribute()** routine also encodes the code sets attribute value into an endian-safe format before storing it in the name service database.

A management application can call **rpc_ns_mgmt_set_attribute()** to add an attribute to an RPC server entry in the name service database on behalf of an application that does not itself have the necessary name service permissions to add one.

Permissions Required

You need both read permission and write permission to the target RPC server entry (which is a CDS object). You also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_name_syntax
The name syntax specified in the call is not valid.

rpc_s_mgmt_bad_type
The attribute type specified in the call does not match that of the attribute to be added to the name service database.

rpc_s_no_memory
The routine was unable to allocate memory to encode the value.

rpc_s_name_service_unavailable
The routine was unable to communicate with the name service.

rpc_s_no_ns_permission
The routine's caller does not have the proper permission for an NSI operation.

Related Information

Functions: **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_ns_mgmt_remove_attribute(3rpc)**, **rpc_rgy_get_codesets(3rpc)**.

rpc_ns_mgmt_set_exp_age

Purpose

Modifies the application's global expiration age for local copies of name service data; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_mgmt_set_exp_age(
        unsigned32 expiration_age,
        unsigned32 *status);
```

Parameters

Input

expiration_age

An integer value that specifies the default expiration age, in seconds, for local name service data. This expiration age applies to all RPC name service interface (NSI) read operations (all the next operations).

An expiration age of 0 (zero) causes an immediate update of the local name service data.

To reset the expiration age to an RPC-assigned random value between 8 and 12 hours, specify a value of **rpc_c_ns_default_exp_age**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_mgmt_set_exp_age()** routine modifies the global expiration age that the application is using. The *expiration_age* parameter represents the amount of time, in

seconds, that a local copy of data from a name service attribute can exist before a request from the application for the attribute requires updating the local copy. When an application begins running, the RPC runtime specifies a random value of between 8 and 12 hours as the default expiration age. The default is global to the application.

Normally, you should avoid using this routine; instead, rely on the default expiration age.

The RPC NSI next operations, which read data from name service attributes, use an expiration age. A next operation normally starts by looking for a local copy of the attribute data that an application requests. In the absence of a local copy, the next operation creates one with fresh attribute data from the name service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data from the name service database. If updating is impossible, the old local data remains in place and the next operation fails, returning the **rpc_s_name_service_unavailable** status code.

Permissions Required

No permissions are required.

Cautions

Use this routine with extreme caution.

Setting the expiration age to a small value causes the RPC NSI next operations to frequently update local data for any name service attribute that your application requests. For example, setting the expiration age to 0 (zero) forces all next operations to update local data for the name service attribute that your application has requested. Therefore, setting small expiration ages can create performance problems for your application. Also, if your application is using a remote server with the name service database, a small expiration age can adversely affect network performance for all applications.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_ns_mgmt_handle_set_exp_age(3rpc)**, **rpc_ns_mgmt_set_exp_age(3rpc)**.

rpc_ns_profile_delete

Purpose

Deletes a profile attribute; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_delete(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        unsigned32 *status);
```

Parameters

Input

profile_name_syntax
: An integer value that specifies the syntax of the *profile_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

profile_name
: Specifies the name of the profile to delete. This can be either the global or cell-relative name.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_delete()** routine deletes the profile attribute from the specified entry in the name service database (the *profile_name* parameter).

Neither the specified entry nor the entry names included as members in each profile element are deleted.

Use this routine cautiously; deleting a profile may break a hierarchy of profiles.

Permissions Required

You need write permission to the CDS object entry (the target profile entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_profile_elt_add(3rpc)**, **rpc_ns_profile_elt_remove(3rpc)**.

rpc_ns_profile_elt_add

Purpose

Adds an element to a profile; if necessary, creates the entry; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_add(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        rpc_if_id_t *if_id,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 priority,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

Parameters

Input

profile_name_syntax
: An integer value that specifies the syntax of the *profile_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

profile_name
: Specifies the RPC profile that receives a new element. This can be either the global or cell-relative name.

if_id
: Specifies the interface identifier of the new profile element. To add or replace the default profile element, specify NULL.

member_name_syntax
: An integer value that specifies the syntax of *member_name*.

To use the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Specifies the entry in the name service database to include in the new profile element. This can be either the global or cell-relative name.

priority An integer value (0 to 7) that specifies the relative priority for using the new profile element during the import and lookup operations. A value of 0 (zero) is the highest priority. A value of 7 is the lowest priority. Two or more elements can have the same priority.

When adding the default profile member, use a value of 0 (zero).

annotation Specifies an annotation string that is stored as part of the new profile element. The string can be up to 17 characters long. Specify NULL or the string **\0** if there is no annotation string.

The string is used by applications for informational purposes only. For example, an application can use this string to store the interface name string (specified in the IDL file).

DCE RPC does not use this string during lookup or import operations, or for enumerating profile elements.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_elt_add()** routine adds an element to the profile attribute of the entry in the name service database specified by the *profile_name* parameter.

If the *profile_name* entry does not exist, this routine creates the entry with a profile attribute and adds the profile element specified by the *if_id*, *member_name*, *priority*, and *annotation* parameters. In this case, the application must have permission to create the entry. Otherwise, a management application with the necessary permissions creates the entry by calling **rpc_ns_mgmt_entry_create()** before the application is run.

If an element with the specified member name and interface identifier are already in the profile, this routine updates the element's priority and annotation string using the values provided in the *priority* and *annotation* parameters.

An application can add the entry in the *member_name* parameter to a profile before it creates the entry itself.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target profile entry). If the entry does not exist, you also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_priority
Invalid profile element priority.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_if_inq_id(3rpc)**, **rpc_ns_mgmt_entry_create(3rpc)**, **rpc_ns_profile_elt_remove(3rpc)**.

rpc_ns_profile_elt_inq_begin

Purpose

Creates an inquiry context for viewing the elements in a profile; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_begin(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        unsigned32 inquiry_type,
        rpc_if_id_t *if_id,
        unsigned32 vers_option,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input

profile_name_syntax
: An integer value that specifies the syntax of the *profile_name* parameter. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

profile_name
: Specifies the name of the profile to view. This can be either the global or cell-relative name.

inquiry_type
: An integer value that specifies the type of inquiry to perform on the profile. The following table describes the valid inquiry types:

Valid Values of *inquiry_ type*	
Value	**Description**
rpc_c_profile_default_elt	Searches the profile for the default profile element, if any. The *if_id*, *vers_option*, and *member_name* parameters are ignored.
rpc_c_profile_all_elts	Returns every element from the profile. The *if_id*, *vers_option*, and *member_name* parameters are ignored.
rpc_c_profile_match_by_if	Searches the profile for those elements that contain the interface identifier specified by the *if_id* and *vers_option* values. The *member_name* parameter is ignored.
rpc_c_profile_match_by_mbr	Searches the profile for those elements that contain the member name specified by the *member_name* parameter. The *if_id* and *vers_option* parameters are ignored.
rpc_c_profile_match_by_both	Searches the profile for those elements that contain the interface identifier and member name specified by the *if_id*, *vers_option*, and *member_name* parameters.

if_id Specifies the interface identifier of the profile elements to be returned by **rpc_ns_profile_elt_inq_next()**.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_if** or **rpc_c_profile_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you can specify the value NULL.

vers_option Specifies how **rpc_ns_profile_elt_inq_next()** uses the *if_id* parameter.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_if** or **rpc_c_profile_match_by_both** for the

inquiry_type parameter. Otherwise, this parameter is ignored and you can specify the value 0 (zero).

The following table describes the valid values for this parameter:

Valid Values of *vers_option*	
Value	**Description**
rpc_c_vers_all	Returns profile elements that offer the specified interface UUID, regardless of the version numbers. For this value, specify 0 (zero) for both the major and minor versions in *if_id*.
rpc_c_vers_compatible	Returns profile elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
rpc_c_vers_exact	Returns profile elements that offer the specified version of the specified interface UUID.
rpc_c_vers_major_only	Returns profile elements that offer the same major version of the specified interface UUID (ignores the minor version). For this value, specify 0 (zero) for the minor version in *if_id*.
rpc_c_vers_upto	Returns profile elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if *if_id* contains V2.0 and the profile contains elements with the versions V1.3, V2.0, and V2.1, **rpc_ns_profile_elt_inq_next()** returns the elements with V1.3 and V2.0.)

member_name_syntax

An integer value that specifies the syntax of the *member_name* parameter in this routine and the syntax of the *member_name* parametr in

rpc_ns_profile_elt_inq_next(). To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name

Specifies the member name that **rpc_ns_profile_elt_inq_next()** looks for in profile elements. This can be either the global or cell-relative name.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_mbr** or **rpc_c_profile_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you specify the value NULL.

Output

inquiry_context

Returns a name service handle for use with the following routines:

- **rpc_ns_profile_elt_inq_next()**
- **rpc_ns_profile_elt_inq_done()**

status Returns the status code from this routine, indicating indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_elt_inq_begin()** routine creates an inquiry context for viewing the elements in a profile.

Using the *inquiry_type* and *vers_option* parameters, an application specifies which of the following profile elements will be returned from calls to **rpc_ns_profile_elt_inq_next()**:

- The default element.
- All elements.
- Those elements with the specified interface identifier.
- Those elements with the specified member name.
- Those elements with both the specified interface identifier and member name.

Before calling **rpc_ns_profile_elt_inq_next()**, the application must first call this routine to create an inquiry context.

When finished viewing the profile elements, the application calls the **rpc_ns_profile_elt_inq_done()** routine to delete the inquiry context.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_if_inq_id(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**, **rpc_ns_profile_elt_inq_done(3rpc)**, **rpc_ns_profile_elt_inq_next(3rpc)**.

rpc_ns_profile_elt_inq_done

Purpose

Deletes the inquiry context for a profile; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_done(
        rpc_ns_handle_t *inquiry_context,
        unsigned32 *status);
```

Parameters

Input/Output

inquiry_context

Specifies the name service handle to delete. (A name service handle is created by calling **rpc_ns_profile_elt_inq_begin()**.)

Returns the value NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_elt_inq_done()** routine deletes an inquiry context created by calling **rpc_ns_profile_elt_inq_begin()**.

An application calls this routine after viewing profile elements using the **rpc_ns_profile_elt_inq_next()** routine.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_profile_elt_inq_begin(3rpc)**, **rpc_ns_profile_elt_inq_next(3rpc)**.

rpc_ns_profile_elt_inq_next

Purpose

Returns one element at a time from a profile; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_inq_next(
        rpc_ns_handle_t inquiry_context,
        rpc_if_id_t *if_id,
        unsigned_char_t **member_name,
        unsigned32 *priority,
        unsigned_char_t **annotation,
        unsigned32 *status);
```

Parameters

Input

inquiry_context

Specifies a name service handle. This handle is returned from the **rpc_ns_profile_elt_inq_begin()** routine.

Output

if_id Returns the interface identifier of the profile element.

member_name

Returns a pointer to the profile element's member name. The name is a global name.

The syntax of the returned name is specified by the **rpc_ns_profile_elt_inq_begin()** *member_name_syntax* parameter.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

priority Returns the profile element priority.

annotation Returns the annotation string for the profile element. If there is no annotation string in the profile element, the string **\0** is returned.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not need to call the **rpc_string_free()** routine.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_elt_inq_next()** routine returns one element from the profile specified by the *profile_name* parameter in the **rpc_ns_profile_elt_inq_begin()** routine.

The selection criteria for the element returned are based on the *inquiry_type* parameter in the **rpc_ns_profile_elt_inq_begin()** routine. The **rpc_ns_profile_elt_inq_next()** routine returns all the components (interface identifier, member name, priority, annotation string) of a profile element.

An application can view all the selected profile entries by repeatedly calling the **rpc_ns_profile_elt_inq_next()** routine. When all the elements have been viewed, this routine returns an **rpc_s_no_more_elements** status code. The returned elements are unordered.

On each call to this routine that returns a profile element, the DCE RPC runtime allocates memory for the returned *member_name* (which points to a global name) and *annotation* strings. The application is responsible for calling the **rpc_string_free()** routine for each returned *member_name* and *annotation* string.

After viewing the profile's elements, the application must call the **rpc_ns_profile_elt_inq_done()** routine to delete the inquiry context.

Permissions Required

You need read permission to the CDS object entry (the target profile entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_elements
No more elements.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_ns_profile_elt_begin(3rpc)**, **rpc_ns_profile_elt_done(3rpc)**, **rpc_string_free(3rpc)**.

rpc_ns_profile_elt_remove

Purpose

Removes an element from a profile; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ns_profile_elt_remove(
        unsigned32 profile_name_syntax,
        unsigned_char_t *profile_name,
        rpc_if_id_t *if_id,
        unsigned32 member_name_syntax,
        unsigned_char_t *member_name,
        unsigned32 *status);
```

Parameters

Input

profile_name_syntax
: An integer value that specifies the syntax of the *profile_name* parameter. To use the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

profile_name Specifies the profile from which to remove an element. This can be either the global or cell-relative name.

if_id Specifies the interface identifier of the profile element to be removed. Specify NULL to remove the default profile member.

member_name_syntax
: An integer value that specifies the syntax of *member_name*. To use the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Specifies the name service entry name in the profile element to remove. This can be either the global or cell-relative name. When *if_id* is NULL, this argument is ignored.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ns_profile_elt_remove()** routine removes a profile element from the profile specified by *profile_name*. Unless *if_id* is NULL, the *member_name* parameter and the *if_id* parameter must match the corresponding profile element attributes exactly for an element to be removed. When *if_id* is NULL, the default profile element is removed, and the *member_name* argument is ignored.

The routine removes the reference to the entry specified by *member_name* from the profile; it does not delete the entry itself.

Use this routine cautiously; removing elements from a profile may break a hierarchy of profiles.

Permissions Required

You need both read permission and write permission to the CDS object entry (the target profile entry).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_profile_element_not_found
Profile element not found.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_profile_delete(3rpc)**, **rpc_ns_profile_elt_add(3rpc)**.

rpc_object_inq_type

Purpose

Returns the type of an object; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_object_inq_type(
        uuid_t *obj_uuid,
        uuid_t *type_uuid,
        unsigned32 *status);
```

Parameters

Input

obj_uuid Specifies the object UUID whose associated type UUID is returned. Supply NULL to specify a nil UUID for this parameter.

Output

type_uuid Returns the type UUID corresponding to the object UUID supplied in the *obj_uuid* parameter.

Specifying NULL here prevents the return of a type UUID. An application, by specifying NULL here, can determine from the value returned in *status* whether *obj_uuid* is registered. This determination occurs without the application specifying an output type UUID variable.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

A server application calls the **rpc_object_inq_type()** routine to obtain the type UUID of an object.

If the object is registered with the RPC runtime using the **rpc_object_set_type()** routine, the registered type is returned.

Optionally, an application can maintain an object/type registration privately. In this case, if the application provides an object inquiry function (see the **rpc_object_set_inq_fn(3rpc)** reference page), the RPC runtime uses that function to determine an object's type.

The table below shows how **rpc_object_inq_type()** obtains the returned type UUID.

Rules for Returning an Object's Type		
Was object UUID registered (using rpc_object_set_type)?	**Was an object inquiry runction registered (using rpc_object_set_inq_fn)?**	**Return Value**
Yes	Ignored	Returns the object's registered type UUID.
No	Yes	Returns the type UUID returned from calling the inquiry function.
No	No	Returns the nil UUID.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_object_not_found
Object not found.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_object_set_inq_fn(3rpc)**, **rpc_object_set_type(3rpc)**.

rpc_object_set_inq_fn

Purpose

Registers an object inquiry function; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_object_set_inq_fn(
        rpc_object_inq_fn_t inquiry_fn,
        unsigned32 *status);
```

Parameters

Input

inquiry_fn Specifies a pointer to an object type inquiry function. When an application calls the **rpc_object_inq_type()** routine and the RPC runtime finds that the specified object is not registered, the runtime automatically calls the **rpc_object_inq_type()** routine to determine the object's type. Specify NULL to remove a previously set inquiry function.

The following C language definition for **rpc_object_inq_fn_t** illustrates the prototype for this function:

```
typedef void (*rpc_object_inq_fn_t)
(
    uuid_t      *object_uuid,  /* in  */
    uuid_t      *type_uuid,    /* out */
    unsigned32   *status       /* out */
);
```

The returned *type_uuid* and *status* values are returned as the output arguments from the **rpc_object_inq_type()** routine.

If you specify NULL, the **rpc_object_set_inq_fn()** routine unregisters (that is, removes) a previously registered object type inquiry function.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

A server application calls **rpc_object_set_inq_fn()** to specify a function to determine an object's type. If an application privately maintains object/type registrations, the specified inquiry function returns the type UUID of an object from that registration.

The RPC runtime automatically calls the inquiry function when the application calls **rpc_object_inq_type()** and the object was not previously registered by **rpc_object_set_type()**. The RPC runtime also automatically calls the inquiry function for every remote procedure call it receives if the object was not previously registered.

Cautions

Use this routine with caution. When the RPC runtime automatically calls this routine in response to a received remote procedure call, the inquiry function can be called from the context of runtime internal threads with runtime internal locks held. The inquiry function should not block or at least not block for long (for example, the inquiry function should not perform a remote procedure call). Also, the inquiry function must not unwind because of an exception. In general, the inquiry function should not call back into the RPC runtime. It is legal to call **rpc_object_set_type()** or any of the **uuid_*** routines. Failure to comply with these restrictions will result in undefined behavior.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_object_inq_type(3rpc)**, **rpc_object_set_type(3rpc)**.

rpc_object_set_type

Purpose

Registers the type of an object with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_object_set_type(
        uuid_t *obj_uuid,
        uuid_t *type_uuid,
        unsigned32 *status);
```

Parameters

Input

obj_uuid Specifies an object UUID to associate with the type UUID in the *type_uuid* parameter. Do not specify NULL or a nil UUID.

type_uuid Specifies the type UUID of the *obj_uuid* parameter.

Specify an argument value of NULL or a nil UUID to reset the object type to the default association of object UUID/nil type UUID.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_object_set_type()** routine assigns a type UUID to an object UUID.

By default, the RPC runtime assumes that the type of all objects is nil. A server program that contains one implementation of an interface (one manager entry point vector) does not need to call this routine, provided that the server registered the

interface with the nil type UUID (see the **rpc_server_register_if(3rpc)** reference page).

A server program that contains multiple implementations of an interface (multiple manager entry point vectors; that is, multiple type UUIDs) calls this routine once for each object UUID the server offers. Associating each object with a type UUID tells the RPC runtime which manager entry point vector (interface implementation) to use when the server receives a remote procedure call for a nonnil object UUID.

The RPC runtime allows an application to set the type for an unlimited number of objects.

To remove the association between an object UUID and its type UUID (established by calling this routine), a server calls this routine again and specifies the value NULL or a nil UUID for the *type_uuid* parameter. This resets the association between an object UUID and type UUID to the default.

A server cannot register a nil object UUID. The RPC runtime automatically registers the nil object UUID with a nil type UUID. Attempting to set the type of a nil object UUID will result in the routine's returning the status code **rpc_s_invalid_object**.

Servers that want to maintain their own object UUID to type UUID mapping can use **rpc_object_set_inq_fn()** in place of, or in addition to, **rpc_object_set_type()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_already_registered
Object already registered.

rpc_s_invalid_object
Invalid object.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_object_set_inq_fn(3rpc)**, **rpc_server_register_if(3rpc)**.

rpc_protseq_vector_free

Purpose

Frees the memory used by a vector and its protocol sequences; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_protseq_vector_free(
        rpc_protseq_vector_t **protseq_vector,
        unsigned32 *status);
```

Parameters

Input/Output

protseq_vector
: Specifies the address of a pointer to a vector of protocol sequences. On return the pointer is set to NULL.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_protseq_vector_free()** routine frees the memory used to store a vector of protocol sequences. The freed memory includes both the protocol sequences and the vector itself.

Call **rpc_network_inq_protseqs()** to obtain a vector of protocol sequences. Follow a call to **rpc_network_inq_protseqs()** with a call to **rpc_protseq_vector_free()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_network_inq_protseqs(3rpc)**.

rpc_rgy_get_codesets

Purpose

Gets supported code sets information from the local host; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_rgy_get_codesets(
        rpc_codeset_mgmt_p_t *code_sets_array,
        error_status_t *status);
```

Parameters

Input

No input is required.

Output

code_sets_array
: An integer array that specifies the code sets that the client's or server's host environment supports. Each array element is an integer value that uniquely identifies one code set.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_rgy_get_codesets()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_rgy_get_codesets()** routine examines the locale environment of the host on which the client or server process is running to determine the local code set currently

in use by the client or server process and the set of supported code set conversion routines that exist on the host into which the client or server process can convert if necessary. It then reads the code sets registry on the local host to retrieve the unique identifiers associated with these supported code sets.

The routine returns a code sets array. The set of values returned in this structure correspond to the process's local code set and the code sets into which processes that run on this host can convert. The array also contains, for each code set, the maximum number of bytes that code set uses to encode one character (*c_max_bytes*).

Server applications use the **rpc_rgy_get_codesets()** routine in their initialization code to get their host's supported character and code sets values in order to export them into the name service database with **rpc_ns_mgmt_set_attribute()**.

Client applications use the **rpc_rgy_get_codesets()** routine during the server binding selection process to retrieve the supported character and code sets at their host in order to evaluate them against the character and code sets that a server supports. Client applications that use the evaluation routines **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()** do not need to call this routine explicitly, because these code sets evaluation routines call it on the client's behalf. Application developers who are writing their own character and code set evaluation routines may need to include **rpc_rgy_get_codesets()** in their user-written evaluation routines.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

rpc_s_ok

rpc_s_no_memory

Related Information

Commands: **csrc(8dce)**.

Functions: **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_ns_mgmt_remove_attribute(3rpc)**, **rpc_ns_mgmt_set_attribute(3rpc)**.

rpc_rgy_get_max_bytes

Purpose

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_rgy_get_max_bytes(
        unsigned32 rgy_code_set_value,
        unsigned16 *rgy_max_bytes,
        error_status_t *status);
```

Parameters

Input

rgy_code_set_value
: The registered hexadecimal value that uniquely identifies the code set.

Output

rgy_max_bytes
: The registered decimal value that indicates the number of bytes this code set uses to encode one character.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_rgy_get_max_bytes()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_rgy_get_max_bytes()** routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

The DCE RPC stub support routines for buffer sizing use the **rpc_rgy_get_max_bytes()** routine as part of their procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets. The DCE RPC stub support routines call the **rpc_rgy_get_max_bytes()** routine once to get the *rgy_max_bytes* value for the code set to be used to transfer the data over the network (the network code set) then call the routine again to get the *rgy_max_bytes* value of their local code set. The stubs then compare the two values to determine whether or not additional buffers are necessary or whether the conversion can be done in place.

Client and server applications that use the following DCE RPC buffer sizing routines do not need to call this routine explicitly because these DCE RPC stub support routines call it on their behalf:

- **byte_net_size()**
- **byte_local_size()**
- **wchar_t_net_size()**
- **wchar_t_local_size()**

Application programmers who are developing their own stub support routines for buffer sizing can use the **rpc_rgy_get_max_bytes()** routine in their code to get code set *max_byte* information for their user-written buffer sizing routines.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

dce_cs_c_notfound

dce_cs_c_unknown

rpc_s_ok

Related Information

Commands: **csrc(8dce)**.

Functions: **dce_cs_loc_to_rgy(3rpc)**, **dce_cs_rgy_to_loc(3rpc)**, **rpc_ns_mgmt_read_code_sets(3rpc)**, **rpc_rgy_get_code_sets(3rpc)**.

rpc_server_inq_bindings

Purpose

Returns binding handles for communications with a server; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_inq_bindings(
        rpc_binding_vector_t **binding_vector,
        unsigned32 *status);
```

Parameters

Input

None.

Output

binding_vector
: Returns the address of a vector of server binding handles.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_inq_bindings()** routine obtains a vector of server binding handles. Binding handles are created by the RPC runtime when a server application calls any of the following routines to register protocol sequences:

- **rpc_server_use_all_protseqs()**
- **rpc_server_use_all_protseqs_if()**
- **rpc_server_use_protseq()**
- **rpc_server_use_protseq_ep()**

- **rpc_server_use_protseq_if()**

The returned binding vector can contain binding handles with dynamic endpoints and binding handles with well-known endpoints, depending on which of the preceding routines the server application called. The **rpc_intro(3rpc)** reference page contains an explanation of dynamic and well-known endpoints.

A server uses the vector of binding handles for exporting to the name service, for registering with the local endpoint map, or for conversion to string bindings.

If there are no binding handles (no registered protocol sequences), this routine returns the **rpc_s_no_bindings** status code and returns the value NULL to the *binding_vector* parameter.

The server is responsible for calling the **rpc_binding_vector_free()** routine to deallocate the memory used by the vector.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_no_bindings
No bindings.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_inq_if

Purpose

Returns the manager entry point vector registered for an interface; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_inq_if(
        rpc_if_handle_t if_handle,
        uuid_t *mgr_type_uuid,
        rpc_mgr_epv_t *mgr_epv,
        unsigned32 *status);
```

Parameters

Input

if_handle Specifies the interface specification whose manager entry point vector (EPV) pointer is returned in the *mgr_epv* parameter.

mgr_type_uuid
Specifies a type UUID for the manager whose EPV pointer is returned in the *mgr_epv* parameter.

Specifying the value NULL (or a nil UUID) has this routine return a pointer to the manager EPV that is registered with *if_handle* and the nil type UUID of the manager.

Output

mgr_epv Returns a pointer to the manager EPV corresponding to *if_handle* and *mgr_type_uuid*.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

A server application calls the **rpc_server_inq_if()** routine to determine the manager EPV for a registered interface and type UUID of the manager.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

Related Information

Functions: **rpc_server_register_if(3rpc)**.

rpc_server_listen

Purpose

Tells the RPC runtime to listen for remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_listen(
        unsigned32 max_calls_exec,
        unsigned32 *status);
```

Parameters

Input

max_calls_exec

Specifies the maximum number of concurrent executing remote procedure calls.

Use the value **rpc_c_listen_max_calls_default** to specify the default value.

Also, the five **rpc_server_use_*_protseq*()** routines limit (according to their *max_call_requests* parameter) the number of concurrent remote procedure call requests that a server can accept.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_listen()** routine makes a server listen for remote procedure calls. DCE RPC allows a server to simultaneously process multiple calls. The *max_calls_exec* parameter specifies the maximum number of concurrent remote procedure calls the

server executes. Each remote procedure call executes in a call execution thread. The implementation of the RPC architecture determines whether it reuses call execution threads for the execution of subsequent remote procedure calls or, instead, it creates a new thread for each execution of a subsequent remote procedure call.

The following conditions affect the number of concurrent remote procedure calls that a server can process:

- Sufficient network resources must be available to accept simultaneous call requests arriving over a particular protocol sequence. The value of *max_call_requests* in the five **rpc_server_use_*_protseq*()** routines advises the RPC runtime about the runtime's request of network resources.
- Enough call threads must be available to execute the simultaneous call requests once they have been accepted. The value of *max_calls_exec* in **rpc_server_listen()** specifies the number of call threads.

These conditions are independent of each other.

A server application that specifies a value for *max_calls_exec* greater than 1 is responsible for concurrency control among the remote procedures since each executes in a separate thread.

If the server receives more remote procedure calls than it can execute (more calls than the value of *max_calls_exec*), the RPC runtime accepts and queues additional remote procedure calls until a call execution thread is available. From the client's perspective, a queued remote procedure call appears the same as one that the server is actively executing. A client call remains blocked and in the queue until any one of the following events occurs:

- The remote procedure call is assigned to an available call execution thread and the call runs to completion.
- The client no longer can communicate with the server.
- The client thread is canceled and the remote procedure call does not complete within the cancel timeout limits.

The implementation of the RPC architecture determines the amount of queuing it provides.

The RPC runtime continues listening for remote procedure calls (that is, the routine does not return to the server) until one of the following events occurs:

- One of the server application's manager routines calls **rpc_mgmt_stop_server_listening()**.

- A client is allowed to, and makes, a remote **rpc_mgmt_stop_server_listening()** call to the server.

On receiving a request to stop listening, the RPC runtime stops accepting new remote procedure calls for all registered interfaces. Executing calls and existing queued calls are allowed to complete.

After all calls complete, **rpc_server_listen()** returns to the caller, which is a server application.

For more information about a server's listening for and handling incoming remote procedure calls, refer to the *DCE 1.2.2 Application Development Guide—Core Components*. It also contains information about canceled threads.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_already_listening
Server already listening.

rpc_s_max_calls_too_small
Maximum calls value too small.

rpc_s_no_protseqs_registered
No protocol sequences registered.

Related Information

Functions: **rpc_mgmt_server_stop_listening(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_server_register_auth_ident

Purpose

Registers user-to-user based authentication information with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_register_auth_ident(
        unsigned_char_p_t *server_princ_name,
        unsigned32 authn_svc,
        rpc_auth_identity_handle_t auth_identity,
        unsigned32 *status);
```

Parameters

Input

server_princ_name
: A pointer to the principal name to use for the server when authenticating remote procedure calls. The content of the name and its syntax is defined by the authentication service in use.

authn_svc
: Specifies the authentication service to use when the server receives a remote procedure call request. The following authentication services are supported:

rpc_c_authn_none
: No authentication.

rpc_c_authn_dce_secret
: DCE shared-secret key authentication.

rpc_c_authn_dce_public
: DCE public key authentication (reserved for future use).

rpc_c_authn_default
: DCE default authentication service.

auth_identity

Specifies a handle for the data structure that contains the client's authentication and authorization credentials appropriate for the selected authentication and authorization services.

When using the **rpc_c_authn_dce_secret** authentication service and any authorization service, this value must be a **sec_login_handle_t**, which can be obtained from one of the following routines:

- **sec_login_setup_identity()**
- **sec_login_get_current_context()**
- **sec_login_import_context()**

Specify NULL to use the default security login context for the current address space.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_auth_ident()** routine registers an authentication service to use for authenticating remote procedure calls to a particular server principal. This routine is used for user-to-user authentication where the server principal's credentials are available, but not the server principal's long-term key. Use the **rpc_server_register_auth_info()** routine for server-key based authentication.

A server calls this routine once for each authentication service and principal name combination that it wants to register. The authentication service specified by a client (using the **rpc_binding_set_auth_info()** routine) must be one of the authentication services registered by the server. If it is not, the client's remote procedure call request fails with an **rpc_s_unknown_authn_service** status code.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_unknown_authn_service
Unknown authentication service.

sec_s_user_to_user_disabled
Account is not allowed to use user-to-user protocol registration.

sec_s_multiple_u2u_req
Server identity has already been registered.

sec_s_svr_type_conflict
Simultaneous registration of both keytable and identity is not suppported. Server has already registered with the **rpc_server_register_auth_info()** routine.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_server_register_auth_info(3rpc)**.

rpc_server_register_auth_info

Purpose

Registers server-key based authentication information with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_register_auth_info(
        unsigned_char_t *server_princ_name,
        unsigned32 authn_svc,
        rpc_auth_key_retrieval_fn_t get_key_fn,
        void *arg,
        unsigned32 *status);
```

Parameters

Input

server_princ_name
: Specifies the principal name to use for the server when authenticating remote procedure calls using the service specified by *authn_svc*. The content of the nam e and its syntax is defined by the authentication service in use.

authn_svc
: Specifies the authentication service to use when the server receives a remote procedure call request. The following authentication services are supported:

rpc_c_authn_none
: No authentication.

rpc_c_authn_dce_secret
: DCE shared-secret key authentication.

rpc_c_authn_dce_public
: DCE public key authentication (reserved for future use).

rpc_c_authn_default
DCE default authentication service.

get_key_fn Specifies the address of a server-provided routine that returns encryption keys.

The following C definition for **rpc_auth_key_ret rieval_fn_t** illustrates the prototype for the encryption key acquisition routine:

```
typedef void (*rpc_auth_key_retrieval_fn_t)
 (
  void            *arg,                 /* in */
  unsigned_char_t *server_princ_name,   /* in */
  unsigned32      key_type,             /* in */
  unsigned32      key_ver,              /* in */
  void            **key,               /* out */
  unsigned32      *status              /* out */
 );
```

The RPC runtime passes the *server_princ_name* parameter value specified on the call to **rpc_server_register_auth_info( )**, as the *server_princ_name* parameter value, to the *get_key_fn* key acquisition routine. The RPC runtime automatically provides a value for the key version (*key_ver*) parameter. For a *key_ver* value of 0 (zero), the key acquisition routine must return the most recent key available. The routine returns the key in the *key* parameter.

Note: The *key_type* parameter specifies a Kerberos encryption key type. Because currently the DCE supports only DES encryption, this parameter can be ignored.

If the key acquisition routine, when called from the **rpc_server_register_auth_info( )** routine, returns a status other than **rpc_s_ok**, the **rpc_server_register_auth_info( )** routine fails and returns the error status to the calling server.

If the key acquisition routine, when called by the RPC runtime while authenticating a client remote procedure call request, returns a status other than **rpc_s_ok**, the request fails and the RPC runtime returns the error status to the client.

arg Specifies an argument to pass to the *get_key_fn* key acquisition routine, if specified. (See the description of the *get_key_fn* parameter for details.)

Specify NULL for *arg* to use the default key table file, **/krb/v5srvtab**. The calling server must be root to access this file.

If *arg* is a key table filename, the file must have been created with the **ktadd** command. If the specified key table file resides in **/krb5**, you can supply only the filename. If the file does not reside in **/krb5**, you must supply the full pathname. You must prepend the file's absolute pathname with the prefix **FILE:**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_auth_info()** routine registers an authentication service to use for authenticating remote procedure calls to a particular server principal. This routine is used for server-key based authentication. Use the **rpc_server_register_auth_ident()** routine for user-to-user authentication.

A server calls this routine once for each authentication service and principal name combination that it wants to register. The authentication service specified by a client (using the **rpc_binding_set_auth_info()** routine) must be one of the authentication services registered by the server. If it is not, the client's remote procedure call request fails with an **rpc_s_unknown_authn_service** status code.

The following table shows the RPC runtime behavior for acquiring encryption keys for each supported authentication service. Note that if *authn_svc* is **rpc_c_authn_default**, then *get_key_fn* must be NULL.

RPC Key Acquisition for Authentication Services			
authn_svc	*get_key_fn*	*arg*	**Runtime Behavior**
rpc_c_authn_default	NULL	NULL	Uses the default method of encryption key acquisition from the default key table.
rpc_c_authn_default	NULL	non-NULL	Uses the default method of encryption key acquisition from the specified key table.
rpc_c_authn_default	non-NULL	Ignored	Error returned.
rpc_c_authn_none	Ignored	Ignored	No authentication performed.
rpc_c_authn_dce_secret	NULL	NULL	Uses the default method of encryption key acquisition from the default key table.

rpc_c_authn_dce_secret	NULL	non-NULL	Uses the default method of encryption key acquisition from the specified key table.
rpc_c_authn_dce_secret	non-NULL	NULL	Uses the specified encryption key acquisition routine to obtain keys from the default key table.

RPC Key Acquisition for Authentication Services			
authn_svc	*get_key_fn*	*arg*	**Runtime Behavior**
rpc_c_authn_dce_secret	non-NULL	non-NULL	Uses the specified encryption key acquisition routine to obtain keys from the specified key table.
rpc_c_authn_dce_public	Ignored	Ignored	(Reserved for future use.)

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_unknown_authn_service
Unknown authentication service.

rpc_s_key_func_not_allowed
authn_svc is **rpc_c_authn_default** and a nonnull value was supplied for *get_key_fn* parameter.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_server_register_auth_ident(3rpc)**.

rpc_server_register_if

Purpose

Registers an interface with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_register_if(
        rpc_if_handle_t if_handle,
        uuid_t *mgr_type_uuid,
        rpc_mgr_epv_t mgr_epv,
        unsigned32 *status);
```

Parameters

Input

if_handle An IDL-generated data structure specifying the interface to register.

mgr_type_uuid
Specifies a type UUID to associate with the *mgr_epv* parameter. Specifying the value NULL (or a nil UUID) registers the *if_handle* with a nil type UUID.

mgr_epv Specifies the manager routines' entry point vector. To use the IDL-generated default entry point vector, specify NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_if()** routine registers a server interface with the RPC runtime. A server can register an unlimited number of interfaces. Once registered, an interface

is available to clients through any binding handle of the server, provided that the binding handle is compatible for the client.

A server must provide the following information to register an interface:

- An interface specification, which is a data structure generated by the IDL compiler. The server specifies the interface specification of the interface using the *if_handle* parameter.
- A type UUID and manager entry point vector (EPV), a data pair that determines which manager routine executes when a server receives a remote procedure call request from a client.

  The server specifies the type UUID and EPV using the *mgr_type_uuid* and *mgr_epv* parameters, respectively. Note that when a nonnil type UUID is specified, the server must also call the **rpc_object_set_type()** routine to register objects of this nonnil type.

A server that only offers a single manager for an interface calls **rpc_server_register_if()** once for that interface. In the simple case where the single manager's entry point names are the same as the operation names in the IDL interface definition, the IDL-generated default manager EPV for the interface may be used. The value NULL in *mgr_epv* specifies the default manager EPV.

Note that if a server offers multiple implementations of an interface, the server code must register a separate manager entry point vector for each interface implementation.

Rules for Invoking Manager Routines

The RPC runtime dispatches an incoming remote procedure call to a manager that offers the requested RPC interface. When multiple managers are registered for an interface, the RPC runtime must select one of them. To select a manager, the RPC runtime uses the object UUID specified by the call's binding handle. The following table summarizes the rules applied for invoking manager routines.

Rules for Invoking Manager Routines			
Object UUID of Call[1]	**Has Server Set Type of Object UUID?**[2]	**Has Server Set Type for Manager EPV?**[3]	**Dispatching Action**
Nil	Not applicable[4]	Yes	Uses the manager with the nil type UUID.
Nil	Not applicable[4]	No	The RPC error (**rpc_s_unknown_mgr_type**). Rejects the remote procedure call.
Non-nil	Yes	Yes	Uses the manager with the same type UUID.

Rules for Invoking Manager Routines			
Object UUID of Call[1]	**Has Server Set Type of Object UUID?**[2]	**Has Server Registered Type for Manager EPV?**[3]	**Dispatching Action**
Non-nil	No	Ignored	Uses the manager with the nil type UUID. If no manager with the nil type UUID, **rpc_s_unknown_mgr_type**. Rejects the remote procedure call.
Non-nil	Yes	No	The error (**rpc_s_unknown_mgr_type**). Rejects the remote procedure call.

1 This is the object UUID found in a binding handle for a remote procedure.

2 By calling **rpc_object_set_type()** to specify the type UUID for an object.

3 By calling **rpc_server_register_if()** using the same type UUID.

4 The nil object UUID is always automatically assigned the nil type UUID. It is illegal to specify a nil object UUID in **rpc_object_set_type()**.

For more information about registering server interfaces and invoking manager routines, refer to the *DCE 1.2.2 Application Development Guide—Core Components*.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_type_already_registered
An interface with the given type of UUID is already registered.

Related Information

Functions: **rpc_binding_set_object(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_object_set_type(3rpc)**, **rpc_server_unregister_if(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_server_unregister_if

Purpose

Removes an interface from the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_unregister_if(
        rpc_if_handle_t if_handle,
        uuid_t *mgr_type_uuid,
        unsigned32 *status);
```

Parameters

Input

if_handle Specifies an interface specification to unregister (remove).

Specify NULL to remove all interfaces previously registered with the type UUID value given in the *mgr_type_uuid* parameter.

mgr_type_uuid

Specifies the type UUID for the manager entry point vector (EPV) to remove. This needs to be the same value as provided in a call to the **rpc_server_register_if()** routine.

Specify NULL to remove the interface given in the *if_handle* parameter for all previously registered type UUIDs.

Specify a nil UUID to remove the IDL-generated default manager EPV. In this case all manager EPVs registered with a nonnil type UUID remain registered.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_unregister_if()** routine removes the association between an interface and a manager entry point vector (EPV).

Specify the manager EPV to remove by providing, in the *mgr_type_uuid* parameter, the type UUID value specified in a call to the **rpc_server_register_if()** routine. Once removed, an interface is no longer available to client applications.

When an interface is removed, the RPC runtime stops accepting new calls for that interface. Executing calls (on that interface) are allowed to complete.

The table below summarizes the actions of this routine.

Rules for Removing an Interface		
if_handle	*mgr_type_uuid*	**Action**
nonNULL	non-NULL	Removes the manager EPV associated with the specified parameters.
nonNULL	NULL	Removes all manager EPVs associated with parameter *if_handle*.
NULL	non-NULL	Removes all manager EPVs associated with parameter *mgr_type_uuid*.
NULL	NULL	Removes all manager EPVs.

Note that when both of the parameters *if_handle* and *mgr_type_uuid* are given the value NULL, this call has the effect of preventing the server from receiving any new remote procedure calls since all the manager EPVs for all interfaces have been removed.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

Related Information

Functions: **rpc_server_register_if(3rpc)**.

rpc_server_use_all_protseqs

Purpose

Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_all_protseqs(
        unsigned32 max_call_requests,
        unsigned32 *status);
```

Parameters

Input

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_all_protseqs()** routine registers all supported protocol sequences with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains a dynamic endpoint that the RPC runtime and operating system generated.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

After registering protocol sequences, a server typically calls the following routines:

rpc_server_inq_bindings()
Obtains a vector containing all of the server's binding handles.

rpc_ep_register()
Registers the binding handles with the local endpoint map.

rpc_ep_register_no_replace()
Registers the binding handles with the local endpoint map.

rpc_ns_binding_export()
Places the binding handles in the name service database for access by any client.

rpc_binding_vector_free()
Frees the vector of server binding handles.

rpc_server_register_if()
Registers with the RPC runtime those interfaces that the server offers.

rpc_server_listen()
Enables the reception of remote procedure calls.

To register protocol sequences selectively, a server calls one of the following routines:

- **rpc_server_use_protseq()**

- **rpc_server_use_all_protseqs_if()**
- **rpc_server_use_protseq_if()**
- **rpc_server_use_protseq_ep()**

For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_use_all_protseqs_if

Purpose

Tells the RPC runtime to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_all_protseqs_if(
        unsigned32 max_call_requests,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

Parameters

Input

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater that the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

if_handle Specifies an interface specification containing the protocol sequences and their corresponding endpoint information to use in creating binding handles. Each created binding handle contains a well-known (nondynamic) endpoint contained in the interface specification.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_all_protseqs_if()** routine registers all protocol sequences and associated endpoint address information provided in the IDL file with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the IDL file.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

If you want to register selected protocol sequences specified in the IDL, your server uses **rpc_server_use_protseq_if()**.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. (However, a server that uses only **rpc_server_use_all_protseqs_if()** does not subsequently call **rpc_ep_register()** or **rpc_ep_register_no_replace()**.) For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_cant_inq_socket
Cannot inquire endpoint from socket.

rpc_s_invalid_endpoint_format
Invalid interface handle.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_use_protseq

Purpose

Tells the RPC runtime to use the specified protocol sequence for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_protseq(
        unsigned_char_t *protseq,
        unsigned32 max_call_requests,
        unsigned32 *status);
```

Parameters

Input

protseq Specifies a string identifier for the protocol sequence to register with the RPC runtime. (For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.)

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, **rpc_server_listen()** limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq()** routine registers a single protocol sequence with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains a dynamic endpoint that the RPC runtime and operating system generated.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

A server calls **rpc_server_use_all_protseqs()** to register all protocol sequences.

The explanation of the **rpc_server_use_all_protseqs()** routine contains a list of the routines a server typically calls after calling this routine. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_network_is_protseq_valid(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_use_protseq_ep

Purpose

Tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_protseq_ep(
        unsigned_char_t *protseq,
        unsigned32 max_call_requests,
        unsigned_char_t *endpoint,
        unsigned32 *status);
```

Parameters

Input

protseq Specifies a string identifier for the protocol sequence to register with the RPC runtime. (For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, **rpc_server_listen()** limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

endpoint Specifies address information for an endpoint. This information is used in creating a binding handle for the protocol sequence specified in the *protseq* parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq_ep()** routine registers a protocol sequence and its specified endpoint address information with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences and endpoints.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the *endpoint* parameter.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**.

rpc_server_use_protseq_if

Purpose

Tells the RPC runtime to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_protseq_if(
        unsigned_char_t *protseq,
        unsigned32 max_call_requests,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

Parameters

Input

protseq Specifies a string identifier for the protocol sequence to register with the RPC runtime. For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

if_handle Specifies an interface specification whose endpoint information is used in creating a binding for the protocol sequence specified in the *protseq* parameter. Each created binding handle contains a well-known (nondynamic) endpoint contained in the interface specification.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq_if()** routine registers one protocol sequence with the RPC runtime, including its endpoint address information as provided in the specified IDL file.

A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the IDL file.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

To register all protocol sequences from the IDL, a server calls the **rpc_server_use_all_protseqs_if()** routine.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. However, a server that uses only **rpc_server_use_protseq_if()** does not subsequently call **rpc_ep_register()** or **rpc_ep_register_no_replace()**. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**.

rpc_sm_allocate

Purpose

Allocates memory within the RPC stub memory management scheme.

Synopsis

```
#include <rpc.h>

idl_void_p_t rpc_sm_allocate(
        unsigned long size,
        unsigned32 *status);
```

Parameters

Input

size Specifies, in bytes, the size of memory to be allocated.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_allocate()** to allocate memory within the RPC stub memory management scheme. Before a call to this routine, the stub memory management environment must have been established. For manager code that is called from the stub, the stub itself normally establishes the necessary environment. When **rpc_sm_allocate()** is used by code that is not called from the stub, the application must establish the required memory management environment by calling **rpc_sm_enable_allocate()**.

When the stub establishes the memory management environment, the stub itself frees any memory allocated by **rpc_sm_allocate()**. The application can free such memory before returning to the calling stub by calling **rpc_sm_free()**.

When the application establishes the memory management environment, it must free any memory allocated, either by calling **rpc_sm_free()** or by calling **rpc_sm_disable_allocate()**.

Multiple threads may call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory within the stub memory management environment. To do so, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

Return Values

A pointer to the allocated memory.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_free(3rpc)**, **rpc_sm_enable_allocate(3rpc)**, **rpc_sm_disable_allocate(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**.

rpc_sm_client_free

Purpose

Frees memory returned from a client stub

Synopsis

```
#include <rpc.h>

void rpc_sm_client_free(
        idl_void_p_t node_to_free,
        unsigned32 *status);
```

Parameters

Input

node_to_free
: Specifies a pointer to memory returned from a client stub.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_client_free()** routine releases memory allocated and returned from a client stub. The thread calling **rpc_sm_client_free()** must have the same thread handle as the thread that made the RPC call. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_free(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_client_alloc_free(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**, **rpc_sm_swap_client_alloc_free(3rpc)**.

rpc_sm_destroy_client_context

Purpose

Reclaims the client memory resources for a context handle, and sets the context handle to null

Synopsis

```
#include <rpc.h>

void rpc_sm_destroy_client_context(
        idl_void_p_t p_unusable_context_handle,
        unsigned32 *status);
```

Parameters

Input

p_unusable_context_handle
Specifies the context handle that can no longer be accessed.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_destroy_client_context()** routine is used by client applications to reclaim the client resources used in maintaining an active context handle. Applications call this routine after a communications error makes the context handle unusable. When the **rpc_sm_destroy_client_context()** routine reclaims the memory resources, it also sets the context handle to null.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_sm_disable_allocate

Purpose

Releases resources and allocated memory within the stub memory management scheme

Synopsis

```
#include <rpc.h>

void rpc_sm_disable_allocate(
        unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_disable_allocate()** routine releases all resources acquired by a call to **rpc_sm_enable_allocate()**, and any memory allocated by calls to **rpc_sm_allocate()** after the call to **rpc_sm_enable_allocate()** was made.

The **rpc_sm_enable_allocate()** and **rpc_sm_disable_allocate()** routines must be used in matching pairs.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_enable_allocate(3rpc)**.

rpc_sm_enable_allocate

Purpose

Enables the stub memory managment environment

Synopsis

```
#include <rpc.h>

void rpc_sm_enable_allocate(
        unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications can call **rpc_sm_enable_allocate()** to establish a stub memory management environment in cases where one is not established by the stub itself. A stub memory management environment must be established before any calls are made to **rpc_sm_allocate()**. For server manager code called from the stub, the stub memory management environment is normally established by the stub itself. Code that is called from other contexts needs to call **rpc_sm_enable_allocate()** before calling **rpc_sm_allocate()**.

Note: For a discussion of how spawned threads acquire a stub memory management environment, see the **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** reference pages.

Return Values

None

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_disable_allocate(3rpc)**.

rpc_sm_free

Purpose

Frees memory allocated by the **rpc_sm_allocate()** routine

Synopsis

```
#include <rpc.h>

void rpc_sm_free(
        idl_void_p_t node_to_free,
        unsigned32 *status);
```

Parameters

Input

node_to_free
: Specifies a pointer to memory allocated by **rpc_sm_allocate()**.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_free()** to release memory allocated by **rpc_sm_allocate()**.

When the stub allocates memory within the stub memory management environment, manager code called from the stub can also use **rpc_sm_free()** to release memory allocated by the stub.

The thread calling **rpc_sm_free()** must have the same thread handle as the thread that allocated the memory with **rpc_sm_allocate()**. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**.

rpc_sm_get_thread_handle

Purpose

Gets a thread handle for the stub memory management environment

Synopsis

```
#include <rpc.h>

rpc_ss_thread_handle_t rpc_sm_get_thread_handle(
        unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_get_thread_handle()** to get a thread handle for the current stub memory management environment. A thread that is managing memory within the stub memory managment scheme calls pc_sm_get_thread_handle() to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle()** with this handle, is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle()** to get a thread handle before spawning new threads that will manage the same memory. The spawned threads then call **rpc_sm_set_thread_handle()** with the handle provided by the parent thread.

Note: Typically, **rpc_sm_get_thread_handle()** is called by a server manager routine before it spawns additional threads. Normally the stub sets up the

memory management environment for the manager routine. The manager calls **rpc_sm_get_thread_handle()** to make this environment available to the spawned threads.

A thread may also use **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** to save and restore its memory management environment.

Return Values

A thread handle.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_set_thread_handle(3rpc**.

rpc_sm_set_client_alloc_free

Purpose

Sets the memory allocation and freeing mechanisms used by the client stubs

Synopsis

```
#include <rpc.h>

void rpc_sm_set_client_alloc_free(
      idl_void_p_t (*p_allocate) (
      unsigned long size),
      void (*p_free) (
      idl_void_p_t ptr),
      unsigned32 *status);
```

Parameters

Input

p_allocate Specifies a memory allocator routine.

p_free Specifies a memory free routine. This routine is used to free memory allocated with the routine specified by *p_allocate*.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_set_client_alloc_free()** routine overrides the default routines that the client stub uses to manage memory.

Note: The default memory management routines are ISO **malloc()** and ISO **free()** except when the remote call occurs within manager code in which case the default memory management routines are **rpc_sm_allocate()** and **rpc_sm_free()**.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**.

rpc_sm_set_thread_handle

Purpose

Sets a thread handle for the stub memory management environment

Synopsis

```
#include <rpc.h>

void rpc_sm_set_thread_handle(
        rpc_ss_thread_handle_t id,
        unsigned32 *status);
```

Parameters

Input

id Specifies a thread handle returned by a call to the routine **rpc_sm_get_thread_handle()**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application thread calls **rpc_sm_set_thread_handle()** to set a thread handle for memory management within the stub memory management environment. A thread that is managing memory within the stub memory managment scheme calls **rpc_sm_get_thread_handle()** to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle()** with this handle is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle()** to get a thread handle before spawning new threads that will manage the same memory. The spawned

threads then call **rpc_sm_set_thread_handle()** with the handle provided by the parent thread.

Note: Typically, **rpc_sm_set_thread_handle()** is called by a thread spawned by a server manager routine. Normally the stub sets up the memory management environment for the manager routine and the manager calls **rpc_sm_get_thread_handle()** to get a thread handle. Each spawned thread then calls **rpc_sm_get_thread_handle()** to get access to the manager's memory management environment.

A thread may also use **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** to save and restore its memory management environment.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**.

rpc_sm_swap_client_alloc_free

Purpose

Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client

Synopsis

```
#include <rpc.h>

void rpc_sm_swap_client_alloc_free (
      idl_void_p_t (*p_allocate) (
      unsigned long size),
      void (*p_free) (
      idl_void_p_t ptr),
      idl_void_p_t (**p_p_old_allocate) (
      unsigned long size),
      void (**p_p_old_free) (
      idl_void_p_t  ptr),
      unsigned32 *status);
```

Parameters

Input

p_allocate Specifies a new memory allocation routine.

p_free Specifies a new memory free routine.

Output

p_p_old_allocate
Returns the memory allocation routine in use before the call to this routine.

p_p_old_free
: Returns the memory free routine in use before the call to this routine.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_swap_client_alloc_free()** routine exchanges the current allocate and free mechanisms used by the client stubs for routines supplied by the caller.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
: Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_set_client_alloc_free(3rpc)**.

rpc_ss_allocate

Purpose

Allocates memory within the RPC stub memory management scheme; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

idl_void_p_t rpc_ss_allocate(
        idl_size_t size);
```

Parameters

Input

size Specifies, in bytes, the size of memory to be allocated.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

Usually, the **rpc_ss_allocate()** routine is used in the manager code that is called from a server stub. Memory allocated by **rpc_ss_allocate()** is released by the server stub after marshalling any output parameters at the end of the remote call in which the memory was allocated. If you want to release memory allocated by **rpc_ss_allocate()** before returning from the manager code use **rpc_ss_free()**.

You can also use **rpc_ss_free()** in manager code to release memory pointed to by a full pointer (**ptr**) in an input parameter.

When the server uses **rpc_ss_allocate()**, the server stub creates the environment the routine needs. If the parameters of the operation include any pointers other than those used for passing parameters by reference, the environment is set up automatically.

If you need to use **rpc_ss_allocate()** in a manager code routine that does not have a pointer in any of its parameters, use an ACF and apply the **enable_allocate** attribute to the relevant operation. This causes the generated server stub to set up the necessary environment.

Note that memory allocated by allocators other than **rpc_ss_allocate()** is not released when the operation on the server side completes execution.

If you want to use **rpc_ss_allocate()** outside the code called from a server stub, you must first create an environment for it by calling **rpc_ss_enable_allocate()**.

See the *DCE 1.2.2 Application Development Guide—Core Components* for more information.

Return Values

A pointer to the allocated memory.

An exception, **rpc_x_no_memory**, when no memory is available for allocation.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_disable_allocate(3rpc)**, **rpc_ss_enable_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

rpc_ss_bind_authn_client

Purpose

Authenticates a client's identity to a server from a client stub; a pointer to the server binding handle for the remote procedure call to which the routine will add authentication and authorization context

Synopsis

```
#include <rpc.h>

void rpc_ss_bind_authn_client(
        rpc_binding_handle_t *binding,
        if_handle_t if_handle,
        error_status_t *status);
```

Parameters

Input/Output

binding A pointer to the server binding handle for the remote procedure call to which the routine will add authentication and authorization context.

Input

if_handle A stub-generated data structure that specifies the interface of interest. The routine can use this parameter to resolve a partial binding or to distinguish between interfaces.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ss_bind_authn_client()** routine is a DCE-supplied binding callout routine for use with the **binding_callout** ACF interface attribute.

The **binding_callout** attribute enables applications to specify the name of a routine that the client stub will call automatically to modify a server binding handle with additional information before it initiates a remote procedure call. This attribute is especially useful for applications using the automatic binding method, where it is the client stub that obtains the binding handle, rather than the application code. The **binding_callout** attribute provides these applications with a way to gain access to a server binding handle from the client stub, since the handle is not accessible from the application code.

Applications can specify **rpc_ss_bind_authn_client()** to the **binding_callout** ACF interface attribute in order to authenticate the client's identity to a server from the client stub before the remote procedure call to the server is initiated. This routine performs one-way authentication: the client does not care which server principal receives the remote procedure call request, but the server verifies that the client is who the client claims to be.

The routine sets the protection level used, the authentication identity, and the authentication service used to their default values. See the **rpc_binding_set_auth_info(3rpc)** reference page for more information on these default values. It sets the authorization service to perform authorization based on the client's principal name.

Applications can also specify user-written binding callout routines with the **binding_callout** attribute to modify server binding handles from client stubs with other types of information. For more information on using the **binding_callout** ACF attribute, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
Success.

rpc_s_no_more_bindings
Directs the client stub not to look for another server binding.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**.

Books: *DCE 1.2.2 Application Development—Introduction and Style Guide*, *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_client_free

Purpose

Frees memory returned from a client stub; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_client_free(
        idl_void_p_t node_to_free);
```

Parameters

Input

node_to_free Specifies a pointer to memory returned from a client stub.

Description

The **rpc_ss_client_free()** routine releases memory allocated and returned from a client stub. The thread calling **rpc_ss_client_free()** must have the same thread handle as the thread that made the RPC call.

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

Note that while this routine is always called from client code, the code can be executing as part of another server.

Return Values

No value is returned.

Related Information

Functions: **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_client_alloc_free(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**, **rpc_ss_swap_client_alloc_free(3rpc)**.

rpc_ss_destroy_client_context

Purpose

Reclaims the client memory resources for the context handle, and sets the context handle to NULL; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_destroy_client_context(
        void *p_unusable_context_handle);
```

Parameters

Input

p_unusable_context_handle
Specifies the context handle that can no longer be accessed.

Description

The **rpc_ss_destroy_client_context()** routine is used by the client application to reclaim the client resources used in maintaining an active context handle. Only call this after a communications error makes the context handle unusable. When **rpc_ss_destroy_client_context()** reclaims the memory resources, it also sets the context handle to null.

Return Values

No value is returned.

The **rpc_ss_destroy_client_context()** routine raises no exceptions.

rpc_ss_disable_allocate

Purpose

Releases resources and allocated memory; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_disable_allocate(
        void);
```

Description

The **rpc_ss_disable_allocate()** routine releases (disables) all resources acquired by a call to **rpc_ss_enable_allocate()**, and any memory allocated by calls to **rpc_ss_allocate()** after the call to **rpc_ss_enable_allocate()** was made.

The **rpc_ss_enable_allocate()** and **rpc_ss_disable_allocate()** routines must be used in matching pairs.

For information about rules for using memory management routines, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_enable_allocate(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_enable_allocate

Purpose

Enables the allocation of memory by the **rpc_ss_allocate()** routine when not in manager code; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_enable_allocate(
        void);
```

Description

In sophisticated servers, it may be necessary to call manager code routines from different environments. This occurs, for example, when the application is both a client and a server of the same interface. Therefore, a manager code routine may need to be called both by the application code and by the stub code. If code, other than manager code, calls the **rpc_ss_allocate()** routine, it must first call **rpc_ss_enable_allocate()** to initialize the memory management environment that **rpc_ss_allocate()** uses.

For information about rules for using memory management routines, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_disable_allocate(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_free

Purpose

Frees memory allocated by the **rpc_ss_allocate()** routine; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_free(
        idl_void_p_t node_to_free);
```

Parameters

Input

node_to_free Specifies a pointer to memory allocated by **rpc_ss_allocate()**.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

The **rpc_ss_free()** routine releases memory allocated by **rpc_ss_allocate()**. The thread calling **rpc_ss_free()** must have the same thread handle as the thread that allocated the memory with **rpc_ss_allocate()**. Use it only in an environment where **rpc_ss_allocate()** is used.

If the manager code allocates memory with **rpc_ss_allocate()** and the memory is not released by **rpc_ss_free()** during manager code execution, then the server stub automatically releases the memory when the manager code completes execution and returns control to the stub.

Manager code can also use **rpc_ss_free()** to release memory that is pointed to by a full pointer in an input parameter.

For information about rules for using memory management routines, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_get_thread_handle

Purpose

Gets a thread handle for the manager code before it spawns additional threads, or for the client code when it becomes a server; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

rpc_ss_thread_handle_t rpc_ss_get_thread_handle(
        void);
```

Description

The **rpc_ss_get_thread_handle()** routine is used by a server manager thread when it spawns additional threads. To spawn additional threads that are able to perform memory management, the server manager code calls **rpc_ss_get_thread_handle()** and passes the thread handle to each spawned thread. Each spawned thread that uses **rpc_ss_allocate()** and **rpc_ss_free()** for memory management must first call **rpc_ss_set_thread_handle()**, using the handle obtained by the original manager thread.

The **rpc_ss_get_thread_handle()** routine can also be used when a program changes from being a client to being a server. The program gets a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**. When the program reverts to being a client it re–establishes the client environment by calling **rpc_ss_set_thread_handle()**, supplying the previously obtained handle as a parameter.

Return Values

A thread handle.

Examples

This function determines the thread handle, creates a thread, and passes the thread handle to the thread so it can share the memory management environment of the calling thread.

```
#include <pthread.h>
#include <idlbase.h>

pthread_t Launch_thread(
    int (*routine_to_launch)(
       pthread_addr_t th
       )
    )
{
    rpc_ss_thread_handle_t th = rpc_ss_get_thread_handle();
    pthread_t t;

    /*
     * Create the thread and pass to it the thread handle
     * so it can use rpc_ss_set_thread_handle.
     */
    pthread_create (&t, pthread_attr_default,
        (pthread_startroutine_t)routine_to_launch,
        (pthread_addr_t)th);

    return t;
}
```

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

rpc_ss_set_client_alloc_free

Purpose

Sets the memory allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage memory for pointed-to nodes; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_set_client_alloc_free (
     idl_void_p_t (*p_allocate) (
     unsigned long size),
     void (*p_free) (
     idl_void_p_t *ptr)
     );
```

Parameters

Input

p_allocate Specifies a pointer to a routine that has the same procedure declaration as the **malloc()** routine and that is used by the client stub to allocate memory.

p_free Specifies a pointer to a routine that has the same procedure declaration as the **free()** routine and that is used to free memory that was allocated using the routine pointed at by *p_allocate*.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

rpc_ss_set_client_alloc_free(3rpc)

Description

The **rpc_ss_set_client_alloc_free()** routine overrides the default routines that the client stub uses to manage memory for pointed-to nodes. The default memory management routines are **malloc()** and **free()**, except when the remote call occurs within manager code, in which case the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

For information about rules for using memory management routines, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_set_thread_handle

Purpose

Sets the thread handle for either a newly created spawned thread or for a server that was formerly a client and is ready to be a client again; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_set_thread_handle(
        rpc_ss_thread_handle_t id);
```

Parameters

Input

id A thread handle returned by a call to **rpc_ss_get_thread_handle()**.

Description

The **rpc_ss_set_thread_handle()** routine is used by a thread spawned in the manager code to associate itself with the main RPC manager thread. Each spawned thread that uses **rpc_ss_allocate()** and **rpc_ss_free()** for memory management must call **rpc_ss_set_thread_handle()**, using the handle that the main RPC manager thread obtained through **rpc_ss_get_thread_handle()**.

The **rpc_ss_set_thread_handle()** routine can also be used by a program that originally was a client, then became a server, and is now reverting to a client. The program must re−establish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the handle it received (through **rpc_ss_get_thread_handle()**) prior to becoming a server, as a parameter.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Examples

When this function is invoked within a spawned thread, its argument is the thread handle of the calling thread. This example assumes the data passed to the thread consists of only the middle thread.

```
#include <pthread.h>
#include <dce/idlbase.h>

 int helper_thread (
     pthread_addr_t th
     )
 {
     /*
      * Set the memory management environment to match
      * the parent environment.
      */
      rpc_ss_set_thread_handle(rpc_ss_thread_handle_t)th;
     /*
      * Real work of this thread follows here ...
      */
  }
```

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ss_swap_client_alloc_free

Purpose

Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_swap_client_alloc_free(
    idl_void_p_t (*p_allocate) (
    idl_size_t size),
    void (*p_free) (
    idl_void_p_t ptr),
    idl_void_p_t (**p_p_old_allocate) (
    idl_size_t size),
    void (**p_p_old_free) (
    idl_void_p_t ptr)
    );
```

Parameters

Input

p_allocate Specifies a pointer to a routine that has the same procedure declaration as the **malloc()** routine and that is used for allocating client stub memory.

p_free Specifies a pointer to a routine that has the same procedure declaration as the **free()** routine and that is used for freeing client stub memory.

Output

p_p_old_allocate
: Specifies a pointer to a pointer to a routine that has the same procedure declaration as the **malloc()** routine. A pointer to the routine that was previously used to allocate client stub memory is returned in this parameter.

p_p_old_free
: Specifies a pointer to a pointer to a routine that has the same procedure declaration as the **free()** routine. A pointer to the routine that was previously used to free client stub memory is returned in this parameter.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

The **rpc_ss_swap_client_alloc_free()** routine exchanges the current client allocate and free mechanism used by the client stubs for one supplied by the caller. If it is appropriate for the client code called by an application to use a certain memory allocation and freeing mechanism, regardless of its caller's state, the client code can swap its own mechanism into place on entry, replacing its caller's mechanism. It can then swap the caller's mechanism back into place prior to returning.

For information about rules for using memory management routines, see the *DCE 1.2.2 Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, is returned when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_set_client_alloc_free(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_string_binding_compose

Purpose

Combines the components of a string binding into a string binding; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_string_binding_compose(
        unsigned_char_t *obj_uuid,
        unsigned_char_t *protseq,
        unsigned_char_t *network_addr,
        unsigned_char_t *endpoint,
        unsigned_char_t *options,
        unsigned_char_t **string_binding,
        unsigned32 *status);
```

Parameters

Input

obj_uuid Specifies a NULL-terminated string representation of an object UUID.

protseq Specifies a NULL-terminated string representation of a protocol sequence.

network_addr
Specifies a NULL-terminated string representation of a network address.

endpoint Specifies a NULL-terminated string representation of an endpoint.

options Specifies a NULL-terminated string representation of network options.

Output

string_binding
Returns a pointer to a NULL-terminated string representation of a binding handle.

Specify NULL to prevent the routine from returning this argument. In this case the application does not call **rpc_string_free()**.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_binding_compose()** routine combines string binding handle components into a string binding handle.

The RPC runtime allocates memory for the string returned in the *string_binding* parameter. The application calls **rpc_string_free()** to deallocate that memory.

Specify NULL or provide a null string (**\0**) for each input string that has no data.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_string_binding_parse(3rpc)**, **rpc_string_free(3rpc)**, **uuid_to_string(3rpc)**.

rpc_string_binding_parse

Purpose

Returns, as separate strings, the components of a string binding; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_string_binding_parse(
        unsigned_char_t *string_binding,
        unsigned_char_t **obj_uuid,
        unsigned_char_t **protseq,
        unsigned_char_t **network_addr,
        unsigned_char_t **endpoint,
        unsigned_char_t **network_options,
        unsigned32 *status);
```

Parameters

Input

string_binding
: Specifies a NULL-terminated string representation of a binding.

Output

obj_uuid
: Returns a pointer to a NULL-terminated string representation of an object UUID.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

protseq
: Returns a pointer to a NULL-terminated string representation of a protocol sequence.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

network_addr
: Returns a pointer to a NULL-terminated string representation of a network address.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

endpoint
: Returns a pointer to a NULL-terminated string representation of an endpoint.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

network_options
: Returns a pointer to a NULL-terminated string representation of network options.

  Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_binding_parse()** routine parses a string representation of a binding handle into its component fields.

The RPC runtime allocates memory for each component string the routine returns. The application calls **rpc_string_free()** once for each returned string to deallocate the memory for that string.

If any field of the *string_binding* field is empty, **rpc_string_binding_parse()** returns the empty string in the corresponding output parameter.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_string_binding
Invalid string binding.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_string_binding_compose(3rpc)**, **rpc_string_free(3rpc)**, **uuid_from_string(3rpc)**.

rpc_string_free

Purpose

Frees a character string allocated by the runtime; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void rpc_string_free(
        unsigned_char_t **string,
        unsigned32 *status);
```

Parameters

Input/Output

string Specifies the address of the pointer to the character string to free.

The value NULL is returned.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_free()** routine deallocates the memory occupied by a character string returned by the RPC runtime.

An application must call this routine once for each character string allocated and returned by calls to other RPC runtime routines. The names of these routines appear at the end of this reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

Related Information

Functions: **dce_error_inq_text(3rpc)**, **rpc_binding_inq_auth_client(3rpc)**, **rpc_binding_inq_auth_info(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_mgmt_ep_elt_inq_next(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**, **rpc_ns_binding_inq_entry_name(3rpc)**, **rpc_ns_entry_expand_name(3rpc)**, **rpc_ns_group_mbr_inq_next(3rpc)**, **rpc_ns_profile_elt_inq_next(3rpc)**, **rpc_string_binding_compose(3rpc)**, **rpc_string_binding_parse(3rpc)**, **uuid_to_string(3rpc)**.

rpc_tower_to_binding

Purpose

Returns a binding handle from a tower representation

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_to_binding(
        byte_p_t prot_tower,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

Parameters

Input

prot_tower Specifies a single protocol tower to convert to a binding handle.

Output

binding Returns the server binding handle.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_tower_to_binding()** routine creates a server binding handle a canonical representation of a protocol tower.

When an application finishes using the *binding* parameter, the application calls the **rpc_binding_free()** routine to release the memory used by the binding handle.

The **rpc_intro(3rpc)** reference page contains an explanation of binding handles.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_vector_free(3rpc)**, **rpc_tower_vector_from_binding(3rpc)**.

rpc_tower_vector_free

Purpose

Releases memory associated with a tower vector

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_vector_free(
        rpc_tower_vector_p_t *twr_vector,
        unsigned32 *status);
```

Parameters

Input

twr_vector Specifies the tower vector to be freed. On return, its value is NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The status code is either **rpc_s_ok** or a value returned from a called routine.

Description

The **rpc_tower_vector_free()** routine releases memory associated with a tower vector, including the towers as well as the vector.

Return Values

No value is returned.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_to_binding(3rpc)**, **rpc_tower_vector_from_binding(3rpc)**.

rpc_tower_vector_from_binding

Purpose

Creates a tower vector from a binding handle

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_vector_from_binding(
        rpc_if_handle_t if_spec,
        rpc_binding_handle_t binding,
        rpc_tower_vector_p_t *twr_vector,
        unsigned32 *status);
```

Parameters

Input

if_spec The interface specification that will be combined with a binding handle to form a tower vector.

binding The binding handle that will be combined with a interface specification to form a tower vector.

Output

twr_vector Returns the allocated tower vector.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The status code is either **rpc_s_ok**, or **rpc_s_no_interfaces**, or a value returned from a called routine.

Description

The **rpc_tower_vector_from_binding()** routine creates a vector of towers from a binding handle. After the caller is finished with the tower vector, the

rpc_tower_vector_free() routine must be called to release the memory used by the vector.

Return Values

No value is returned.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_to_binding(3rpc)**, **rpc_tower_vector_free(3rpc)**.

uuid_compare

Purpose

Compares two UUIDs and determines their order; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

signed32 uuid_compare(
        uuid_t *uuid1,
        uuid_t *uuid2,
        unsigned32 *status);
```

Parameters

Input

uuid1 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid2*.

Use the value NULL to specify a nil UUID for this parameter.

uuid2 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid1*.

Use the value NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_compare()** routine compares two UUIDs and determines their order. A nil UUID is considered the first element in order. The order of UUIDs is defined by the RPC architecture and is not a temporal (related to time) ordering. Comparing two

specific UUIDs always returns the same result regardless of the implementation or system architecture.

You can use this routine to sort data with UUIDs as a key.

Return Values

Returns one of the following constants:

–1	The *uuid1* parameter precedes the *uuid2* parameter in order.
0	The *uuid1* parameter is equal to the *uuid2* parameter in order.
1	The *uuid1* parameter follows the *uuid2* parameter in order.

Note that a value of 0 (zero) has the same meaning as if **uuid_equal**(*&uuid1*, *&uuid2*) returned a value of TRUE.

A nil UUID is the first UUID in order. This means the following:

- If *uuid1* is NULL and *uuid2* is nonnil, the routine returns -1.
- If *uuid1* is NULL and *uuid2* is NULL, the routine returns 0.
- If *uuid1* is nonnil and *uuid2* is NULL, the routine returns 1.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_equal(3rpc)**, **uuid_is_nil(3rpc)**.

uuid_create

Purpose

Creates a new UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_create(
        uuid_t *uuid,
        unsigned32 *status);
```

Parameters

Input

None.

Output

uuid Returns the new UUID.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_create()** routine creates a new UUID.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_getconf_failure
Cannot get network interface device configuration.

uuid_s_no_address
Cannot get Ethernet hardware address.

uuid_s_socket_failure
Cannot create socket.

Related Information

Functions: **uuid_create_nil(3rpc)**, **uuid_from_string(3rpc)**, **uuid_to_string(3rpc)**.

uuid_create_nil

Purpose

Creates a nil UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_create_nil(
        uuid_t *nil_uuid,
        unsigned32 *status);
```

Parameters

Input

None.

Output

nil_uuid Returns a nil UUID.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_create_nil()** routine creates a nil UUID.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

Related Information

Functions: **uuid_create(3rpc)**.

uuid_equal

Purpose

Determines if two UUIDs are equal; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

boolean32 uuid_equal(
        uuid_t *uuid1,
        uuid_t *uuid2,
        unsigned32 *status);
```

Parameters

Input

uuid1 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid2*. Supply the value NULL to specify a nil UUID for this parameter.

uuid2 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid1*. Supply the value NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_equal()** routine compares two UUIDs and determines if they are equal.

Return Values

The possible return values and their meanings are as follows:

TRUE — The *uuid1* parameter is equal to the *uuid2* parameter. Parameter *status* contains the status code **uuid_s_ok**.

FALSE — The *uuid1* parameter is not equal to the *uuid2* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_compare(3rpc)**.

uuid_from_string

Purpose

Converts a string UUID to its binary representation; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_from_string(
        unsigned_char_t *string_uuid,
        uuid_t *uuid,
        unsigned32 *status);
```

Parameters

Input

string_uuid Specifies a string representation of a UUID. Supply the value NULL or the null string (**\0**) to specify a nil UUID.

Output

uuid Returns the binary form of the UUID specified by the *string_uuid* parameter into the address specified by this parameter.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application calls the **uuid_from_string()** routine to convert a string UUID to its binary representation.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

uuid_s_invalid_string_uuid
Invalid format for a string UUID.

Related Information

Functions: **uuid_to_string(3rpc)**.

uuid_hash

Purpose

Creates a hash value for a UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

unsigned16 uuid_hash(
        uuid_t *uuid,
        unsigned32 *status);
```

Parameters

Input

uuid Specifies the UUID for which a hash value is created. Supply NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_hash()** routine generates a hash value for a specified UUID.

Note that the return value for a single *uuid* value may differ across platforms.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

Return Values

Returns a hash value for the specified UUID.

uuid_is_nil

Purpose

Determines if a UUID is nil; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

boolean32 uuid_is_nil(
        uuid_t *uuid,
        unsigned32 *status);
```

Parameters

Input

uuid Specifies a UUID to test as a nil UUID. Supply NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_is_nil()** routine determines whether the specified UUID is a nil UUID. This routine yields the same result as if an application did the following:

- Called the **uuid_create_nil()** routine.
- Called the **uuid_equal()** routine to compare the returned nil UUID to the UUID specified in the *uuid* parameter.

Return Values

The possible return values and their meanings are as follows:

TRUE — The *uuid* parameter is a nil UUID. Parameter *status* contains the status code **uuid_s_ok**.

FALSE — The *uuid* parameter is not a nil UUID.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_compare(3rpc)**, **uuid_create_nil(3rpc)**, **uuid_equal(3rpc)**.

uuid_to_string

Purpose

Converts a UUID from a binary representation to a string representation; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_to_string(
        uuid_t *uuid,
        unsigned_char_t **string_uuid,
        unsigned32 *status);
```

Parameters

Input

uuid Specifies a UUID in its binary format. Supply NULL to specify a nil UUID for this parameter.

Output

string_uuid Returns a pointer to the string representation of the UUID specified in the *uuid* parameter. Specify NULL for this parameter to prevent the routine from returning this information.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_to_string()** routine converts a UUID from its binary representation to its string representation.

The RPC runtime allocates memory for the string returned in the *string_uuid* parameter. The application calls **rpc_string_free()** to deallocate that memory. It is

not necessary to call **rpc_string_free()** when you supply NULL for the *string_uuid* parameter.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_string_free(3rpc)**, **uuid_from_string(3rpc)**.

wchar_t_from_netcs

Purpose

Converts international character data from a network code set to a local code set prior to unmarshalling; used by client and server applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_from_netcs(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        idl_byte *network_data,
        unsigned32 network_data_length,
        unsigned32 local_buffer_size,
        wchar_t *local_data,
        unsigned32 *local_data_length,
        error_status_t *status);
```

Parameters

Input

binding
Specifies the target binding handle from which to obtain code set conversion information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next( )** or **rpc_ns_binding_select( )**routine.

network_code_set_value
The registered hexadecimal integer value that represents the code set that was used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the receiving tag. When the caller is the server stub, this value is the sending tag.

network_data
A pointer to the international character data that has been received, in the network code set encoding.

network_data_length
The number of **idl_byte** data elements to be converted. For a varying array or a conformant varyihg array, the value is the local value of the **length_is** variable. For a conformant array, the value is the local value of the **size_is** variable. For a fixed array, the value is the array size specified in the interface definition.

local_buffer_size
A pointer to the buffer size to be allocated to contain the converted data, in units of **wchar_t**. The value specified in this parameter is the local buffer size returned by the **wchar_t_local_size( )** routine.

Output

local_data A pointer to the converted data, in **wchar_t** format.

local_data_length
The length of the converted data, in units of **wchar_t**. NULL is specified if a fixed array or varying array is to be converted.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_from_netcs()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_from_netcs()** routine is one of the DCE RPC stub code set conversion routines that RPC stubs use before they marshall or unmarshall data to convert international character data to and from local and network code sets.

Client and server stubs call the **wchar_t_*_netcs** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application.

Client and server stubs call the **wchar_t_from_netcs()** routine before they unmarshall the international character data received from the network. The routine takes a binding handle, a code set value that identifies the code set used to transfer international

character data over the network, the address of the network data, in **idl_byte** format, that may need to be converted, and the data length, in units of **idl_byte**.

The routine compares the sending code set to the local code set currently in use. If the routine finds that code set conversion is necessary, (because the local code set differs from the code set specified to be used on the network), it determines which host code set converter to call to convert the data and then invokes that converter.

The routine then returns the converted data, in **wchar_t** format. If the data is a conformant or conformant varying array, the routine also returns the length of the converted data, in units of **wchar_t**.

Prior to calling **wchar_t_from_netcs()**, client and server stubs call the **wchar_t_local_size()** routine to calculate the size of the buffer required to hold the converted data. Because **wchar_t_local_size()** cannot make this calculation for fixed and varying arrays, applications should either restrict use of **wchar_t_from_netcs()** to conformant and conformant varying arrays, or independently ensure that the buffer allocated for converted data is large enough.

Applications can specify local data types other than **cs_byte** and **wchar_t** (the local data types for which DCE RPC supplies stub code set conversion routines) with the **cs_char** ACF attribute. In this case, the application must also supply *local_type*_**to_netcs()** and *local_type*_**from_netcs()** stub conversion routines for this type.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain code set evaluation information. If this error occurs in the server stub, an exception is raised to the client application.

When the routine is running the host converter routines, the following errors can be returned:

- **rpc_s_ss_invalid_char_support**
- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, this routine calls the **dce_cs_loc_to_rgy()** routine and the host converter routines. If one of these routines returns an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_local_size

Purpose

Calculates the necessary buffer size for code set conversion from a network code set to a local code set prior to unmarshalling; used by client and server stubs, but not directly by applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_local_size(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        unsigned32 network_buffer_size,
        idl_cs_convert_t *conversion_type,
        unsigned32 *local_buffer_size,
        error_status_t *status);
```

Parameters

Input

binding Specifies the target binding handle from which to obtain buffer size evaluation information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()** routine.

network_code_set_value
The registered hexadecimal integer value that represents the code set used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the receiving tag. When the caller is the server stub, this value is the sending tag.

network_buffer_size
: The size, in units of **idl_byte**, of the buffer that is allocated for the international character data, For a conformant or conformant varying array, this value is the network value of the **size_is** variable for the array; that is, the value is the size of the unmarshalled string if no conversion is done.

Output

conversion_type
: A pointer to the enumerated type defined in **dce/idlbase.h** that indicates whether data conversion is necessary and whether or not the existing buffer is sufficient for storing the results of the conversion. Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, and **idl_byte** to **wchar_t** conversion always takes place, the conversion type returned is always **idl_cs_new_buffer_convert**.

local_buffer_size
: A pointer to the buffer size that needs to be allocated to contain the converted data, in units of **wchar_t**. This value is to be used as the local value of the **size_is** variable for the array, and is nonNULL only if a conformant or conformant varying array is to be unmarshalled. A value of NULL in this parameter indicates that a fixed or varying array is to be unmarshalled.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_local_size()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_local_size()** routine is one of the four DCE RPC buffer sizing routines that RPC stubs use before they marshall or unmarshall data to determine whether or not the buffers allocated for code set conversion need to be enlarged to hold the converted data. The buffer sizing routines determine the type of conversion required and calculate the size of the necessary buffer (if a conformant or conformant varying array is to be marshalled or unmarshalled); the RPC stub then allocates a buffer of that size before it calls one of the code set conversion routines.

Client and server stubs call the two **wchar_t_*_size** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **wchar_t_local_size()** routine is used to evaluate buffer size requirements prior to unmarshalling data received over the network.

Applications do not call the **wchar_t_local_size()** routine directly. Client and server stubs call the routine before they unmarshall any data. The stubs pass the routine a binding handle and a code set value that identifies the code set that was used to transfer international character data over the network. The stubs also specify the network storage size of the data, in units of **idl_byte**.

Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, **wchar_t_local_size()** always sets *conversion_type* to **idl_cs_new_buffer_convert**, regardless of whether it is called from a client or server stub, or whether client and server code set tag information has been stored in the binding handle by a code sets evaluation or tag-setting routine. If a conformant or conformant varying array is to be unmarshalled, the routine then calculates a new buffer size by dividing the value of *network_buffer_size* by the number of bytes required to encode one **wchar_t** unit. The routine returns the new buffer size in the *local_buffer_size* argument. The size is specified in units of **wchar_t**, which is the local representation used for international character data in wide character format.

When a fixed or varying array is being unmarshalled, the **wchar_t_local_size()** routine cannot calculate the required buffer size and does not return a value in the *local_buffer_size* argument.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain the information necessary to evaluate the code set. If this error occurs in the server stub, an exception is raised to the client application.

When invoked from the server stub, this routine calls the routines **dce_cs_loc_to_rgy()** and **rpc_rgy_get_max_bytes()**. If either of these routines returns an error, the **wchar_t_local_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_net_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_net_size

Purpose

Calculates the necessary buffer size for code set conversion from a local code set to a network code set prior to marshalling; used by client and server stubs but not directly by applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_net_size(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        unsigned32 local_buffer_size,
        idl_cs_convert_t *conversion_type,
        unsigned32 *network_buffer_size,
        error_status_t *status);
```

Parameters

Input

binding Specifies the target binding handle from which to obtain buffer size evaluation information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()** routine.

network_code_set_value
The registered hexadecimal integer value that represents the code set to be used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the sending tag. When the caller is the server stub, this value is the receiving tag.

local_buffer_size
: The size, in units of **wchar_t**, of the buffer that is allocated for the international character data. For a conformant or conformant varying array, this value is the local value of the **size_is** variable for the array; that is, the value is the size of the marshalled string if no conversion is done.

Output

conversion_type
: A pointer to the enumerated type defined in **dce/idlbase.h** that indicates whether data conversion is necessary and whether or not the existing buffer is sufficient for storing the results of the conversion. Because **wchar_t** to **idl_byte** require different numbers of bytes to encode one character, and **wchar_t** to **idl_byte** conversion always takes place, the conversion type returned is always **idl_cs_new_buffer_convert**.

network_buffer_size
: A pointer to the buffer size that needs to be allocated to contain the converted data, in units of **idl_byte**. This value is to be used as the network value of the **size_is** variable for the array, and is non-NULL only if a conformant or conformant varying array is to be marshalled. A value of NULL in this parameter indicates that a fixed or varying array is to be marshalled.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_net_size()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_net_size()** routine is one of the four DCE RPC buffer sizing routines that RPC stubs use before they marshall or unmarshall data to determine whether or not the buffers allocated for code set conversion need to be enlarged to hold the converted data. The buffer sizing routines determine the type of conversion required and calculate the size of the necessary buffer (if a conformant or conformant varying array is to be marshalled or unmarshalled); the RPC stub then allocates a buffer of that size before it calls one of the code set conversion routines.

Client and server stubs call the two **wchar_t_*_size** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **wchar_t_net_size()** routine is used to evaluate buffer size requirements prior to marshalling data to be sent over the network.

Applications do not call the **wchar_t_net_size()** routine directly. Client and server stubs call the routine before they marshall any data. The stubs pass the routine a binding handle and a code set value that identifies the code set to be used to transfer international character data over the network. The stubs also specify the local storage size of the data, in units of **wchar_t**.

Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, **wchar_t_net_size()** always sets *conversion_type* to **idl_cs_new_buffer_convert**, regardless of whether it is called from a client or server stub, or whether client and server code set tag information has been stored in the binding handle by a code sets evaluation or tag-setting routine. If a conformant or conformant varying array is to be marshalled, the routine then calculates a new buffer size by multiplying the value of *local_buffer_size* by the number of bytes required to encode one **wchar_t** unit. The routine returns the new buffer size in the *network_buffer_size* argument. The size is specified in units of **idl_byte**, which is the network representation used for international character data.

When a fixed or varying array is being marshalled, the **wchar_t_net_size()** routine cannot calculate the required buffer size and does not return a value in the *network_buffer_size* argument.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain the information necessary to evaluate the code set. If this error occurs in the server stub, an exception is raised to the client application.

When invoked from the server stub, this routine calls the routines **dcs_cs_loc_to_rgy()** and **rpc_rgy_get_max_bytes()**. If either of these routines returns an error, the **wchar_t_net_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **dcs_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_to_netcs

Purpose

Converts international character data from a local code set to a network code set prior to marshalling; used by client and server applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_to_netcs(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        wchar_t *local_data,
        unsigned32 local_data_length,
        idl_byte *network_data,
        unsigned32 *network_data_length,
        error_status_t *status);
```

Parameters

Input

binding Specifies the target binding handle from which to obtain code set conversion information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next( )** or **rpc_ns_binding_select( )** routine.

network_code_set_value
The registered hexadecimal integer value that represents the code set to be used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the sending tag. When the caller is the server stub, this value is the receiving tag.

local_data A pointer to the international character data to be transmitted, in the local code set encoding.

local_data_length
: The number of **wchar_t** data elements to be converted. For a varying array or a conformant varying array, this value is the local value of the **length_is** variable. For a conformant array, this value is the local value of the **size_is** variable. For a fixed array, the value is the array size specified in the interface definition.

Output

network_data
: A pointer to the converted data, in **idl_byte** format.

network_data_length
: A pointer to the length of the converted data, in units of **idl_byte**. NULL is specified if a fixed or varying array is to be converted.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_to_netcs()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_to_netcs()** routine is one of the DCE RPC stub code set conversion routines that RPC stubs use before they marshall or unmarshall data to convert international character data to and from local and network code sets.

Client and server stubs call the **wchar_t_*_netcs()** routines when the **wchar_t** type has been specified as the local data type with the **cs_char** attribute in the attribute configuration file for the application.

Client and server stubs call the **wchar_t_to_netcs()** routine before they marshall any data. The routine takes a binding handle, a code set value that identifies the code set to be used to transfer international character data over the network, the address of the data to be converted, and the length of the data, in units of **wchar_t**.

The routine first converts the character data from **wchar_t** values to **idl_byte** values. The routine next compares the sending code set to the local code set currently in use. If the routine finds that code set conversion is necessary, (because the local code set differs from the code set specified to be used on the network), it determines which host code set converter to call to convert the data and then invokes that converter.

The routine then returns the converted data, in **idl_byte** format. If the data is a conformant or conformant varying array, the routine also returns the length of the converted data, in units of **idl_byte**.

Prior to calling **wchar_t_to_netcs()**, client and server stubs call the **wchar_t_net_size()** routine to calculate the size of the buffer required to hold the converted data. Because **wchar_t_net_size()** cannot make this calculation for fixed and varying arrays, applications should either restrict use of **wchar_t_to_netcs()** to conformant and conformant varying arrays, or independently ensure that the buffer allocated for converted data is large enough.

Applications can specify local data types other than **cs_byte** and **wchar_t** (the local data types for which DCE RPC supplies stub support routines for code set conversion) with the **cs_char** ACF attribute. In this case, the application must also supply *local_type*_**to_netcs()** and *local_type*_**from_netcs()** stub conversion routines for the application-defined local type.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain code set evaluation information. If this error occurs in the server stub, an exception is raised to the client application.

When this routine is running the host converter routines, the following errors can be returned:

- **rpc_s_ss_invalid_char_input**
- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, this routine calls the **dce_cs_loc_to_rgy()** routine and host converter routines. If any of these routines returns an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**,

Chapter 4

DCE Directory Service

xds_intro

Purpose

Introduction to X/OPEN Directory Services (XDS) functions

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <xdsext.h>
```

Description

This **xds_intro** reference page lists the XDS interface functions in the following table. XDS provides a C language binding.

Service Interface Functions—xds_intro(3xds)	
Function	**Description**
dsX_extract_attr_values()	Extracts attribute values from an OM object.
ds_abandon()	Function not supported.
ds_add_entry()	Adds a leaf entry to the directory information tree (DIT).
ds_bind()	Opens a session with a directory user agent.
ds_compare()	Compares a purported attribute value with the attribute value stored in the directory for a particular entry.
ds_initialize()	Initializes the interface.
ds_list()	Enumerates the immediate subordinates of a particular directory entry.
ds_modify_entry()	Performs an atomic modification of a directory entry.

ds_modify_rdn()	Changes the relative distinguished name (RDN) of a leaf entry.
ds_read()	Queries information on a directory entry by name.
ds_receive_result()	Function partially supported.
ds_remove_entry()	Removes a leaf entry from the DIT.
ds_search()	Finds entries of interest in a portion of the DIT.
ds_shutdown()	Shuts down the interface.
ds_unbind()	Unbinds from a directory session.
ds_version()	Negotiates features of the interface and service.
gds_decode_alt_addr()	Used by DME applications for alternate address mapping.
gds_encode_alt_addr()	Used by DME applications for alternate address mapping.

The Distributed Computing Environment (DCE) XDS interface does not support asynchronous operations within the same thread. Thus, **ds_abandon()** is redundant. A **ds_abandon()** call returns with a **DS_C_ABANDON_FAILED** (**DS_E_TOO_LATE**) error. For **ds_receive_result()**, if there are any outstanding operations (when multiple threads issue XDS calls in parallel), this function returns **DS_SUCCESS** with the *completion_flag_return* parameter set to **DS_OUTSTANDING_OPERATIONS**.

If no XDS calls are outstanding, **ds_receive_result()** returns with *DS_status* set to **DS_SUCCESS**, and with the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION**.

The following differences exist between Global Directory Service (GDS) and Cell Directory Service (CDS):

- All functions operate on the GDS namespace.
- CDS does not support the **ds_modify_rdn()** or **ds_search()**. If either of these two functions is attempted on CDS, the error message **DS_C_SERVICE_ERROR** is returned (**DS_E_UNWILLING_TO_PERFORM**).
- In CDS, no X.500 schema rules apply. There is

— No concept of an object class.

— No mandatory attributes for a given object.

— No set of attributes expressly permitted for a given object.

— No predefined definition of single and multivalued attributes.

The absence of these schema rules means that the usual errors, which are returned by GDS for breach of schema rules, are not returned by CDS.

The CDS naming DIT is modeled on a typical file system architecture, where directories are used for storing objects and directories can contain subdirectories. Leaf objects in the CDS DIT are similar to X.500 naming objects. However, subtree objects are called directories as in a file system directory. All new objects must be added to an existing directory. CDS directory objects cannot be added, removed, modified, or compared using the XDS programming interface.

In CDS, the naming attribute of an object is not stored in the object. Consequently, in CDS, **ds_read()** never returns this attribute. Note that the **ds_compare()** routine applied to this attribute returns with **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).

Notes

See the notes in the relevant reference page for function-specific differences.

XDS functions check for NULL pointers and will return an error. The pointers are only checked at the function interface. The check is only for NULL and not for validity. If NULL pointers are passed, this may result in an undetermined behavior.

decode_alt_addr

Purpose

Converts an alternate address attribute from internal GDS format to a structured format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

int decode_alt_addr(
        const D2_str *in,
        D2_alt_addr **out);
```

Parameters

in A pointer to a **D2_str** structure that contains the alternate address attribute in an internal GDS format.

Description

The **decode_alt_addr()** routine converts a linearized string that is stored in a structure **D2_str** into a structured alternate address format stored in a **D2_alt_addr** structure. This function is provided for use by DME applications. It converts an alternate address attribute from an internal GDS format (linear octet string) to a structured format for application usage.

in->d2_size contains the length of the encoded octet string.

in->d2_value is a pointer to the beginning of the encoded octet string.

The **decode_alt_addr()** routine allocates memory for the structured alternate address. The parameter (**out*) contains the address of the memory area that should later be freed by the application.

The **D2_alt_addr** structure contains one field **D2_str** for the address, followed by a structured field for the set of object identifiers. The structure **D2_str** consists of

the length of the address and a pointer to the beginning of the address (not zero-terminated). The second component of the **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To read a set of object identifiers, the address of the first **D2_obj_id** structure should be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be read.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means that **decode_alt_addr()** does no BER conversion for object identifiers.

Return Values

***out*	A pointer to the structure **D2_alt_addr** that stores the alternate address attribute in a structured format.
int	0 if successful.
	-1 if unsuccessful (**malloc()** failure).

Related Information

Functions: **encode_alt_addr(3xds)**.

dsX_extract_attr_values

Purpose

Extracts attribute values from an OM object

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <xdsext.h>

OM_return_code dsX_extract_attr_values(
        OM_private_object object,
        OM_object_identifier attribute_type,
        OM_boolean local_strings,
        OM_public_object *values,
        OM_value_position *total_number);
```

Parameters

Input

object The private object from which the attribute values are to be extracted. Objects of type **DS_C_ATTRIBUTE_LIST** or **DS_C_ENTRY_INFO** are supported.

attribute_type
The attribute type from which the values are to extracted.

local_strings Indicates if results should be converted to a local string format.

Output

values The *values* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS**. It points to a public object containing an array of OM descriptors with the extracted attribute values.

total_number
Contains the total number of attribute values that have been extracted.

Note that the total includes only the attribute descriptors in the *values* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **dsX_extract_attr_values()** routine is used to extract the attribute values associated with the specified attribute type from an OM object. The OM object must be of type **DS_C_ATTRIBUTE_LIST** or **DS_C_ENTRY_INFO**. It returns an object containing an array of OM descriptors.

Notes

The memory space for the *values* return parameter is allocated by **dsX_extract_attr_values()**. The calling application is responsible for releasing this memory with the **om_delete()** routine.

Return Values

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of possible error values that can be returned in *OM_return_code*. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

ds_add_entry

Purpose

Adds a leaf entry to the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_add_entry(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_object entry,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant. Note that **DS_DONT_DEREFERENCE_ALIASES** and **DS_SIZE_LIMIT** do not apply to this operation.

name (Object(**DS_C_NAME**)). The name of the entry to be added. The immediate superior of the new entry is determined by removing the last RDN component, which belongs to the new entry.

The immediate superior must exist in the same Directory Service Agent, or the function can fail with **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**). Any aliases in the name are *not* dereferenced.

entry (Object(**DS_C_ATTRIBUTE_LIST**)). The attribute information that, together with that from the RDN, constitutes the entry to be created. Note that an instance of OM class **DS_C_ENTRY_INFO** can be supplied as the value of this parameter, since OM class **DS_C_ENTRY_INFO** is a subclass of OM class **DS_C_ATTRIBUTE_LIST**.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_add_entry()** function adds a leaf entry to the directory. The entry can be either an object or an alias. The directory checks that the resulting entry conforms to the directory schema.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_add_entry()** operation:

- Only leaf objects (that is, objects that are not CDS directory objects) can be added to CDS through the XDS interface. In other words, the immediate superior of the new entry must exist.
- Only the **DS_A_COMMON_NAME** and **DS_A_MEMBER** attributes are valid for the **DS_O_GROUP_OF_NAMES** object in CDS.
- GDS-structured attribute types are not supported by CDS. If an attempt is made to add a GDS-structured attribute type to CDS, then it returns with a **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).

Since CDS does not implement the X.500 schema rules, some CDS objects may not contain mandatory attributes like object class and so on.

Return Values

DS_status **DS_SUCCESS** is returned if the entry was added; otherwise, an error is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

The **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**) error, referred to earlier in this reference page, need not be returned if there is local agreement between the DSAs to allow the entry to be added.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_bind

Purpose

Opens a session with the directory

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_bind(
        OM_object session,
        OM_workspace workspace,
        OM_private_object *bound_session_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). Specifies a particular directory service provider, together with other details of the service required. This parameter can be either a public object or a private object. The **DS_DEFAULT_SESSION** constant can also be used as the value of this parameter, causing a new session to be created with default values for all its OM attributes.

workspace Specifies the workspace obtained from a call to **ds_initialize()** that is to be associated with the session. All function results from directory operations using this session will be returned as private objects in this workspace. If the *session* parameter is a private object, it must be a private object in this workspace.

Output

bound_session_return
(Object(**DS_C_SESSION**)). Upon successful completion, this parameter contains an instance of a directory session that can be used as a parameter to other functions (for example, **ds_read()**). This is a new private object if the value of the *session* parameter

was **DS_DEFAULT_SESSION** or a public object; otherwise, it is that value supplied as a parameter. The function supplies default values for any of the OM attributes that are not present in the *session* parameter instance supplied as a parameter. It also sets the value of the **DS_FILE_DESCRIPTOR** OM attribute to **DS_NO_VALID_FILE_DESCRIPTOR**, since the functionality is not supported.

Description

The **ds_bind()** function sets up a communications link to the DSA.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions at the start of this chapter for general differences between operations on GDS and CDS.)

Note that in order to use CDS when GDS is not active, **ds_bind()** must be called with the value of the *session* parameter set to **DS_DEFAULT_SESSION**.

Return Values

DS_status **DS_SUCCESS** is returned if the function is completed successfully; otherwise, it indicates the error that has occurred.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_SESSION**
- **DS_E_BAD_WORKSPACE**

- **DS_E_MISCELLANEOUS**
- **DS_E_NOT_SUPPORTED**
- **DS_E_TOO_MANY_SESSIONS**

The function can return the following directory errors:

- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

Related Information

Functions: **ds_unbind(3xds)**.

ds_compare

Purpose

Compares an attribute value with the attribute value stored in the directory for a particular entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_compare(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_object ava,
        OM_private_object *result_return,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** does not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are dereferenced unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

ava (Object(**DS_C_AVA**)). The attribute value assertion that specifies the attribute type and value to be compared with those in the entry.

Output

result_return (Object(**DS_C_COMPARE_RESULT**)). Upon successful completion, the result contains flags indicating whether the values matched and whether the comparison was made against the original entry. It also contains the DN of the target object if an alias is dereferenced.

invoke_id_return
(Integer). Not supported.

Description

The **ds_compare()** function compares the value supplied in the given *ava* parameter with the value or values of the same attribute type in the named entry.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_compare()** operation:

- In CDS, the naming attribute of an object is not stored in the attribute list of an object. Thus in CDS, a **ds_compare()** of the purported naming attribute value with the naming attribute value of the directory object always fails to match.
- GDS-structured types are not supported by CDS. If a GDS-structured attribute type is used as a parameter to **ds_compare()** on a CDS object, then it returns with the error **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).
- In CDS, **ds_compare()** can only be used on leaf objects; otherwise, a **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**) is returned.
- In CDS, if the *name* parameter is a CDS soft link and the **Dont_Dereference_Aliases** context parameter is set to **TRUE**, the only allowed attribute for comparison is the **DS_A_ALIASED_OBJECT_NAME** attribute. This attribute is compared with the Distinguished Name of the soft link target.

Return Values

DS_status Indicates whether the comparison is completed or not. If successful, **DS_SUCCESS** is returned. Note that the operation fails and an error is returned either if the target object is not found or if it does not have an attribute of the required type.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_initialize

Purpose

Initializes the XDS interface

Synopsis

```
#include <xom.h>
#include <xds.h>

OM_workspace ds_initialize(
        void);
```

Description

The **ds_initialize()** function performs any necessary initialization of the XDS application program interface (API), including the creation of a workspace. It must be called before any other directory interface functions are called. If it is subsequently called before **ds_shutdown()**, the function returns NULL.

Return Values

OM_workspace

Upon successful completion, **OM_workspace** contains a handle to a workspace in which OM objects can be created and manipulated. Objects created in this workspace, and only such objects, can be used as parameters to the other directory interface functions. This function returns NULL if it fails.

Related Information

Functions: **ds_shutdown(3xds)**.

ds_list

Purpose

Enumerates the immediate subordinates of a particular directory entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_list(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_private_object *result_return,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the object entry whose immediate subordinates are to be listed. Any aliases in the name are dereferenced unless this is prohibited by the service control attribute **DS_DONT_DEREFERENCE_ALIASES** of the **DS_C_CONTEXT** object.

Output

result_return (Object(**DS_C_LIST_RESULT**)). Upon successful completion, the result contains some information about the target object's immediate subordinates. It also contains the DN of the target object, if an alias

was dereferenced to find it. Aliases in the subordinate names are not dereferenced. In addition, there can be a partial outcome qualifier, which indicates that the result is incomplete. It also explains the reason for this (for example, because the time limit expired), and it contains information that can be helpful when attempting to complete the operation.

invoke_id_return
(Integer). Not supported.

Description

The **ds_list()** function is used to obtain a list of the immediate subordinates of the named entry. The list can be incomplete in some circumstances; for example, if the results exceed **DS_SIZE_LIMIT**.

Return Values

DS_status Takes the value **DS_SUCCESS** if the named object is located (even if there are no subordinates) and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_modify_entry

Purpose

Performs an atomic modification on a directory entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_modify_entry(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_object changes,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are *not* dereferenced.

changes (Object(**DS_C_ENTRY_MOD_LIST**)). A sequence of modifications to the named entry.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_modify_entry()** routine is used to make a series of one or more of the following changes to a single directory entry:

- Add a new attribute (**DS_ADD_ATTRIBUTE**).
- Remove an attribute (**DS_REMOVE_ATTRIBUTE**).
- Add attribute values (**DS_ADD_VALUES**).
- Remove attribute values (**DS_REMOVE_VALUES**).

Values can be replaced by a combination of adding values and removing values in a single operation. The RDN can only be changed by using **ds_modify_rdn()**.

The result of the operation is as if each modification is made in the order specified in the *changes* parameter. If any of the individual modifications fails, then a **DS_C_ATTRIBUTE_ERROR** is reported and the entry is left as it was prior to the whole operation. The operation is atomic; that is, either all or none of the changes are made. The directory checks that the resulting entry conforms to the directory schema.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_modify_entry()** operation:

- Naming schema rules do not apply in CDS. Thus, the following attribute errors are never returned by CDS:
  - **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**
  - **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**
- Naming operations that would normally return these errors succeed in CDS. In particular, the addition of an attribute that already exists does not return with an error. Instead, the values of the attribute to be added are combined with the values of the existing attribute.
- GDS-structured attribute types are not supported by CDS. If a GDS-structured attribute type is used as a parameter to **ds_modify_entry()**

on a CDS object, then it returns with a **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**). In CDS, **ds_modify_entry()** can only be used on leaf objects; otherwise, a **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**) is returned.

- In CDS, if the *name* parameter is a CDS soft link and the **Dont_Dereference_Alias** flag is set to **TRUE**, the soft link entry itself is modified. In this case, the only allowed modifications are to the **DS_A_ALIASED_OBJECT_NAME** attribute.

Return Values

DS_status Takes the value **DS_SUCCESS** if all the modifications succeeded and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**

- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

The following situations apply to GDS:

- An attempt to use **DS_ADD_ATTRIBUTE** to add an existing attribute results in a **DS_C_ATTRIBUTE_ERROR**.
- An attempt to use **DS_ADD_VALUES** to add an existing value results in a **DS_C_ATTRIBUTE_ERROR**, as does an attempt to add a value to a nonexistent attribute type.
- An attempt to use **DS_REMOVE_ATTRIBUTE** to remove a nonexisting attribute results in a **DS_C_ATTRIBUTE_ERROR**, whereas an attempt to remove an attribute that is part of the object's RDN results in a **DS_C_UPDATE_ERROR**.
- An attempt to use **DS_REMOVE_VALUES** to remove a nonexisting value results in a **DS_C_ATTRIBUTE_ERROR**, whereas an attempt to remove a value of an attribute that is part of the object's RDN, or to modify the object class attribute, results in a **DS_C_UPDATE_ERROR**.

ds_modify_rdn

Purpose

Changes the RDN of a leaf entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_modify_rdn(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_object new_RDN,
        OM_boolean delete_old_RDN,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The current name of the target leaf entry. Any aliases in the name are *not* dereferenced. The immediate superior must *not* have any nonspecific subordinate references; if it does, the function can fail with a **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**).

A nonspecific subordinate reference is an indication that another DSA holds some number of children, but does not indicate their RDNs. This

means that it is not possible to check the uniqueness of the requested new RDN within a single DSA.

new_RDN (Object(**DS_C_RELATIVE_NAME**)). The requested new RDN. If an attribute value in the new RDN does not already exist in the entry (either as part of the old RDN or as a nondistinguished value), the new value is added. If it cannot be added, an error is reported.

delete_old_RDN
(Boolean). If this value is **OM_TRUE**, all attribute values that are in the old RDN but not in the new RDN are deleted. If the value is **OM_FALSE**, the old values should remain in the entry (not as part of the RDN). The value must be **OM_TRUE** when a single value attribute in the RDN has its value changed by the operation. If this operation removes the last attribute value of an attribute, that attribute is deleted.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_modify_rdn()** function is used to change the RDN of a leaf entry (either an object entry or an alias entry).

Notes

CDS does not support **ds_modify_rdn()**, and returns with **DS_C_SERVICE_ERROR** (**DS_E_UNWILLING_TO_%PERFORM**).

Return Values

DS_status Indicates whether the name of the entry is changed (**DS_SUCCESS** is returned); otherwise, an error is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

The **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**) error, referred to earlier in this reference page, need not be returned if there is local agreement between the DSAs to allow the entry to be modified.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_read

Purpose

Queries information on an entry by name

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_read(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_object selection,
        OM_private_object *result_return,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** does not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are dereferenced unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

selection (Object(**DS_C_ENTRY_INFO_SELECTION**)). Specifies what information from the entry is requested. Information about no attributes, all attributes, or just for a named set can be chosen. Attribute types are always returned, but the attribute values need not be returned.

The possible values of this parameter are given in the *DCE 1.2.2 Application Development Guide—Directory Services*.

Output

result_return (Object(**DS_C_READ_RESULT**)). Upon successful completion, the result contains the DN of the target object, and a flag indicating whether the result came from the original entry or a copy, as well as any requested attribute types and values. Attribute information is only returned if access rights are sufficient.

invoke_id_return
(Integer). Not supported.

Description

The **ds_read()** function is used to extract information from an explicitly named entry. It can also be used to verify a DN.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_read()** operation:

- Since CDS does not implement the X.500 schema rules, some CDS objects may not contain mandatory attributes like object class and so on. In CDS, a read of an alias object fails if the **DS_A_ALIASED_OBJECT_NAME** does not exist. Instead, CDS returns with **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**).
- In CDS, the naming attribute of an object is not stored in the attribute list for the object. Thus in CDS, **ds_read()** does not return this attribute in the attribute list for an object.

Return Values

DS_status Indicates whether or not the read operation is completed. This is **DS_SUCCESS** if completed.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_ATTRIBUTE**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

Note that the directory error **DS_C_ATTRIBUTE_ERROR** (**DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**) is reported in GDS if an explicit list of attributes is specified by the *selection* parameter, but none of them are present in the entry. This error is not reported if any of the selected attributes are present.

A **DS_C_SECURITY_ERROR** (**DS_E_INSUFFICIENT_ACCESS_RIGHTS**) is only reported where access rights preclude the reading of all requested attribute values.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_remove_entry

Purpose

Removes a leaf entry from the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_remove_entry(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are *not* dereferenced.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_remove_entry()** function is used to remove a leaf entry from the directory (either an object entry or an alias entry).

Return Values

DS_status Indicates whether or not the entry was deleted.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_search

Purpose

Finds entries of interest in a part of the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_search(
        OM_private_object session,
        OM_private_object context,
        OM_object name,
        OM_sint subset,
        OM_object filter,
        OM_boolean search_aliases,
        OM_object selection,
        OM_private_object *result_return,
        OM_sint *invoke_id_return);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context (Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the object entry that forms the base of **ds_search()**. Any aliases in the name are dereferenced, unless dereferencing is prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

subset (Integer). Specifies the portion of the DIT to be searched. Its value must be one of the following:

- **DS_BASE_OBJECT** Searches just the given object entry.
- **DS_ONE_LEVEL** Searches just the immediate subordinates of the given object entry.
- **DS_WHOLE_SUBTREE** Searches the given object and all its subordinates.

filter (Object(**DS_C_FILTER**)). The filter is used to eliminate entries from the search that are not wanted. Information is only returned on entries that satisfy the filter. The **DS_NO_FILTER** constant can be used as the value of this parameter if all entries are searched and none eliminated. This corresponds to a filter with a **DS_FILTER_TYPE** value of **DS_AND** and no values of the **DS_FILTER** or **DS_FILTER_ITEM** OM attributes.

search_aliases
(Boolean). Any aliases in the subordinate entries being searched are dereferenced if the value of this parameter is **OM_TRUE**, and they are not dereferenced if its value is **OM_FALSE**.

selection (Object(**DS_C_ENTRY_INFO_SELECTION**)). Specifies what information from the entry is requested. Information about no attributes, all attributes, or just for a named set can be chosen. Attribute types are always returned, but the attribute values need not be. The possible values of this parameter are listed in the *DCE 1.2.2 Application Development Guide—Directory Services*.

Output

result_return (Object(**DS_C_SEARCH_RESULT**)). If completion is successful, the result contains the requested information from each object in the search space that satisfied the filter. The DN of the target object is present if an alias is dereferenced. In addition, there may be a partial outcome qualifier, which indicates that the result is incomplete. It also explains why it is not complete and how it could be completed.

invoke_id_return
(Integer). Not supported.

Description

The **ds_search()** function is used to search a portion of the directory and return selected information from entries of interest. The information may be incomplete in some circumstances; for example, if the results exceed **DS_SIZE_LIMIT**.

Notes

CDS does not support **ds_search()**, and it returns with **DS_C_SERVICE_ERROR** (**DS_E_UNWILLING_TO_PERFORM**).

Return Values

DS_status Takes the value **DS_SUCCESS** if the named object is located and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**

- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

Note that an unfiltered search of just the base object succeeds even if none of the requested attributes are found, while the **ds_read()** call fails with the same selected attributes.

A **DS_C_SECURITY_ERROR** (**DS_E_INSUFFICIENT_ACCESS_RIGHTS**) is only reported where access rights preclude the reading of all requested attribute values.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_shutdown

Purpose

Deletes a directory workspace

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_shutdown(
        OM_workspace workspace);
```

Parameters

Input

workspace Specifies the workspace (obtained from a call to **ds_initialize()**) that is to be deleted.

Description

The **ds_shutdown()** function deletes the workspace established by **ds_initialize()** and enables the service to release resources. All sessions associated with the workspace must be terminated by calling **ds_unbind()** prior to calling **ds_shutdown()**. No other directory function can reference the specified workspace after it has been deleted. However, **om_delete()** and **om_instance()** may be called if referring to public objects.

Return Values

DS_status **DS_SUCCESS** if the function completed successfully; otherwise, it indicates the error that has occurred.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SERVICE_ERROR** (value **DS_E_BUSY**) if **ds_shutdown()** is called before all directory connections have been released with **ds_unbind()**.

This function can return the error constant **DS_NO_WORKSPACE**.

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors.

Related Information

Functions: **ds_initialize(3xds)**.

ds_unbind

Purpose

Unbinds from a directory session

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_unbind(
        OM_private_object session);
```

Parameters

Input

session (Object(**DS_C_SESSION**)). The directory session to be unbound. This must be a private object. The value of the **DS_FILE_DESCRIPTOR** OM attribute is **DS_NO_VALID_FILE_DESCRIPTOR** if the function succeeds. The remaining OM attributes are unchanged.

Description

The **ds_unbind()** function terminates the given directory session and makes the parameter unavailable for use with other interface functions (except **ds_bind()**).

The unbound session can be used again as a parameter to **ds_bind()** possibly after modification by the OM functions. When it is no longer required, it must be deleted by using the OM functions.

Return Values

DS_status Takes the value **DS_SUCCESS** if the *session* parameter is unbound and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**

If **ds_unbind()** is called while there are outstanding directory operations (from other threads), then this function will return a **DS_SERVICE_ERROR** with the value **DS_E_BUSY**.

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors. However, this function can return the error constant **DS_NO_WORKSPACE**.

Related Information

Functions: **ds_bind(3xds)**.

ds_version

Purpose

Negotiates features of the interface and service

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status ds_version(
        DS_feature feature_list[ ],
        OM_workspace workspace);
```

Parameters

Input

workspace Specifies the workspace obtained from a call to **om_initialize()** for which the features are to be negotiated. The features will be in effect for operations that use the workspace or directory sessions associated with the workspace.

Input/Output

feature_list[ ]

(**DS_feature**). On input contains an ordered sequence of features, each represented by an object identifier. The sequence is terminated by an object identifier having no components (a length of 0 (zero) and any value for the data pointer.)

If the function completed successfully, an ordered sequence of boolean values are returned, with the same number of elements as the *feature_list*[] parameter. If **OM_TRUE**, each value indicates that the corresponding feature is now part of the interface. If **OM_FALSE**, each value indicates that the corresponding feature is not available.

This result is combined with the *feature_list*[ ] parameter as a single array of structures of type **DS_feature**, which is defined as follows:

```
typedef struct
 {
  OM_object_identifier feature;
  OM_boolean           activated;
 }
  DS_feature;
```

Description

The **ds_version()** function negotiates features of the interface, which are represented by object identifiers. The **DS_BASIC_DIR_CONTENTS_PKG**, **DS_STRONG_AUTHENT_PKG**, and the **MHS_DIR_USER_PKG** specified in the *DCE 1.2.2 Application Development Guide—Directory Services* are negotiable features in this specification. Features can also include vendor extensions, such as the **DSX_GDS_PKG**, and new features in future versions of the XDS specification. Versions are negotiated after a workspace is initialized with **ds_initialize()**.

Return Values

DS_status Takes the value **DS_SUCCESS** if the function completed successfully.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_WORKSPACE**
- **DS_E_MISCELLANEOUS**

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors. However, this function can return the error constant **DS_NO_WORKSPACE**.

encode_alt_addr

Purpose

Converts an alternate address attribute structure into an internal GDS format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

int encode_alt_addr(
        const D2_alt_addr *in,
        D2_str **out);
```

Parameters

in A pointer to an alternate address attribute in a structured format.

Description

The **encode_alt_addr()** converts an alternate address stored in a **D2_alt_addr** structure into a linearized string that is stored in a structure of type **D2_str**. This function is provided for use by DME applications. It converts a structured alternate address attribute into a linear octet string for internal use by GDS.

The **D2_alt_addr** structure contains one field of type **D2_str** for storing the address, followed by a structured field for a set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the start of the address (not zero-terminated). The second component of **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To store additional object identifiers, the address of the first **D2_obj_id** structure has to be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be added.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier

is treated as an octet string; that means there is no BER conversion done by **encode_alt_addr()**.

encode_alt_addr() will allocate memory for the encoded string. (**out*) contains the address of the memory area that should later be freed by the application.

Return Values

***out*	A pointer to the structure **D2_str** which stores the alternate address attribute in an internal GDS format.	
	*(*out)->d2_size* will contain the length of the encoded octet string.	
	*(*out)->d2_value* will be a pointer to the beginning of the encoded octet string. This string is not zero-terminated.	
int	0	If successful.
	–1	If unsuccessful (**malloc()** failure).

Related Information

Functions: **decode_alt_addr(3xds)**.

gds_decode_alt_addr

Purpose

Converts an alternate address attribute from internal GDS format to a structured format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

d2_ret_val gds_decode_alt_addr(
        const D2_str *in,
        D2_alt_addr **out);
```

Parameters

Input

in A pointer to a **D2_str** structure that contains the alternate address attribute in an internal GDS format.

Output

out A pointer to the structure **D2_alt_addr** that stores the alternate address attribute in a structured format.

Description

The **gds_decode_alt_addr()** function converts a linearized string that is stored in a structure **D2_str** into a structured alternate address format stored in a **D2_alt_addr** structure. This function is provided for use by DME applications. It converts an alternate address attribute from an internal GDS format (linear octet string) to a structured format for application usage.

The *in->d2_size* parameter contains the length of the encoded octet string;*in->d2_value* is a pointer to the beginning of the encoded octet string.

The **gds_decode_alt_addr()** function allocates memory for the structured alternate address. The *(*out)* parameter contains the address of the memory area that should later be freed by the application.

The **D2_alt_addr** structure contains one field **D2_str** for the address, followed by a structured field for the set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the beginning of the address (not zero-terminated). The second component of the **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To read a set of object identifiers, the address of the first **D2_obj_id** structure should be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be read.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means that **gds_decode_alt_addr()** does no BER conversion for object identifiers.

Return Values

d2_ret_val **D2_NOERROR** (that is, 0) if successful.

D2_ERROR (that is, -1), if unsuccessful (**malloc()** failure).

Related Information

Functions: **gds_encode_alt_addr(3xds)**.

gds_encode_alt_addr

Purpose

Converts an alternate address attribute structure into an internal GDS format

Synopsis

```
#include <xom.h>#include <xds.h>#include <dce/d2dir.h>

d2_ret_val gds_encode_alt_addr(
        const D2_alt_addr *in,
        D2_str **out);
```

Parameters

Input

in A pointer to an alternate address attribute in a structured format.

Output

out A pointer to the structure **D2_str** that stores the alternate address attribute in an internal GDS format.

The *(*out)->d2_size* parameter will contain the length of the encoded octet string; the *(*out)->d2_value* parameter will be a pointer to the beginning of the encoded octet string. This string is not zero-terminated.

Description

The **gds_encode_alt_addr()** function converts an alternate address stored in a **D2_alt_addr** structure into a linearized string that is stored in a structure of type **D2_str**. This function is provided for use by DME applications. It converts a structured alternate address attribute into a linear octet string for internal use by GDS.

The **D2_alt_addr** structure contains one field of type **D2_str** for storing the address, followed by a structured field for a set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the start of the address (not

zero-terminated). The second component of **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To store additional object identifiers, the address of the first **D2_obj_id** structure has to be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be added.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means there is no BER conversion done by **gds_encode_alt_addr()**.

The **gds_encode_alt_addr()** function will allocate memory for the encoded string. The (**out*) parameter contains the address of the memory area that should later be freed by the application.

Return Values

d2_ret_val **D2_NOERROR** (that is, 0), if successful.

D2_ERROR (that is, -1), if unsuccessful (**malloc()** failure).

Related Information

Functions: **gds_decode_alt_addr(3xds)**.

xds_intro

Purpose

Introduction to XDS header files

Description

There are nine XDS headers, as follows:

xds.h Contains definitions for the XDS functions and directory service package.

xdsbdcp.h Contains definitions for the basic directory contents package.

xdssap.h Contains definitions for the strong authentication package.

xdscds.h Contains definitions for the cell directory service.

xdsdme.h Contains definitions for the DME specific directory object and attribute.

xdsgds.h Contains definitions for the global directory service package.

xdsmdup.h Contains definitions for the MHS directory user package.

xmhp.h Contains definitions for the MHS directory objects/attributes.

xmsga.h Contains definitions for the message store general attributes.

The **xds.h** header file is a mandatory include for all applications using the XDS API.

The **xdsbdcp.h**,**xdsmdup.h**, and **xdssap.h** headers are part of the X/Open XDS specifications. They are required when using the basic directory contents package, MHS directory user package, and strong authentication package respectively.

The **xdsgds.h** and **xdscds.h** headers are DCE extensions to the XDS API. The **xdsgds.h** header is required when using the GDS package. The **xdscds.h** header is required when using CDS.

The **xmhp.h** and **xmsga.h** headers are required when using the MHS directory user package.

The **xdsdme.h** header is required when using the DME specific directory object class and attribute.

xds.h

Purpose

Definitions for the directory service package

Synopsis

#include <xom.h>
#include <xds.h>

Description

The **xds.h** header declares the interface functions, the structures passed to and from those functions, and the defined constants used by the functions and structures.

All application programs that include this header must first include the **xom.h** object management header.

```
#ifndef XDS_HEADER
#define XDS_HEADER

/* DS package object identifier */
/* { iso(1) identified-organization(3) icd-ecma(12)
     member-company(2) dec(1011) xopen(28) dsp(0) } */

#define OMP_O_DS_SERVICE_PKG        "\x2B\x0C\x02\x87\x73\x1C\x00"

/*Defined constants */

/* Intermediate object identifier macro */

#define dsP_c(X)        OMP_O_DS_SERVICE_PKG #X
```

```
/* OM class names (prefixed by DS_C_) */

/* Every application program which makes use of a class or other */
/* Object Identifier must explicitly import it into every        */
/* compilation unit (C source program) which uses it.  Each such */
/* class or Object Identifier name must be explicitly exported   */
/* from just one compilation unit.                               */

/* In the header file, OM class constants are prefixed with the  */
/* OMP_O prefix to denote that they are OM classes.  However,    */
/* when using the OM_IMPORT and OM_EXPORT macros, the base       */
/* names (without the OMP_O prefix) should be used.              */
/* For example:                                                  */
/*        OM_IMPORT (DS_C_AVA)                                   */


#define OMP_O_DS_C_ABANDON_FAILED             dsP_c(\x85\x3D)
#define OMP_O_DS_C_ACCESS_POINT               dsP_c(\x85\x3E)
#define OMP_O_DS_C_ADDRESS                    dsP_c(\x85\x3F)
#define OMP_O_DS_C_ATTRIBUTE                  dsP_c(\x85\x40)
#define OMP_O_DS_C_ATTRIBUTE_ERROR            dsP_c(\x85\x41)
#define OMP_O_DS_C_ATTRIBUTE_LIST             dsP_c(\x85\x42)
#define OMP_O_DS_C_ATTRIBUTE_PROBLEM          dsP_c(\x85\x43)
#define OMP_O_DS_C_AVA                        dsP_c(\x85\x44)
#define OMP_O_DS_C_COMMON_RESULTS             dsP_c(\x85\x45)
#define OMP_O_DS_C_COMMUNICATIONS_ERROR       dsP_c(\x85\x46)
#define OMP_O_DS_C_COMPARE_RESULT             dsP_c(\x85\x47)
#define OMP_O_DS_C_CONTEXT                    dsP_c(\x85\x48)
#define OMP_O_DS_C_CONTINUATION_REF           dsP_c(\x85\x49)
#define OMP_O_DS_C_DS_DN                      dsP_c(\x85\x4A)
#define OMP_O_DS_C_DS_RDN                     dsP_c(\x85\x4B)
#define OMP_O_DS_C_ENTRY_INFO                 dsP_c(\x85\x4C)
#define OMP_O_DS_C_ENTRY_INFO_SELECTION       dsP_c(\x85\x4D)
#define OMP_O_DS_C_ENTRY_MOD                  dsP_c(\x85\x4E)
#define OMP_O_DS_C_ENTRY_MOD_LIST             dsP_c(\x85\x4F)
#define OMP_O_DS_C_ERROR                      dsP_c(\x85\x50)
#define OMP_O_DS_C_EXT                        dsP_c(\x85\x51)
#define OMP_O_DS_C_FILTER                     dsP_c(\x85\x52)
#define OMP_O_DS_C_FILTER_ITEM                dsP_c(\x85\x53)
#define OMP_O_DS_C_LIBRARY_ERROR              dsP_c(\x85\x54)
```

```
#define OMP_O_DS_C_LIST_INFO                dsP_c(\x85\x55)
#define OMP_O_DS_C_LIST_INFO_ITEM           dsP_c(\x85\x56)
#define OMP_O_DS_C_LIST_RESULT              dsP_c(\x85\x57)
#define OMP_O_DS_C_NAME                     dsP_c(\x85\x58)
#define OMP_O_DS_C_NAME_ERROR               dsP_c(\x85\x59)
#define OMP_O_DS_C_OPERATION_PROGRESS       dsP_c(\x85\x5A)
#define OMP_O_DS_C_PARTIAL_OUTCOME_QUAL     dsP_c(\x85\x5B)
#define OMP_O_DS_C_PRESENTATION_ADDRESS     dsP_c(\x85\x5C)
#define OMP_O_DS_C_READ_RESULT              dsP_c(\x85\x5D)
#define OMP_O_DS_C_REFERRAL                 dsP_c(\x85\x5E)
#define OMP_O_DS_C_RELATIVE_NAME            dsP_c(\x85\x5F)
#define OMP_O_DS_C_SEARCH_INFO              dsP_c(\x85\x60)
#define OMP_O_DS_C_SEARCH_RESULT            dsP_c(\x85\x61)
#define OMP_O_DS_C_SECURITY_ERROR           dsP_c(\x85\x62)
#define OMP_O_DS_C_SERVICE_ERROR            dsP_c(\x85\x63)
#define OMP_O_DS_C_SESSION                  dsP_c(\x85\x64)
#define OMP_O_DS_C_SYSTEM_ERROR             dsP_c(\x85\x65)
#define OMP_O_DS_C_UPDATE_ERROR             dsP_c(\x85\x66)


/* OM attribute names */

#define DS_ACCESS_POINTS                    ((OM_type) 701)
#define DS_ADDRESS                          ((OM_type) 702)
#define DS_AE_TITLE                         ((OM_type) 703)
#define DS_ALIASED_RDNS                     ((OM_type) 704)
#define DS_ALIAS_DEREFERENCED               ((OM_type) 705)
#define DS_ALIAS_ENTRY                      ((OM_type) 706)
#define DS_ALL_ATTRIBUTES                   ((OM_type) 707)
#define DS_ASYNCHRONOUS                     ((OM_type) 708)
#define DS_ATTRIBUTES                       ((OM_type) 709)
#define DS_ATTRIBUTES_SELECTED              ((OM_type) 710)
#define DS_ATTRIBUTE_TYPE                   ((OM_type) 711)
#define DS_ATTRIBUTE_VALUE                  ((OM_type) 712)
#define DS_ATTRIBUTE_VALUES                 ((OM_type) 713)
#define DS_AUTOMATIC_CONTINUATION           ((OM_type) 714)
#define DS_AVAS                             ((OM_type) 715)
#define DS_CHAINING_PROHIB                  ((OM_type) 716)
#define DS_CHANGES                          ((OM_type) 717)
#define DS_CRIT                             ((OM_type) 718)
```

```
#define DS_DONT_DEREFERENCE_ALIASES          ((OM_type) 719)
#define DS_DONT_USE_COPY                     ((OM_type) 720)
#define DS_DSA_ADDRESS                       ((OM_type) 721)
#define DS_DSA_NAME                          ((OM_type) 722)
#define DS_ENTRIES                           ((OM_type) 723)
#define DS_ENTRY                             ((OM_type) 724)
#define DS_EXT                               ((OM_type) 725)
#define DS_FILE_DESCRIPTOR                   ((OM_type) 726)
#define DS_FILTERS                           ((OM_type) 727)
#define DS_FILTER_ITEMS                      ((OM_type) 728)
#define DS_FILTER_ITEM_TYPE                  ((OM_type) 729)
#define DS_FILTER_TYPE                       ((OM_type) 730)
#define DS_FINAL_SUBSTRING                   ((OM_type) 731)
#define DS_FROM_ENTRY                        ((OM_type) 732)
#define DS_IDENT                             ((OM_type) 733)
#define DS_INFO_TYPE                         ((OM_type) 734)
#define DS_INITIAL_SUBSTRING                 ((OM_type) 735)
#define DS_ITEM_PARAMETERS                   ((OM_type) 736)
#define DS_LIMIT_PROBLEM                     ((OM_type) 737)
#define DS_LIST_INFO                         ((OM_type) 738)
#define DS_LOCAL_SCOPE                       ((OM_type) 739)
#define DS_MATCHED                           ((OM_type) 740)
#define DS_MOD_TYPE                          ((OM_type) 741)
#define DS_NAME_RESOLUTION_PHASE             ((OM_type) 742)
#define DS_NEXT_RDN_TO_BE_RESOLVED           ((OM_type) 743)
#define DS_N_ADDRESSES                       ((OM_type) 744)
#define DS_OBJECT_NAME                       ((OM_type) 745)
#define DS_OPERATION_PROGRESS                ((OM_type) 746)
#define DS_PARTIAL_OUTCOME_QUAL              ((OM_type) 747)
#define DS_PERFORMER                         ((OM_type) 748)
#define DS_PREFER_CHAINING                   ((OM_type) 749)
#define DS_PRIORITY                          ((OM_type) 750)
#define DS_PROBLEM                           ((OM_type) 751)
#define DS_PROBLEMS                          ((OM_type) 752)
#define DS_P_SELECTOR                        ((OM_type) 753)
#define DS_RDN                               ((OM_type) 754)
#define DS_RDNS                              ((OM_type) 755)
#define DS_RDNS_RESOLVED                     ((OM_type) 756)
#define DS_REQUESTOR                         ((OM_type) 757)
#define DS_SCOPE_OF_REFERRAL                 ((OM_type) 758)
```

```
#define DS_SEARCH_INFO                          ((OM_type) 759)
#define DS_SIZE_LIMIT                           ((OM_type) 760)
#define DS_SUBORDINATES                         ((OM_type) 761)
#define DS_S_SELECTOR                           ((OM_type) 762)
#define DS_TARGET_OBJECT                        ((OM_type) 763)
#define DS_TIME_LIMIT                           ((OM_type) 764)
#define DS_T_SELECTOR                           ((OM_type) 765)
#define DS_UNAVAILABLE_CRIT_EXT                 ((OM_type) 766)
#define DS_UNCORRELATED_LIST_INFO               ((OM_type) 767)
#define DS_UNCORRELATED_SEARCH_INFO             ((OM_type) 768)
#define DS_UNEXPLORED                           ((OM_type) 769)


/* DS_Filter_Item_Type: */

enum DS_Filter_Item_Type {
        DS_EQUALITY          = 0,
        DS_SUBSTRINGS        = 1,
        DS_GREATER_OR_EQUAL  = 2,
        DS_LESS_OR_EQUAL     = 3,
        DS_PRESENT           = 4,
        DS_APPROXIMATE_MATCH = 5
};


/* DS_Filter_Type: */

enum DS_Filter_Type {
        DS_ITEM = 0,
        DS_AND  = 1,
        DS_OR   = 2,
        DS_NOT  = 3
};


/* DS_Information_Type: */

enum DS_Information_Type {
        DS_TYPES_ONLY       = 0,
        DS_TYPES_AND_VALUES = 1
```

```
};


/* DS_Limit_Problem: */

enum DS_Limit_Problem {
        DS_NO_LIMIT_EXCEEDED            = -1,
        DS_TIME_LIMIT_EXCEEDED          = 0,
        DS_SIZE_LIMIT_EXCEEDED          = 1,
        DS_ADMIN_LIMIT_EXCEEDED         = 2
};


/* DS_Modification_Type: */

enum DS_Modification_Type {
        DS_ADD_ATTRIBUTE    = 0,
        DS_REMOVE_ATTRIBUTE = 1,
        DS_ADD_VALUES       = 2,
        DS_REMOVE_VALUES    = 3
};


/* DS_Name_Resolution_Phase: */

enum DS_Name_Resolution_Phase {
        DS_NOT_STARTED = 1,
        DS_PROCEEDING  = 2,
        DS_COMPLETED   = 3
};


/* DS_Priority: */

enum DS_Priority {
        DS_LOW    = 0,
        DS_MEDIUM = 1,
        DS_HIGH   = 2
};
```

```
/* DS_Problem: */

enum DS_Problem {
 DS_E_ADMIN_LIMIT_EXCEEDED                = 1,
 DS_E_AFFECTS_MULTIPLE_DSAS               = 2,
 DS_E_ALIAS_DEREFERENCING_PROBLEM         = 3,
 DS_E_ALIAS_PROBLEM                       = 4,
 DS_E_ATTRIBUTE_OR_VALUE_EXISTS           = 5,
 DS_E_BAD_ARGUMENT                        = 6,
 DS_E_BAD_CLASS                           = 7,
 DS_E_BAD_CONTEXT                         = 8,
 DS_E_BAD_NAME                            = 9,
 DS_E_BAD_SESSION                         = 10,
 DS_E_BAD_WORKSPACE                       = 11,
 DS_E_BUSY                                = 12,
 DS_E_CANNOT_ABANDON                      = 13,
 DS_E_CHAINING_REQUIRED                   = 14,
 DS_E_COMMUNICATIONS_PROBLEM              = 15,
 DS_E_CONSTRAINT_VIOLATION                = 16,
 DS_E_DIT_ERROR                           = 17,
 DS_E_ENTRY_EXISTS                        = 18,
 DS_E_INAPPROP_AUTHENTICATION             = 19,
 DS_E_INAPPROP_MATCHING                   = 20,
 DS_E_INSUFFICIENT_ACCESS_RIGHTS          = 21,
 DS_E_INVALID_ATTRIBUTE_SYNTAX            = 22,
 DS_E_INVALID_ATTRIBUTE_VALUE             = 23,
 DS_E_INVALID_CREDENTIALS                 = 24,
 DS_E_INVALID_REF                         = 25,
 DS_E_INVALID_SIGNATURE                   = 26,
 DS_E_LOOP_DETECTED                       = 27,
 DS_E_MISCELLANEOUS                       = 28,
 DS_E_MISSING_TYPE                        = 29,
 DS_E_MIXED_SYNCHRONOUS                   = 30,
 DS_E_NAMING_VIOLATION                    = 31,
 DS_E_NO_INFO                             = 32,
 DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE          = 33,
 DS_E_NO_SUCH_OBJECT                      = 34,
 DS_E_NO_SUCH_OPERATION                   = 35,
 DS_E_NOT_ALLOWED_ON_NON_LEAF             = 36,
```

```
 DS_E_NOT_ALLOWED_ON_RDN                  = 37,
 DS_E_NOT_SUPPORTED                       = 38,
 DS_E_OBJECT_CLASS_MOD_PROHIB             = 39,
 DS_E_OBJECT_CLASS_VIOLATION              = 40,
 DS_E_OUT_OF_SCOPE                        = 41,
 DS_E_PROTECTION_REQUIRED                 = 42,
 DS_E_TIME_LIMIT_EXCEEDED                 = 43,
 DS_E_TOO_LATE                            = 44,
 DS_E_TOO_MANY_OPERATIONS                 = 45,
 DS_E_TOO_MANY_SESSIONS                   = 46,
 DS_E_UNABLE_TO_PROCEED                   = 47,
 DS_E_UNAVAILABLE                         = 48,
 DS_E_UNAVAILABLE_CRIT_EXT                = 49,
 DS_E_UNDEFINED_ATTRIBUTE_TYPE            = 50,
 DS_E_UNWILLING_TO_PERFORM                = 51
};


/* DS_Scope_Of_Referral: */

enum DS_Scope_Of_Referral {
        DS_DMD     = 0,
        DS_COUNTRY = 1
};


/* Typedefs */

typedef OM_private_object  DS_status;

typedef struct
{
    OM_object_identifier feature;
    OM_boolean           activated;
} DS_feature;


/* OM_object constants */

#define DS_DEFAULT_CONTEXT             ((OM_object) 0)
```

```
#define DS_DEFAULT_SESSION              ((OM_object) 0)
#define DS_OPERATION_NOT_STARTED        ((OM_object) 0)
#define DS_NO_FILTER                    ((OM_object) 0)
#define DS_NULL_RESULT                  ((OM_object) 0)
#define DS_SELECT_ALL_TYPES             ((OM_object) 1)
#define DS_SELECT_ALL_TYPES_AND_VALUES  ((OM_object) 2)
#define DS_SELECT_NO_ATTRIBUTES         ((OM_object) 0)
#define DS_SUCCESS                      ((DS_status) 0)
#define DS_NO_WORKSPACE                 ((DS_status) 1)


/* ds_search() subset */

#define DS_BASE_OBJECT                  ((OM_sint) 0)
#define DS_ONE_LEVEL                    ((OM_sint) 1)
#define DS_WHOLE_SUBTREE                ((OM_sint) 2)


/* ds_receive_result() completion_flag_return */

#define DS_COMPLETED_OPERATION          ((OM_uint) 1)
#define DS_OUTSTANDING_OPERATIONS       ((OM_uint) 2)
#define DS_NO_OUTSTANDING_OPERATION     ((OM_uint) 3)


/* asynchronous operations limit (implementation-defined) */

#define DS_MAX_OUTSTANDING_OPERATIONS   0  /* no asynchronous */
                                           /* operation       */
/*asynchronous event posting */

#define DS_NO_VALID_FILE_DESCRIPTOR     -1

/* Function Prototypes */

DS_status ds_abandon(
        OM_private_object   session,
        OM_sint             invoke_id
);
```

```
DS_status ds_add_entry(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_object           entry,
        OM_sint             *invoke_id_return
);

DS_status ds_bind(
        OM_object           session,
        OM_workspace        workspace,
        OM_private_object   *bound_session_return
);

DS_status ds_compare(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_object           ava,
        OM_private_object   *result_return,
        OM_sint             *invoke_id_return
);

OM_workspace ds_initialize(
        void
);

DS_status ds_list(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_private_object   *result_return,
        OM_sint             *invoke_id_return
);

DS_status ds_modify_entry(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_object           changes,
```

```
        OM_sint             *invoke_id_return
);

DS_status ds_modify_rdn(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_object           new_RDN,
        OM_boolean          delete_old_RDN,
        OM_sint             *invoke_id_return
);

DS_status ds_read(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_object           selection,
        OM_private_object   *result_return,
        OM_sint             *invoke_id_return
);

DS_status ds_receive_result(
        OM_private_object   session,
        OM_uint             *completion_flag_return,
        DS_status           *operation_status_return,
        OM_private_object   *result_return,
        OM_sint             *invoke_id_return
);

DS_status ds_remove_entry(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
        OM_sint             *invoke_id_return
);

DS_status ds_search(
        OM_private_object   session,
        OM_private_object   context,
        OM_object           name,
```

```
        OM_sint             subset,
        OM_object           filter,
        OM_boolean          search_aliases,
        OM_object           selection,
        OM_private_object   *result_return,
        OM_sint             *invoke_id_return
);

DS_status ds_shutdown(
        OM_workspace        workspace
);

DS_status ds_unbind(
        OM_private_object   session
);

DS_status ds_version(
        DS_feature          feature_list[]
        OM_workspace        workspace
);


#endif  /* XDS_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services.*

xdsbdcp.h

Purpose

Definitions for the basic directory contents package

Synopsis

#include <xom.h>#include <xds.h>#include <xdsbdcp.h>

Description

The **xdsbdcp.h** header defines the object identifiers of directory attribute types and object classes supported by the basic directory contents package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

Object identifiers are defined for the (directory) attribute types that are specified in the following list. The actual values of the object identifiers are listed in the *DCE 1.2.2 Application Development Guide—Directory Services*.

```
#ifndef XDSBDCP_HEADER
#define XDSBDCP_HEADER

/* BDC package object identifier */
/* { iso(1) identified-organization(3) icd-ecma(12)
     member-company(2) dec(1011) xopen(28) bdcp(1) } */

#define OMP_O_DS_BASIC_DIR_CONTENTS_PKG \
 "\x2B\x0C\x02\x87\x73\x1c\x01"


/* Intermediate object identifier macros */
```

```
#ifndef dsP_attributeType     /* joint-iso-ccitt(2)         */
                                /* ds(5) attributeType(4) ... */
#define dsP_attributeType (X)   ("\x55\x04" #X)
#endif

#ifndef dsP_objectClass       /* joint-iso-ccitt(2)         */
                                /* ds(5) objectClass(6) ...   */
#define dsP_objectClass(X)      ("\x55\x06" #X)
#endif

#define dsP_bdcp_c(X) (OMP_O_DS_BASIC_DIR_CONTENTS_PKG #X)


/* OM class names (prefixed by DS_C_)                       */
/* Directory attribute types (prefixed by DS_A_)            */
/* Directory object classes (prefixed by DS_O_)             */

/* Every application program which makes use of a class or  */
/* other Object Identifier must explicitly import it into   */
/* every compilation unit (C source program) which uses it. */
/* Each such class or Object Identifier name must be        */
/* explicitly exported from just one compilation unit.      */

/* In the header file, OM class constants are prefixed with */
/* the OMP_O prefix to denote that they are OM classes.     */
/* However, when using the OM_IMPORT and OM_EXPORT macros,  */
/* the base names (without the OMP_O prefix) should be used.*/
/* For example:                                             */
/*         OM_IMPORT (DS_O_COUNTRY)                         */


/* Directory attribute types */

#define OMP_O_DS_A_ALIASED_OBJECT_NAME    dsP_attributeType(\x01)
#define OMP_O_DS_A_BUSINESS_CATEGORY      dsP_attributeType(\x0F)
#define OMP_O_DS_A_COMMON_NAME            dsP_attributeType(\x03)
#define OMP_O_DS_A_COUNTRY_NAME           dsP_attributeType(\x06)
#define OMP_O_DS_A_DESCRIPTION            dsP_attributeType(\x0D)
#define OMP_O_DS_A_DEST_INDICATOR         dsP_attributeType(\x1B)
#define OMP_O_DS_A_FACSIMILE_PHONE_NBR    dsP_attributeType(\x17)
```

```
#define OMP_O_DS_A_INTERNAT_ISDN_NBR      dsP_attributeType(\x19)
#define OMP_O_DS_A_KNOWLEDGE_INFO         dsP_attributeType(\x02)
#define OMP_O_DS_A_LOCALITY_NAME          dsP_attributeType(\x07)
#define OMP_O_DS_A_MEMBER                 dsP_attributeType(\x1F)
#define OMP_O_DS_A_OBJECT_CLASS           dsP_attributeType(\x00)
#define OMP_O_DS_A_ORG_NAME               dsP_attributeType(\x0A)
#define OMP_O_DS_A_ORG_UNIT_NAME          dsP_attributeType(\x0B)
#define OMP_O_DS_A_OWNER                  dsP_attributeType(\x20)
#define OMP_O_DS_A_PHYS_DELIV_OFF_NAME    dsP_attributeType(\x13)
#define OMP_O_DS_A_POST_OFFICE_BOX        dsP_attributeType(\x12)
#define OMP_O_DS_A_POSTAL_ADDRESS         dsP_attributeType(\x10)
#define OMP_O_DS_A_POSTAL_CODE            dsP_attributeType(\x11)
#define OMP_O_DS_A_PREF_DELIV_METHOD      dsP_attributeType(\x1C)
#define OMP_O_DS_A_PRESENTATION_ADDRESS   dsP_attributeType(\x1D)
#define OMP_O_DS_A_REGISTERED_ADDRESS     dsP_attributeType(\x1A)
#define OMP_O_DS_A_ROLE_OCCUPANT          dsP_attributeType(\x21)
#define OMP_O_DS_A_SEARCH_GUIDE           dsP_attributeType(\x0E)
#define OMP_O_DS_A_SEE_ALSO               dsP_attributeType(\x22)
#define OMP_O_DS_A_SERIAL_NBR             dsP_attributeType(\x05)
#define OMP_O_DS_A_STATE_OR_PROV_NAME     dsP_attributeType(\x08)
#define OMP_O_DS_A_STREET_ADDRESS         dsP_attributeType(\x09)
#define OMP_O_DS_A_SUPPORT_APPLIC_CONTEXT dsP_attributeType(\x1E)
#define OMP_O_DS_A_SURNAME                dsP_attributeType(\x04)
#define OMP_O_DS_A_PHONE_NBR              dsP_attributeType(\x14)
#define OMP_O_DS_A_TELETEX_TERM_IDENT     dsP_attributeType(\x16)
#define OMP_O_DS_A_TELEX_NBR              dsP_attributeType(\x15)
#define OMP_O_DS_A_TITLE                  dsP_attributeType(\x0C)
#define OMP_O_DS_A_USER_PASSWORD          dsP_attributeType(\x23)
#define OMP_O_DS_A_X121_ADDRESS           dsP_attributeType(\x18)


/* Directory object classes */

#define OMP_O_DS_O_ALIAS                  dsP_objectClass(\x01)
#define OMP_O_DS_O_APPLIC_ENTITY          dsP_objectClass(\x0C)
#define OMP_O_DS_O_APPLIC_PROCESS         dsP_objectClass(\x0B)
#define OMP_O_DS_O_COUNTRY                dsP_objectClass(\x02)
#define OMP_O_DS_O_DEVICE                 dsP_objectClass(\x0E)
#define OMP_O_DS_O_DSA                    dsP_objectClass(\x0D)
#define OMP_O_DS_O_GROUP_OF_NAMES         dsP_objectClass(\x09)
```

```
#define OMP_O_DS_O_LOCALITY              dsP_objectClass(\x03)
#define OMP_O_DS_O_ORG                   dsP_objectClass(\x04)
#define OMP_O_DS_O_ORG_PERSON            dsP_objectClass(\x07)
#define OMP_O_DS_O_ORG_ROLE              dsP_objectClass(\x08)
#define OMP_O_DS_O_ORG_UNIT              dsP_objectClass(\x05)
#define OMP_O_DS_O_PERSON                dsP_objectClass(\x06)
#define OMP_O_DS_O_RESIDENTIAL_PERSON    dsP_objectClass(\x0A)
#define OMP_O_DS_O_TOP                   dsP_objectClass(\x00)


/* OM class names */

#define OMP_O_DS_C_FACSIMILE_PHONE_NBR        dsP_bdcp_c(\x86\x21)
#define OMP_O_DS_C_POSTAL_ADDRESS             dsP_bdcp_c(\x86\x22)
#define OMP_O_DS_C_SEARCH_CRITERION           dsP_bdcp_c(\x86\x23)
#define OMP_O_DS_C_SEARCH_GUIDE               dsP_bdcp_c(\x86\x24)
#define OMP_O_DS_C_TELETEX_TERM_IDENT         dsP_bdcp_c(\x86\x25)
#define OMP_O_DS_C_TELEX_NBR                  dsP_bdcp_c(\x86\x26)


/* OM attribute names */

#define DS_ANSWERBACK          ((OM_type) 801)
#define DS_COUNTRY_CODE        ((OM_type) 802)
#define DS_CRITERIA            ((OM_type) 803)
#define DS_OBJECT_CLASS        ((OM_type) 804)
#define DS_PARAMETERS          ((OM_type) 805)
#define DS_POSTAL_ADDRESS      ((OM_type) 806)
#define DS_PHONE_NBR           ((OM_type) 807)
#define DS_TELETEX_TERM        ((OM_type) 808)
#define DS_TELEX_NBR           ((OM_type) 809)

/* DS_Preferred_Delivery_Method: */

#define DS_ANY_DELIV_METHOD         0
#define DS_MHS_DELIV                1
#define DS_PHYS_DELIV               2
#define DS_TELEX_DELIV              3
#define DS_TELETEX_DELIV            4
#define DS_G3_FACSIMILE_DELIV       5
```

```
#define DS_G4_FACSIMILE_DELIV        6
#define DS_IA5_TERMINAL_DELIV        7
#define DS_VIDEOTEX_DELIV            8
#define DS_PHONE_DELIV               9


/* Upper bounds on string lengths and the number of repeated OM */
/* attribute values                                             */

#define DS_VL_A_BUSINESS_CATEGORY         ((OM_value_length) 128)
#define DS_VL_A_COMMON_NAME               ((OM_value_length) 64)
#define DS_VL_A_DESCRIPTION               ((OM_value_length) 1024)
#define DS_VL_A_DEST_INDICATOR            ((OM_value_length) 128)
#define DS_VL_A_INTERNAT_ISDN_NBR         ((OM_value_length) 16)
#define DS_VL_A_LOCALITY_NAME             ((OM_value_length) 128)
#define DS_VL_A_ORG_NAME                  ((OM_value_length) 64)
#define DS_VL_A_ORG_UNIT_NAME             ((OM_value_length) 64)
#define DS_VL_A_PHYS_DELIV_ OFF_NAME      ((OM_value_length) 128)
#define DS_VL_A_POST_OFFICE_BOX           ((OM_value_length) 40)
#define DS_VL_A_POSTAL_CODE               ((OM_value_length) 40)
#define DS_VL_A_SERIAL_NBR                ((OM_value_length) 64)
#define DS_VL_A_STATE_OR_PROV_NAME        ((OM_value_length) 128)
#define DS_VL_A_STREET_ADDRESS            ((OM_value_length) 128)
#define DS_VL_A_SURNAME                   ((OM_value_length) 64)
#define DS_VL_A_PHONE_NBR                 ((OM_value_length) 32)
#define DS_VL_A_TITLE                     ((OM_value_length) 64)
#define DS_VL_A_USER_PASSWORD             ((OM_value_length) 128)
#define DS_VL_A_X121_ADDRESS              ((OM_value_length) 15)
#define DS_VL_ANSWERBACK                  ((OM_value_length) 8)
#define DS_VL_COUNTRY_CODE                ((OM_value_length) 4)
#define DS_VL_POSTAL_ADDRESS              ((OM_value_length) 30)
#define DS_VL_PHONE_NBR                   ((OM_value_length) 32)
#define DS_VL_TELETEX_TERM                ((OM_value_length) 1024)
#define DS_VL_TELEX_NBR                   ((OM_value_length) 14)
#define DS_VN_POSTAL_ADDRESS              ((OM_value_length) 6)

#endif  /* XDSBDCP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services*.

xdscds.h

Purpose

Definitions for the Cell Directory Service (CDS)

Synopsis

#include <xom.h>#include <xds.h>#include <xdscds.h>

Description

The **xdscds.h** header declares the object identifiers of directory attribute types supported by CDS.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSCDS_HEADER
#define XDSCDS_HEADER

/* iso(1) identified-organization(3) osf(22) dce(1) cds(3)
    = "\x2B\x16\x01\x03" */

/* Cell Directory Service attribute types */

#define OMP_O_DSX_A_CDS_Members         "\x2B\x16\x01\x03\x0A"
#define OMP_O_DSX_A_CDS_GroupRevoke     "\x2B\x16\x01\x03\x0B"
#define OMP_O_DSX_A_CDS_CTS             "\x2B\x16\x01\x03\x0C"
#define OMP_O_DSX_A_CDS_UTS             "\x2B\x16\x01\x03\x0D"
#define OMP_O_DSX_A_CDS_ACS             "\x2B\x16\x01\x03\x0E"
#define OMP_O_DSX_A_CDS_Class           "\x2B\x16\x01\x03\x0F"
#define OMP_O_DSX_A_CDS_ClassVersion    "\x2B\x16\x01\x03\x10"
#define OMP_O_DSX_A_CDS_ObjectUID       "\x2B\x16\x01\x03\x11"
#define OMP_O_DSX_A_CDS_Address         "\x2B\x16\x01\x03\x12"
#define OMP_O_DSX_A_CDS_Replicas        "\x2B\x16\x01\x03\x13"
```

```
#define OMP_O_DSX_A_CDS_AllUpTo         "\x2B\x16\x01\x03\x14"
#define OMP_O_DSX_A_CDS_Convergence     "\x2B\x16\x01\x03\x15"
#define OMP_O_DSX_A_CDS_InCHName        "\x2B\x16\x01\x03\x16"
#define OMP_O_DSX_A_CDS_ParentPointer   "\x2B\x16\x01\x03\x17"
#define OMP_O_DSX_A_CDS_DirecoryVersion "\x2B\x16\x01\x03\x18"
#define OMP_O_DSX_A_CDS_UpgradeTo       "\x2B\x16\x01\x03\x19"
#define OMP_O_DSX_A_CDS_LinkTarget      "\x2B\x16\x01\x03\x1B"
#define OMP_O_DSX_A_CDS_LinkTimeout     "\x2B\x16\x01\x03\x1C"
#define OMP_O_DSX_A_CDS_Towers          "\x2B\x16\x01\x03\x1E"
#define OMP_O_DSX_A_CDS_CHName          "\x2B\x16\x01\x03\x20"
#define OMP_O_DSX_A_CDS_CHLastAddress   "\x2B\x16\x01\x03\x22"
#define OMP_O_DSX_A_CDS_CHUpPointers    "\x2B\x16\x01\x03\x23"
#define OMP_O_DSX_A_CDS_CHState         "\x2B\x16\x01\x03\x24"

/* iso(1) identified-organization(3) osf(22) dce(1) gds(2)
    = "\x2B\x16\x01\x02" */

#define OMP_O_DSX_UUID                  "\x2B\x16\x01\x01\x01"
#define OMP_O_DSX_TYPELESS_RDN          "\x2B\x16\x01\x01\x02"
#define OMP_O_DSX_NORMAL_SIMPLE_NAME    "\x2B\x16\x01\x03\x00"
#define OMP_O_DSX_BINARY_SIMPLE_NAME    "\x2B\x16\x01\x03\x02"

#endif  /*XDSCDS_HEADER*/
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services.*

xdsdme.h

Purpose

Definitions for the DME NMO requirements.

Synopsis

#include <xom.h>#include <xds.h>#include <xdsdme.h>

Description

The **xdsdme.h** header declares the object identifiers of directory attribute types and directory object classes supported for DME use.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSDME_HEADER
#define XDSDME_HEADER


/* Intermediate object identifier macros */

/* iso(1) identified-organization(3) osf(22) dme(2)
   components(1) nmo(2) dmeNmoAttributeType(1) ...
*/

#define dsP_NMOattributeType(X)   "\x2B\x16\x02\x01\x02\x01" #X

/* iso(1) identified-organization(3) osf(22) dme(2)
   components(1) nmo(2) dmeNmoObjectClass(2) ...
*/

#define dsP_NMOobjectClass(X)     "\x2B\x16\x02\x01\x02\x02" #X
```

```
/* Directory attribute types (prefixed by DSX_A_)
   Directory object classes (prefixed by DSX_O_)
*/


/* Directory attribute types */

#define OMP_O_DSX_A_ALTERNATE_ADDRESS dsP_NMOattributeType(\x01)

/* Directory object classes */

#define OMP_O_DSX_O_DME_NMO_AGENT dsP_NMOobjectClass(\x01)


#endif /* XDSDME_HEADER */
```

Related Information

Books: *DCE 1.2.2 Application Development Guide—Directory Services.*

xdsgds.h

Purpose

Definitions for the global directory service package

Synopsis

#include <xom.h>#include <xds.h>#include <xdsgds.h>

Description

The **xdsgds.h** header declares the object identifiers of directory attribute types and directory object classes supported by the GDS package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSGDS_HEADER
#define XDSGDS_HEADER


/* GDS package object identifier */
/* iso(1) identified-organization(3) icd-ecma(0012)
   member-company(2) siemens-units(1107) sni(1) directory(3)
   xds-api(100)gdsp(0) */

#define OMP_O_DSX_GDS_PKG   \
"\x2B\x0C\x02\x88\x53\x01\x03\x64\x00"

/*Intermediate object identifier macros */

/* iso(1) identified-organization(3) icd-ecma(0012)
   member-company(2) siemens-units(1107) sni(1) directory(3)
   attribute-type(4) ...*/
```

```
#define dsP_GDSattributeType(X) \
("\x2B\x0C\x02\x88\x53\x01\x03\x04" #X)

/* iso(1) identified-organization(3) icd-ecma(0012)
   member-company(2) siemens-units(1107) sni(1) directory(3)
   object-class(6) ...*/

#define dsP_GDSobjectClass(X) \
("\x2B\x0C\x02\x88\x53\x01\x03\x06" #X)

#define dsP_gdsp_c(X)    OMP_O_DSX_GDS_PKG #X

/* OM class names (prefixed by DSX_C_)
   Directory attribute types (prefixed by DSX_A_)
   Directory object classes (prefixed by DSX_O_)
*/


/* Directory attribute types */

#define OMP_O_DSX_A_MASTER_KNOWLEDGE dsP_GDSattributeType(\x00)
#define OMP_O_DSX_A_ACL              dsP_GDSattributeType(\x01)
#define OMP_O_DSX_A_TIME_STAMP       dsP_GDSattributeType(\x02)
#define OMP_O_DSX_A_SHADOWED_BY      dsP_GDSattributeType(\x03)
#define OMP_O_DSX_A_SRT              dsP_GDSattributeType(\x04)
#define OMP_O_DSX_A_OCT              dsP_GDSattributeType(\x05)
#define OMP_O_DSX_A_AT               dsP_GDSattributeType(\x06)
#define OMP_O_DSX_A_DEFAULT_DSA      dsP_GDSattributeType(\x08)
#define OMP_O_DSX_A_LOCAL_DSA        dsP_GDSattributeType(\x09)
#define OMP_O_DSX_A_CLIENT           dsP_GDSattributeType(\x0A)
#define OMP_O_DSX_A_DNLIST           dsP_GDSattributeType(\x0B)
#define OMP_O_DSX_A_SHADOWING_JOB    dsP_GDSattributeType(\x0C)
#define OMP_O_DSX_A_CDS_CELL         dsP_GDSattributeType(\x0D)
#define OMP_O_DSX_A_CDS_REPLICA      dsP_GDSattributeType(\x0E)


/* Directory object classes */

#define OMP_O_DSX_O_SCHEMA           dsP_GDSobjectClass(\x00)
```

```
/* OM class names */

#define OMP_O_DSX_C_GDS_SESSION         dsP_gdsp_c(\x00)
#define OMP_O_DSX_C_GDS_CONTEXT         dsP_gdsp_c(\x01)
#define OMP_O_DSX_C_GDS_ACL             dsP_gdsp_c(\x02)
#define OMP_O_DSX_C_GDS_ACL_ITEM        dsP_gdsp_c(\x03)


/* OM attribute names */

#define DSX_PASSWORD                ((OM_type) 850)
#define DSX_DIR_ID                  ((OM_type) 851)
#define DSX_DUAFIRST                ((OM_type) 852)
#define DSX_DONT_STORE              ((OM_type) 853)
#define DSX_NORMAL_CLASS            ((OM_type) 854)
#define DSX_PRIV_CLASS              ((OM_type) 855)
#define DSX_RESIDENT_CLASS          ((OM_type) 856)
#define DSX_USEDSA                  ((OM_type) 857)
#define DSX_DUA_CACHE               ((OM_type) 858)
#define DSX_MODIFY_PUBLIC           ((OM_type) 859)
#define DSX_READ_STANDARD           ((OM_type) 860)
#define DSX_MODIFY_STANDARD         ((OM_type) 861)
#define DSX_READ_SENSITIVE          ((OM_type) 862)
#define DSX_MODIFY_SENSITIVE        ((OM_type) 863)
#define DSX_INTERPRETATION          ((OM_type) 864)
#define DSX_USER                    ((OM_type) 865)
#define DSX_PREFER_ADM_FUNCS        ((OM_type) 866)
#define DSX_AUTH_MECHANISM          ((OM_type) 867)
#define DSX_AUTH_INFO               ((OM_type) 868) /* future use */
#define DSX_SIGN_MECHANISM          ((OM_type) 869) /* future use */
#define DSX_PROT_REQUEST            ((OM_type) 870) /* future use */


/* DSX_Interpretation */

enum DSX_Interpretation {
        DSX_SINGLE_OBJECT    = 0,
        DSX_ROOT_OF_SUBTREE  = 1
```

```
};


enum DSX_Auth_Mechanism {
        DSX_DEFAULT          = 1,
        DSX_SIMPLE           = 2,
        DSX_SIMPLE_PROT1     = 3,
        DSX_SIMPLE_PROT2     = 4,
        DSX_DCE_AUTH         = 5,
        DSX_STRONG           = 6
};


enum DSX_Prot_Request {
        DSX_NONE             = 0,
        DSX_SIGNED           = 1
};


/* upper bound on string lengths*/

#define DSX_VL_PASSWORD         ((OM_value_length) 16)

#endif  /* XDSGDS_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services*.

xdsmdup.h

Purpose

Definitions for the MHS directory user package

Synopsis

#include <xom.h>#include <xds.h>#include <xdsmdup.h>

Description

The **xdsmdup.h** header declares the object identifiers of directory attribute types and object classes supported by the MHS directory user package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the object management header **xom.h** and the **xds.h** header.

```
#ifndef XDSMDUP_HEADER
#define XDSMDUP_HEADER

#ifndef XMHP_HEADER
#include <xmhp.h>
#endif  /* XMHP_HEADER */

/* MDUP package object identifier */

/* { iso(1) identified-organization(3) icd-ecma(12)
     member-company(2) dec(1011) xopen(28) mdup(3) } */

#define OMP_O_DS_MHS_DIR_USER_PKG \
"\x2B\x0C\x02\x87\x73\x1C\x03"

/* Intermediate object identifier macros */
```

```
/* { joint-iso-ccitt(2) mhs-motis(6) arch(5) at(2) } */

#define dsP_MHSattributeType(X) ("\x56\x5\x2" #X)

/* { joint-iso-ccitt(2) mhs-motis(6) arch(5) oc(1) } */

#define dsP_MHSobjectClass(X)   ("\x56\x5\x1" #X)

#define dsP_mdup_c(X)      (OMP_O_DS_MHS_DIR_USER_PKG #X)

/* OM class names (prefixed DS_C_),                              */
/* Directory attribute types (prefixed DS_A_),                   */
/* and Directory object classes (prefixed DS_O_)                 */

/* Every application program which makes use of a class or       */
/* other Object Identifier must explicitly import it into        */
/* every compilation unit (C source program) which uses it.      */
/* Each such class or Object Identifier name must be             */
/* explicitly exported from just one compilation unit.           */

/* In the header file, OM class constants are prefixed with      */
/* the OMP_O prefix to denote that they are OM classes.          */
/* However, when using the OM_IMPORT and OM_EXPORT macros,       */
/* the base names (without the OMP_O prefix) should be used.     */
/* For example:                                                  */
/*        OM_IMPORT(DS_O_CERT_AUTHORITY)                         */


/* Directory attribute types */

#define OMP_O_DS_A_DELIV_CONTENT_LENGTH      dsP_MHSattributeType(\x00)
#define OMP_O_DS_A_DELIV_CONTENT_TYPES       dsP_MHSattributeType(\x01)
#define OMP_O_DS_A_DELIV_EITS                dsP_MHSattributeType(\x02)
#define OMP_O_DS_A_DL_MEMBERS                dsP_MHSattributeType(\x03)
#define OMP_O_DS_A_DL_SUBMIT_PERMS           dsP_MHSattributeType(\x04)
#define OMP_O_DS_A_MESSAGE_STORE             dsP_MHSattributeType(\x05)
#define OMP_O_DS_A_OR_ADDRESSES              dsP_MHSattributeType(\x06)
#define OMP_O_DS_A_PREF_DELIV_METHODS        dsP_MHSattributeType(\x07)
#define OMP_O_DS_A_SUPP_AUTO_ACTIONS         dsP_MHSattributeType(\x08)
```

```
#define OMP_O_DS_A_SUPP_CONTENT_TYPES        dsP_MHSattributeType(\x09)
#define OMP_O_DS_A_SUPP_OPT_ATTRIBUTES       dsP_MHSattributeType(\x0A)


/* Directory object classes */

#define OMP_O_DS_O_MHS_DISTRIBUTION_LIST     dsP_MHSobjectClass(\x00)
#define OMP_O_DS_O_MHS_MESSAGE_STORE         dsP_MHSobjectClass(\x01)
#define OMP_O_DS_O_MHS_MESSAGE_TRANS_AG      dsP_MHSobjectClass(\x02)
#define OMP_O_DS_O_MHS_USER                  dsP_MHSobjectClass(\x03)
#define OMP_O_DS_O_MHS_USER_AG               dsP_MHSobjectClass(\x04)


/* OM class names */

#define OMP_O_DS_C_DL_SUBMIT_PERMS      dsP_mdup_c(\x87\x05)

/* OM attribute names */

#define DS_PERM_TYPE            ( (OM_type) 901 )
#define DS_INDIVIDUAL           ( (OM_type) 902 )
#define DS_MEMBER_OF_DL         ( (OM_type) 903 )
#define DS_PATTERN_MATCH        ( (OM_type) 904 )
#define DS_MEMBER_OF_GROUP      ( (OM_type) 905 )


/* DS_Permission_Type */

enum DS_Permission_Type {
        DS_PERM_INDIVIDUAL      = 0,
        DS_PERM_MEMBER_OF_DL    = 1,
        DS_PERM_PATTERN_MATCH   = 2,
        DS_PERM_MEMBER_OF_GROUP = 3
};


#endif /* XDSMDUP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services, X/Open CAE Specification (November 1991), API to Electronic Mail (X.400).*

xdssap.h

Purpose

Definitions for the strong authentication package

Synopsis

#include <xom.h>#include <xds.h>#include <xdssap.h>

Description

The **xdssap.h** header defines the object identifiers of directory attribute types and object classes supported by the strong authentication package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSSAP_HEADER
#define XDSSAP_HEADER

/* Strong Authentication Package object identifier */
/* { iso(1) identified-organization(3) icd-ecma(12)
     member-company(2) dec(1011) xopen(28) sap(2) } */

#define OMP_O_DS_STRONG_AUTHENT_PKG \
 "\x2B\x0C\x02\x87\x73\x1c\x02"


/* Intermediate object identifier macros */

#ifndef dsP_attributeType     /* joint-iso-ccitt(2)         */
                                /* ds(5) attributeType(4) ... */
#define dsP_attributeType (X)   ("\x55\x04" #X)
#endif
```

```
#ifndef dsP_objectClass       /* joint-iso-ccitt(2)         */
                                /* ds(5) objectClass(6) ...   */
#define dsP_objectClass(X)      ("\x55\x06" #X)
#endif

#define dsP_sap_c(X) (OMP_O_DS_STRONG_AUTHENT_PKG #X)

/* OM class names (prefixed by DS_C_)                       */
/* Directory attribute types (prefixed by DS_A_)            */
/* Directory object classes (prefixed by DS_O_)             */

/* Every application program which makes use of a class or  */
/* other Object Identifier must explicitly import it into   */
/* every compilation unit (C source program) which uses it. */
/* Each such class or Object Identifier name must be        */
/* explicitly exported from just one compilation unit.      */

/* In the header file, OM class constants are prefixed with */
/* the OMP_O prefix to denote that they are OM classes.     */
/* However, when using the OM_IMPORT and OM_EXPORT macros,  */
/* the base names (without the OMP_O prefix) should be used.*/
/* For example:                                             */
/*         OM_IMPORT (DS_O_CERT_AUTHORITY)                  */


/* Directory attribute types */

#define OMP_O_DS_A_AUTHORITY_REVOC_LIST   dsP_attributeType(\x26)
#define OMP_O_DS_A_CA_CERT                dsP_attributeType(\x25)
#define OMP_O_DS_A_CERT_REVOC_LIST        dsP_attributeType(\x27)
#define OMP_O_DS_A_CROSS_CERT_PAIR        dsP_attributeType(\x28)
#define OMP_O_DS_A_USER_CERT              dsP_attributeType(\x24)


/* Directory object classes */

#define OMP_O_DS_O_CERT_AUTHORITY         dsP_objectClass(\x10)
#define OMP_O_DS_O_STRONG_AUTHENT_USER    dsP_objectClass(\x0F)
```

```
/* OM class names */

#define OMP_O_DS_C_ALGORITHM_IDENT        dsP_sap_c(\x6\x35)
#define OMP_O_DS_C_CERT                   dsP_sap_c(\x6\x36)
#define OMP_O_DS_C_CERT_LIST              dsP_sap_c(\x6\x37)
#define OMP_O_DS_C_CERT_PAIR              dsP_sap_c(\x6\x38)
#define OMP_O_DS_C_CERT_SUBLIST           dsP_sap_c(\x6\x39)
#define OMP_O_DS_C_SIGNATURE              dsP_sap_c(\x6\x3A)


/* OM attribute names */

#define DS_ALGORITHM            ((OM_type) 821)
#define DS_FORWARD              ((OM_type) 822)
#define DS_ISSUER               ((OM_type) 823)
#define DS_LAST_UPDATE          ((OM_type) 824)
#define DS_ALGORITHM_PARAMETERS ((OM_type) 825)
#define DS_REVERSE              ((OM_type) 826)
#define DS_REVOCATION_DATE      ((OM_type) 827)
#define DS_REVOKED_CERTS        ((OM_type) 828)
#define DS_SERIAL_NUMBER        ((OM_type) 829)
#define DS_SERIAL_NUMBERS       ((OM_type) 830)
#define DS_SIGNATURE            ((OM_type) 831)
#define DS_SIGNATURE_VALUE      ((OM_type) 832)
#define DS_SUBJECT              ((OM_type) 833)
#define DS_SUBJECT_ALGORITHM    ((OM_type) 834)
#define DS_SUBJECT_PUBLIC_KEY   ((OM_type) 835)
#define DS_VALIDITY_NOT_AFTER   ((OM_type) 836)
#define DS_VALIDITY_NOT_BEFORE  ((OM_type) 837)
#define DS_VERSION              ((OM_type) 838)


/* DS_Version */

#define DS_V1988                ((OM_enumeration) 1)

/* Upper bounds on string lengths and the number of repeated OM */
/* attribute values                                             */
```

```
#define DS_VL_LAST_UPDATE                ((OM_value_length) 17)
#define DS_VL_REVOC_DATE                 ((OM_value_length) 17)
#define DS_VL_VALIDITY_NOT_AFTER         ((OM_value_length) 17)
#define DS_VL_VALIDITY_NOT_BEFORE        ((OM_value_length) 17)
#define DS_VN_REVOC_DATE                 ((OM_value_length) 2)

#endif  /* XDSSAP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services.*

xmhp.h

Purpose

Definitions for the MHS directory objects/attributes.

Synopsis

#include <xom.h>#include <xds.h>#include <xdsmdup.h>
#include <xmhp.h>

Description

The **xmhp.h** header defines the constants used by the message handling packages. It is required when using the MHS directory user package. The **xdsmdup.h** header explicitly includes **xmhp.h**.

xmhp.h contains definitions for the X.400 message handling package. Some of these definitions are needed when negotiating use of the MDUP.

The following four message handling classes are referenced:

- **MH_C_G3_FAX_NBPS**
- **MH_C_OR_ADDRESS**
- **MH_C_OR_NAME**
- **MH_C_TELETEX_NBPS**

The only enumerations referenced are **Delivery Mode** and **Terminal Type**. For referenced OM attribute types and OM value lengths see the *DCE 1.2.2 Application Development Guide—Directory Services*.

```
/*
 Note that the identifier for the variable name of type OM_STRING
of a class in the Message Handling package can usually be
```

```
derived using the name of the class, preceded by "MH_C_", and
replacing a blank space with an underscore.  To be in line with the
ANSI C language limitation, some words in the class names are
excepted and are abbreviated as below:

        BILATERAL_INFORMATION is abbreviated to BILATERAL_INFO
        DELIVERED                               DELIV
        CONFIRMATION                            CONFIRM
        CONFIRMATIONS                           CONFIRMS
        PER_RECIPIENT_                          PER_RECIP_
        DELIV_PER_RECIP_REPORT                  DELIV_PER_RECIP_REP
*/

/* BEGIN MH PORTION OF INTERFACE */

/* SYMBOLIC CONSTANTS */

/* Class */

#define OMP_O_MH_C_ALGORITHM            "\x56\x06\x01\x02\x05\x0B\x00"
#define OMP_O_MH_C_ALGORITHM_AND_RESULT "\x56\x06\x01\x02\x05\x0B\x01"
#define OMP_O_MH_C_ASYMMETRIC_TOKEN     "\x56\x06\x01\x02\x05\x0B\x02"
#define OMP_O_MH_C_BILATERAL_INFO       "\x56\x06\x01\x02\x05\x0B\x03"
#define OMP_O_MH_C_COMMUNIQUE           "\x56\x06\x01\x02\x05\x0B\x04"
#define OMP_O_MH_C_CONTENT              "\x56\x06\x01\x02\x05\x0B\x05"
#define OMP_O_MH_C_DELIV_MESSAGE        "\x56\x06\x01\x02\x05\x0B\x06"
#define OMP_O_MH_C_DELIV_PER_RECIP_DR   "\x56\x06\x01\x02\x05\x0B\x07"
#define OMP_O_MH_C_DELIV_PER_RECIP_NDR  "\x56\x06\x01\x02\x05\x0B\x08"
#define OMP_O_MH_C_DELIV_PER_RECIP_REP  "\x56\x06\x01\x02\x05\x0B\x09"
#define OMP_O_MH_C_DELIV_REPORT         "\x56\x06\x01\x02\x05\x0B\x0A"
#define OMP_O_MH_C_DELIVERY_CONFIRM     "\x56\x06\x01\x02\x05\x0B\x0B"
#define OMP_O_MH_C_DELIVERY_ENVELOPE    "\x56\x06\x01\x02\x05\x0B\x0C"
#define OMP_O_MH_C_EITS                 "\x56\x06\x01\x02\x05\x0B\x0D"
#define OMP_O_MH_C_EXPANSION_RECORD     "\x56\x06\x01\x02\x05\x0B\x0E"
#define OMP_O_MH_C_EXTENSIBLE_OBJECT    "\x56\x06\x01\x02\x05\x0B\x0F"
#define OMP_O_MH_C_EXTENSION            "\x56\x06\x01\x02\x05\x0B\x10"
#define OMP_O_MH_C_EXTERNAL_TRACE_ENTRY "\x56\x06\x01\x02\x05\x0B\x11"
#define OMP_O_MH_C_G3_FAX_NBPS          "\x56\x06\x01\x02\x05\x0B\x12"
#define OMP_O_MH_C_GENERAL_CONTENT      "\x56\x06\x01\x02\x05\x0B\x13"
#define OMP_O_MH_C_INTERNAL_TRACE_ENTRY "\x56\x06\x01\x02\x05\x0B\x14"
```

```
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRM  "\x56\x06\x01\x02\x05\x0B\x15"
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRMS "\x56\x06\x01\x02\x05\x0B\x16"
#define OMP_O_MH_C_LOCAL_NDR            "\x56\x06\x01\x02\x05\x0B\x17"
#define OMP_O_MH_C_LOCAL_PER_RECIP_NDR  "\x56\x06\x01\x02\x05\x0B\x18"
#define OMP_O_MH_C_MESSAGE              "\x56\x06\x01\x02\x05\x0B\x19"
#define OMP_O_MH_C_MESSAGE_RD           "\x56\x06\x01\x02\x05\x0B\x1A"
#define OMP_O_MH_C_MTS_IDENTIFIER       "\x56\x06\x01\x02\x05\x0B\x1B"
#define OMP_O_MH_C_OR_ADDRESS           "\x56\x06\x01\x02\x05\x0B\x1C"
#define OMP_O_MH_C_OR_NAME              "\x56\x06\x01\x02\x05\x0B\x1D"
#define OMP_O_MH_C_PER_RECIP_DR         "\x56\x06\x01\x02\x05\x0B\x1E"
#define OMP_O_MH_C_PER_RECIP_NDR        "\x56\x06\x01\x02\x05\x0B\x1F"
#define OMP_O_MH_C_PER_RECIP_REPORT     "\x56\x06\x01\x02\x05\x0B\x20"
#define OMP_O_MH_C_PROBE                "\x56\x06\x01\x02\x05\x0B\x21"
#define OMP_O_MH_C_PROBE_RD             "\x56\x06\x01\x02\x05\x0B\x22"
#define OMP_O_MH_C_RD                   "\x56\x06\x01\x02\x05\x0B\x23"
#define OMP_O_MH_C_REDIRECTION_RECORD   "\x56\x06\x01\x02\x05\x0B\x24"
#define OMP_O_MH_C_REPORT               "\x56\x06\x01\x02\x05\x0B\x25"
#define OMP_O_MH_C_SECURITY_LABEL       "\x56\x06\x01\x02\x05\x0B\x26"
#define OMP_O_MH_C_SESSION              "\x56\x06\x01\x02\x05\x0B\x27"
#define OMP_O_MH_C_SUBMISSION_RESULTS   "\x56\x06\x01\x02\x05\x0B\x28"
#define OMP_O_MH_C_SUBMITTED_COMMUNIQUE "\x56\x06\x01\x02\x05\x0B\x29"
#define OMP_O_MH_C_SUBMITTED_MESSAGE    "\x56\x06\x01\x02\x05\x0B\x2A"
#define OMP_O_MH_C_SUBMITTED_MESSAGE_RD "\x56\x06\x01\x02\x05\x0B\x2B"
#define OMP_O_MH_C_SUBMITTED_PROBE      "\x56\x06\x01\x02\x05\x0B\x2C"
#define OMP_O_MH_C_SUBMITTED_PROBE_RD   "\x56\x06\x01\x02\x05\x0B\x2D"
#define OMP_O_MH_C_TELETEX_NBPS         "\x56\x06\x01\x02\x05\x0B\x2E"
#define OMP_O_MH_C_DELIVERY_REPORT      "\x56\x06\x01\x02\x05\x0B\x2F"
#define OMP_O_MH_C_MT_PUBLIC_DATA       "\x56\x06\x01\x02\x05\x0B\x30"
#define OMP_O_MH_C_TOKEN_PUBLIC_DATA    "\x56\x06\x01\x02\x05\x0B\x31"


/* Enumeration */

/* Action */
#define MH_AC_EXPANDED          ( (OM_enumeration) -2 )
#define MH_AC_REDIRECTED        ( (OM_enumeration) -1 )
#define MH_AC_RELAYED           ( (OM_enumeration) 0 )
#define MH_AC_REROUTED          ( (OM_enumeration) 1 )

/* Builtin EIT */
```

```
#define MH_BE_UNDEFINED         ( (OM_enumeration) 0 )
#define MH_BE_TELEX             ( (OM_enumeration) 1 )
#define MH_BE_IA5_TEXT          ( (OM_enumeration) 2 )
#define MH_BE_G3_FAX            ( (OM_enumeration) 3 )
#define MH_BE_G4_CLASS1         ( (OM_enumeration) 4 )
#define MH_BE_TELETEX           ( (OM_enumeration) 5 )
#define MH_BE_VIDEOTEX          ( (OM_enumeration) 6 )
#define MH_BE_MIXED_MODE        ( (OM_enumeration) 9 )
#define MH_BE_ODA               ( (OM_enumeration) 10 )
#define MH_BE_ISO_6937_TEXT     ( (OM_enumeration) 11 )

/* Delivery Mode */
#define MH_DM_ANY               ( (OM_enumeration) 0 )
#define MH_DM_MTS               ( (OM_enumeration) 1 )
#define MH_DM_PDS               ( (OM_enumeration) 2 )
#define MH_DM_TELEX             ( (OM_enumeration) 3 )
#define MH_DM_TELETEX           ( (OM_enumeration) 4 )
#define MH_DM_G3_FAX            ( (OM_enumeration) 5 )
#define MH_DM_G4_FAX            ( (OM_enumeration) 6 )
#define MH_DM_IA5_TERMINAL      ( (OM_enumeration) 7 )
#define MH_DM_VIDEOTEX          ( (OM_enumeration) 8 )
#define MH_DM_TELEPHONE         ( (OM_enumeration) 9 )

/* Delivery Point */
#define MH_DP_PUBLIC_UA         ( (OM_enumeration) 0 )
#define MH_DP_PRIVATE_UA        ( (OM_enumeration) 1 )
#define MH_DP_MS                ( (OM_enumeration) 2 )
#define MH_DP_DL                ( (OM_enumeration) 3 )
#define MH_DP_PDAU              ( (OM_enumeration) 4 )
#define MH_DP_PDS_PATRON        ( (OM_enumeration) 5 )
#define MH_DP_OTHER_AU          ( (OM_enumeration) 6 )

/* Diagnostic */

#define MH_DG_NO_DIAGNOSTIC                     ( (OM_enumeration) -1 )
#define MH_DG_OR_NAME_UNRECOGNIZED              ( (OM_enumeration) 0 )
#define MH_DG_OR_NAME_AMBIGUOUS                 ( (OM_enumeration) 1 )
#define MH_DG_MTS_CONGESTED                     ( (OM_enumeration) 2 )
#define MH_DG_LOOP_DETECTED                     ( (OM_enumeration) 3 )
#define MH_DG_RECIPIENT_UNAVAILABLE             ( (OM_enumeration) 4 )
```

```
#define MH_DG_MAXIMUM_TIME_EXPIRED            ( (OM_enumeration) 5 )
#define MH_DG_EITS_UNSUPPORTED                ( (OM_enumeration) 6 )
#define MH_DG_CONTENT_TOO_LONG                ( (OM_enumeration) 7 )
#define MH_DG_IMPRACTICAL_TO_CONVERT          ( (OM_enumeration) 8 )
#define MH_DG_PROHIBITED_TO_CONVERT           ( (OM_enumeration) 9 )
#define MH_DG_CONVERSION_UNSUBSCRIBED         ( (OM_enumeration) 10 )
#define MH_DG_PARAMETERS_INVALID              ( (OM_enumeration) 11 )
#define MH_DG_CONTENT_SYNTAX_IN_ERROR         ( (OM_enumeration) 12 )
#define MH_DG_LENGTH_CONSTRAINT_VIOLATD       ( (OM_enumeration) 13 )
#define MH_DG_NUMBER_CONSTRAINT_VIOLATD       ( (OM_enumeration) 14 )
#define MH_DG_CONTENT_TYPE_UNSUPPORTED        ( (OM_enumeration) 15 )
#define MH_DG_TOO_MANY_RECIPIENTS             ( (OM_enumeration) 16 )
#define MH_DG_NO_BILATERAL_AGREEMENT          ( (OM_enumeration) 17 )
#define MH_DG_CRITICAL_FUNC_UNSUPPORTED       ( (OM_enumeration) 18 )
#define MH_DG_CONVERSION_LOSS_PROHIB          ( (OM_enumeration) 19 )
#define MH_DG_LINE_TOO_LONG                   ( (OM_enumeration) 20 )
#define MH_DG_PAGE_TOO_LONG                   ( (OM_enumeration) 21 )
#define MH_DG_PICTORIAL_SYMBOL_LOST           ( (OM_enumeration) 22 )
#define MH_DG_PUNCTUATION_SYMBOL_LOST         ( (OM_enumeration) 23 )
#define MH_DG_ALPHABETIC_CHARACTER_LOST       ( (OM_enumeration) 24 )
#define MH_DG_MULTIPLE_INFO_LOSSES            ( (OM_enumeration) 25 )
#define MH_DG_REASSIGNMENT_PROHIBITED         ( (OM_enumeration) 26 )
#define MH_DG_REDIRECTION_LOOP_DETECTED       ( (OM_enumeration) 27 )
#define MH_DG_EXPANSION_PROHIBITED            ( (OM_enumeration) 28 )
#define MH_DG_SUBMISSION_PROHIBITED           ( (OM_enumeration) 29 )
#define MH_DG_EXPANSION_FAILED                ( (OM_enumeration) 30 )
#define MH_DG_RENDITION_UNSUPPORTED           ( (OM_enumeration) 31 )
#define MH_DG_MAIL_ADDRESS_INCORRECT          ( (OM_enumeration) 32 )
#define MH_DG_MAIL_OFFICE_INCOR_OR_INVD       ( (OM_enumeration) 33 )
#define MH_DG_MAIL_ADDRESS_INCOMPLETE         ( (OM_enumeration) 34 )
#define MH_DG_MAIL_RECIPIENT_UNKNOWN          ( (OM_enumeration) 35 )
#define MH_DG_MAIL_RECIPIENT_DECEASED         ( (OM_enumeration) 36 )
#define MH_DG_MAIL_ORGANIZATION_EXPIRED       ( (OM_enumeration) 37 )
#define MH_DG_MAIL_REFUSED                    ( (OM_enumeration) 38 )
#define MH_DG_MAIL_UNCLAIMED                  ( (OM_enumeration) 39 )
#define MH_DG_MAIL_RECIPIENT_MOVED            ( (OM_enumeration) 40 )
#define MH_DG_MAIL_RECIPIENT_TRAVELLING       ( (OM_enumeration) 41 )
#define MH_DG_MAIL_RECIPIENT_DEPARTED         ( (OM_enumeration) 42 )
#define MH_DG_MAIL_NEW_ADDRESS_UNKNOWN        ( (OM_enumeration) 43 )
#define MH_DG_MAIL_FORWARDING_UNWANTED        ( (OM_enumeration) 44 )
```

```
#define MH_DG_MAIL_FORWARDING_PROHIB            ( (OM_enumeration) 45 )
#define MH_DG_SECURE_MESSAGING_ERROR            ( (OM_enumeration) 46 )
#define MH_DG_DOWNGRADING_IMPOSSIBLE            ( (OM_enumeration) 47 )

/* Explicit Conversion */

#define MH_EC_NO_CONVERSION             ( (OM_enumeration) -1 )
#define MH_EC_IA5_TEXT_TO_TELETEX       ( (OM_enumeration) 0 )
#define MH_EC_TELETEX_TO_TELEX          ( (OM_enumeration) 1 )
#define MH_EC_TELEX_TO_IA5_TEXT         ( (OM_enumeration) 2 )
#define MH_EC_TELEX_TO_TELETEX          ( (OM_enumeration) 3 )
#define MH_EC_TELEX_TO_G4_CLASS1        ( (OM_enumeration) 4 )
#define MH_EC_TELEX_TO_VIDEOTEX         ( (OM_enumeration) 5 )
#define MH_EC_IA5_TEXT_TO_TELEX         ( (OM_enumeration) 6 )
#define MH_EC_TELEX_TO_G3_FAX           ( (OM_enumeration) 7 )
#define MH_EC_IA5_TEXT_TO_G3_FAX        ( (OM_enumeration) 8 )
#define MH_EC_IA5_TEXT_TO_G4_CLASS1     ( (OM_enumeration) 9 )
#define MH_EC_IA5_TEXT_TO_VIDEOTEX      ( (OM_enumeration) 10 )
#define MH_EC_TELETEX_TO_IA5_TEXT       ( (OM_enumeration) 11 )
#define MH_EC_TELETEX_TO_G3_FAX         ( (OM_enumeration) 12 )
#define MH_EC_TELETEX_TO_G4_CLASS1      ( (OM_enumeration) 13 )
#define MH_EC_TELETEX_TO_VIDEOTEX       ( (OM_enumeration) 14 )
#define MH_EC_VIDEOTEX_TO_TELEX         ( (OM_enumeration) 15 )
#define MH_EC_VIDEOTEX_TO_IA5_TEXT      ( (OM_enumeration) 16 )
#define MH_EC_VIDEOTEX_TO_TELETEX       ( (OM_enumeration) 17 )

/* Postal Mode */

#define MH_PM_ORDINARY_MAIL             ( (OM_enumeration) 0 )
#define MH_PM_SPECIAL_DELIVERY          ( (OM_enumeration) 1 )
#define MH_PM_EXPRESS_MAIL              ( (OM_enumeration) 2 )
#define MH_PM_CC                        ( (OM_enumeration) 3 )
#define MH_PM_CC_WITH_TELEPHONE_ADVICE  ( (OM_enumeration) 4 )
#define MH_PM_CC_WITH_TELEX_ADVICE      ( (OM_enumeration) 5 )
#define MH_PM_CC_WITH_TELETEX_ADVICE    ( (OM_enumeration) 6 )

/* Postal Report */

#define MH_PR_UNDELIVBLE_MAIL_VIA_PDS   ( (OM_enumeration) 0 )
#define MH_PR_NOTIFICN_VIA_PDS          ( (OM_enumeration) 1 )
```

```
#define MH_PR_NOTIFICN_VIA_MTS          ( (OM_enumeration) 2 )
#define MH_PR_NOTIFICN_VIA_MTS_AND_PDS  ( (OM_enumeration) 3 )

/* Priority */

#define MH_PTY_NORMAL   ( (OM_enumeration) 0 )
#define MH_PTY_LOW      ( (OM_enumeration) 1 )
#define MH_PTY_URGENT   ( (OM_enumeration) 2 )

/* Reason */

#define MH_RE_TRANSFER_FAILED               ( (OM_enumeration) 0 )
#define MH_RE_TRANSFER_IMPOSSIBLE           ( (OM_enumeration) 1 )
#define MH_RE_CONVERSION_NOT_PERFORMED      ( (OM_enumeration) 2 )
#define MH_RE_PHYSICAL_RENDITN_NOT_DONE     ( (OM_enumeration) 3 )
#define MH_RE_PHYSICAL_DELIV_NOT_DONE       ( (OM_enumeration) 4 )
#define MH_RE_RESTRICTED_DELIVERY           ( (OM_enumeration) 5 )
#define MH_RE_DIRECTORY_OPERATN_FAILED      ( (OM_enumeration) 6 )

/* Redirection Reason */

#define MH_RR_RECIPIENT_ASSIGNED               ( (OM_enumeration) 0 )
#define MH_RR_ORIGINATOR_REQUESTED             ( (OM_enumeration) 1 )
#define MH_RR_RECIPIENT_DOMAIN_ASSIGNED        ( (OM_enumeration) 2 )

/* Registration */

#define MH_RG_UNREGISTERED_MAIL                ( (OM_enumeration) 0 )
#define MH_RG_REGISTERED_MAIL                  ( (OM_enumeration) 1 )
#define MH_RG_REGISTERED_MAIL_IN_PERSON        ( (OM_enumeration) 2 )

/* Report Request */

#define MH_RQ_NEVER            ( (OM_enumeration) 0 )
#define MH_RQ_NON_DELIVERY     ( (OM_enumeration) 1 )
#define MH_RQ_ALWAYS           ( (OM_enumeration) 2 )
#define MH_RQ_ALWAYS_AUDITED   ( (OM_enumeration) 3 )

/* Security Classification */
```

```
#define MH_SC_UNMARKED          ( (OM_enumeration) 0 )
#define MH_SC_UNCLASSIFIED      ( (OM_enumeration) 1 )
#define MH_SC_RESTRICTED        ( (OM_enumeration) 2 )
#define MH_SC_CONFIDENTIAL      ( (OM_enumeration) 3 )
#define MH_SC_SECRET            ( (OM_enumeration) 4 )
#define MH_SC_TOP_SECRET        ( (OM_enumeration) 5 )

/* Terminal Type */

#define MH_TT_TELEX             ( (OM_enumeration) 3 )
#define MH_TT_TELETEX           ( (OM_enumeration) 4 )
#define MH_TT_G3_FAX            ( (OM_enumeration) 5 )
#define MH_TT_G4_FAX            ( (OM_enumeration) 6 )
#define MH_TT_IA5_TERMINAL      ( (OM_enumeration) 7 )
#define MH_TT_VIDEOTEX          ( (OM_enumeration) 8 )


/* Integer */

/* Content Type */

#define MH_CTI_UNIDENTIFIED     ( (OM_integer) 0 )
#define MH_CTI_EXTERNAL         ( (OM_integer) 1 )
#define MH_CTI_P2_1984          ( (OM_integer) 2 )
#define MH_CTI_P2_1988          ( (OM_integer) 22 )


/* Object Identifier (Elements component) */

/* Content Type */
#define OMP_O_MH_CTO_INNER_MESSAGE      "\x56\x03\x03\x01"
#define OMP_O_MH_CTO_UNIDENTIFIED       "\x56\x03\x03\x00"

/* External EITs */
#define OMP_O_MH_EE_G3_FAX              "\x56\x03\x04\x03"
#define OMP_O_MH_EE_G4_CLASS_1          "\x56\x03\x04\x04"
#define OMP_O_MH_EE_IA5_TEXT            "\x56\x03\x04\x02"
#define OMP_O_MH_EE_MIXED_MODE          "\x56\x03\x04\x09"
#define OMP_O_MH_EE_TELETEX             "\x56\x03\x04\x05"
#define OMP_O_MH_EE_TELEX               "\x56\x03\x04\x01"
```

```
#define OMP_O_MH_EE_UNDEFINED           "\x56\x03\x04\x00"
#define OMP_O_MH_EE_VIDEOTEX            "\x56\x03\x04\x06"

/* Rendition Attributes */
#define OMP_O_MH_RA_BASIC_RENDITION     "\x56\x03\x05\x00"

/* Type */

#define MH_T_A3_WIDTH                           ( (OM_type) 200 )
#define MH_T_ACTION                             ( (OM_type) 201 )
#define MH_T_ACTUAL_RECIPIENT_NAME              ( (OM_type) 202 )
#define MH_T_ADMD_NAME                          ( (OM_type) 203 )
#define MH_T_ALGORITHM_DATUM                    ( (OM_type) 204 )
#define MH_T_ALGORITHM_ID                       ( (OM_type) 205 )
#define MH_T_ALGORITHM_RESULT                   ( (OM_type) 206 )
#define MH_T_ALTERNATE_RECIP_ALLOWED            ( (OM_type) 207 )
#define MH_T_ALTERNATE_RECIPIENT_NAME           ( (OM_type) 208 )
#define MH_T_ARRIVAL_TIME                       ( (OM_type) 209 )
#define MH_T_ATTEMPTED_ADMD_NAME                ( (OM_type) 210 )
#define MH_T_ATTEMPTED_COUNTRY_NAME             ( (OM_type) 211 )
#define MH_T_ATTEMPTED_MTA_NAME                 ( (OM_type) 212 )
#define MH_T_ATTEMPTED_PRMD_IDENTIFIER          ( (OM_type) 213 )
#define MH_T_B4_LENGTH                          ( (OM_type) 214 )
#define MH_T_B4_WIDTH                           ( (OM_type) 215 )
#define MH_T_BILATERAL_INFO                     ( (OM_type) 216 )
#define MH_T_BINARY_CONTENT                     ( (OM_type) 217 )
#define MH_T_BUILTIN_EITS                       ( (OM_type) 218 )
#define MH_T_BUREAU_FAX_DELIVERY                ( (OM_type) 219 )
#define MH_T_COMMON_NAME                        ( (OM_type) 220 )
#define MH_T_CONFIDENTIALITY_ALGORITHM          ( (OM_type) 221 )
#define MH_T_CONFIDENTIALITY_KEY                ( (OM_type) 222 )
#define MH_T_CONTENT                            ( (OM_type) 223 )
#define MH_T_CONTENT_CORRELATOR                 ( (OM_type) 224 )
#define MH_T_CONTENT_EXTENSIONS                 ( (OM_type) 225 )
#define MH_T_CONTENT_IDENTIFIER                 ( (OM_type) 226 )
#define MH_T_CONTENT_LENGTH                     ( (OM_type) 227 )
#define MH_T_CONTENT_RETURN_REQUESTED           ( (OM_type) 228 )
#define MH_T_CONTENT_TYPE                       ( (OM_type) 229 )
#define MH_T_CONTROL_CHARACTER_SETS             ( (OM_type) 230 )
#define MH_T_CONVERSION_LOSS_PROHIBITED         ( (OM_type) 231 )
```

```
#define MH_T_CONVERSION_PROHIBITED            ( (OM_type) 232 )
#define MH_T_CONVERTED_EITS                   ( (OM_type) 233 )
#define MH_T_COUNTRY_NAME                     ( (OM_type) 234 )
#define MH_T_CRITICAL_FOR_DELIVERY            ( (OM_type) 235 )
#define MH_T_CRITICAL_FOR_SUBMISSION          ( (OM_type) 236 )
#define MH_T_CRITICAL_FOR_TRANSFER            ( (OM_type) 237 )
#define MH_T_DEFERRED_DELIVERY_TIME           ( (OM_type) 238 )
#define MH_T_DEFERRED_TIME                    ( (OM_type) 239 )
#define MH_T_DELIVERY_CONFIRMS                ( (OM_type) 240 )
#define MH_T_DELIVERY_POINT                   ( (OM_type) 241 )
#define MH_T_DELIVERY_TIME                    ( (OM_type) 242 )
#define MH_T_DIRECTORY_NAME                   ( (OM_type) 243 )
#define MH_T_DISCLOSURE_ALLOWED               ( (OM_type) 244 )
#define MH_T_DISTINGUISHED_RECIP_ADDR         ( (OM_type) 245 )
#define MH_T_DOMAIN_TYPE_1                    ( (OM_type) 246 )
#define MH_T_DOMAIN_TYPE_2                    ( (OM_type) 247 )
#define MH_T_DOMAIN_TYPE_3                    ( (OM_type) 248 )
#define MH_T_DOMAIN_TYPE_4                    ( (OM_type) 249 )
#define MH_T_DOMAIN_VALUE_1                   ( (OM_type) 250 )
#define MH_T_DOMAIN_VALUE_2                   ( (OM_type) 251 )
#define MH_T_DOMAIN_VALUE_3                   ( (OM_type) 252 )
#define MH_T_DOMAIN_VALUE_4                   ( (OM_type) 253 )
#define MH_T_ENVELOPES                        ( (OM_type) 254 )
#define MH_T_EVENT_HANDLE                     ( (OM_type) 255 )
#define MH_T_EXPANSION_HISTORY                ( (OM_type) 256 )
#define MH_T_EXPANSION_PROHIBITED             ( (OM_type) 257 )
#define MH_T_EXPLICIT_CONVERSION              ( (OM_type) 258 )
#define MH_T_EXTENSION_TYPE                   ( (OM_type) 259 )
#define MH_T_EXTENSION_VALUE                  ( (OM_type) 260 )
#define MH_T_EXTENSIONS                       ( (OM_type) 261 )
#define MH_T_EXTERNAL_EITS                    ( (OM_type) 262 )
#define MH_T_EXTERNAL_TRACE_INFO              ( (OM_type) 263 )
#define MH_T_FINE_RESOLUTION                  ( (OM_type) 264 )
#define MH_T_FORWARDING_ADDRESS               ( (OM_type) 265 )
#define MH_T_FORWARDING_ADDR_REQUESTED        ( (OM_type) 266 )
#define MH_T_FORWARDING_PROHIBITED            ( (OM_type) 267 )
#define MH_T_G3_FAX_NBPS                      ( (OM_type) 268 )
#define MH_T_G4_FAX_NBPS                      ( (OM_type) 269 )
#define MH_T_GENERATION                       ( (OM_type) 270 )
#define MH_T_GIVEN_NAME                       ( (OM_type) 271 )
```

```
#define MH_T_GRAPHIC_CHARACTER_SETS           ( (OM_type) 272 )
#define MH_T_INFORMATION                      ( (OM_type) 273 )
#define MH_T_INITIALS                         ( (OM_type) 274 )
#define MH_T_INTEGRITY_CHECK                  ( (OM_type) 275 )
#define MH_T_INTENDED_RECIPIENT_NAME          ( (OM_type) 276 )
#define MH_T_INTENDED_RECIPIENT_NUMBER        ( (OM_type) 277 )
#define MH_T_INTERNAL_TRACE_INFO              ( (OM_type) 278 )
#define MH_T_ISDN_NUMBER                      ( (OM_type) 279 )
#define MH_T_ISDN_SUBADDRESS                  ( (OM_type) 280 )
#define MH_T_LATEST_DELIVERY_TIME             ( (OM_type) 281 )
#define MH_T_LOCAL_IDENTIFIER                 ( (OM_type) 282 )
#define MH_T_MESSAGE_SEQUENCE_NUMBER          ( (OM_type) 283 )
#define MH_T_MISCELANEOUS_CAPABILITIES        ( (OM_type) 284 )
#define MH_T_MTA_CERTIFICATE                  ( (OM_type) 285 )
#define MH_T_MTA_NAME                         ( (OM_type) 286 )
#define MH_T_MTA_REPORT_REQUEST               ( (OM_type) 287 )
#define MH_T_MTA_RESPONSIBILITY               ( (OM_type) 288 )
#define MH_T_MTS_IDENTIFIER                   ( (OM_type) 289 )
#define MH_T_NAME                             ( (OM_type) 290 )
#define MH_T_NON_DELIVERY_DIAGNOSTIC          ( (OM_type) 291 )
#define MH_T_NON_DELIVERY_REASON              ( (OM_type) 292 )
#define MH_T_NUMERIC_USER_IDENTIFIER          ( (OM_type) 293 )
#define MH_T_ORGANIZATION_NAME                ( (OM_type) 294 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_1       ( (OM_type) 295 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_2       ( (OM_type) 296 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_3       ( (OM_type) 297 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_4       ( (OM_type) 298 )
#define MH_T_ORIG_AND_EXPANSION_HISTORY       ( (OM_type) 299 )
#define MH_T_ORIGIN_CHECK                     ( (OM_type) 300 )
#define MH_T_ORIGINAL_EITS                    ( (OM_type) 301 )
#define MH_T_ORIGINALLY_INTENDED_RECIP        ( (OM_type) 302 )
#define MH_T_ORIGINATOR_CERTIFICATE           ( (OM_type) 303 )
#define MH_T_ORIGINATOR_NAME                  ( (OM_type) 304 )
#define MH_T_ORIGINATOR_REPORT_REQUEST        ( (OM_type) 305 )
#define MH_T_ORIGINATOR_RETURN_ADDRESS        ( (OM_type) 306 )
#define MH_T_OTHER_RECIPIENT_NAMES            ( (OM_type) 307 )
#define MH_T_PAGE_FORMATS                     ( (OM_type) 308 )
#define MH_T_PER_RECIP_REPORTS                ( (OM_type) 309 )
#define MH_T_POSTAL_ADDRESS_DETAILS           ( (OM_type) 310 )
#define MH_T_POSTAL_ADDRESS_IN_FULL           ( (OM_type) 311 )
```

```
#define MH_T_POSTAL_ADDRESS_IN_LINES          ( (OM_type) 312 )
#define MH_T_POSTAL_CODE                      ( (OM_type) 313 )
#define MH_T_POSTAL_COUNTRY_NAME              ( (OM_type) 314 )
#define MH_T_POSTAL_DELIVERY_POINT_NAME       ( (OM_type) 315 )
#define MH_T_POSTAL_DELIV_SYSTEM_NAME         ( (OM_type) 316 )
#define MH_T_POSTAL_GENERAL_DELIV_ADDR        ( (OM_type) 317 )
#define MH_T_POSTAL_LOCALE                    ( (OM_type) 318 )
#define MH_T_POSTAL_MODE                      ( (OM_type) 319 )
#define MH_T_POSTAL_OFFICE_BOX_NUMBER         ( (OM_type) 320 )
#define MH_T_POSTAL_OFFICE_NAME               ( (OM_type) 321 )
#define MH_T_POSTAL_OFFICE_NUMBER             ( (OM_type) 322 )
#define MH_T_POSTAL_ORGANIZATION_NAME         ( (OM_type) 323 )
#define MH_T_POSTAL_PATRON_DETAILS            ( (OM_type) 324 )
#define MH_T_POSTAL_PATRON_NAME               ( (OM_type) 325 )
#define MH_T_POSTAL_REPORT                    ( (OM_type) 326 )
#define MH_T_POSTAL_STREET_ADDRESS            ( (OM_type) 327 )
#define MH_T_PREFERRED_DELIVERY_MODES         ( (OM_type) 328 )
#define MH_T_PRESENTATION_ADDRESS             ( (OM_type) 329 )
#define MH_T_PRIORITY                         ( (OM_type) 330 )
#define MH_T_PRIVACY_MARK                     ( (OM_type) 331 )
#define MH_T_PRIVATE_USE                      ( (OM_type) 332 )
#define MH_T_PRMD_IDENTIFIER                  ( (OM_type) 333 )
#define MH_T_PRMD_NAME                        ( (OM_type) 334 )
#define MH_T_PROOF_OF_DELIVERY                ( (OM_type) 335 )
#define MH_T_PROOF_OF_DELIV_REQUESTED         ( (OM_type) 336 )
#define MH_T_PROOF_OF_SUBMISSION              ( (OM_type) 337 )
#define MH_T_PROOF_OF_SUBMISN_REQUEST         ( (OM_type) 338 )
#define MH_T_PUBLIC_INFORMATION               ( (OM_type) 339 )
#define MH_T_RANDOM_NUMBER                    ( (OM_type) 340 )
#define MH_T_REASON                           ( (OM_type) 341 )
#define MH_T_REASSIGNMENT_PROHIBITED          ( (OM_type) 342 )
#define MH_T_RECIPIENT_CERTIFICATE            ( (OM_type) 343 )
#define MH_T_RECIPIENT_DESCRIPTORS            ( (OM_type) 344 )
#define MH_T_RECIPIENT_NAME                   ( (OM_type) 345 )
#define MH_T_RECIPIENT_NUMBER                 ( (OM_type) 346 )
#define MH_T_RECIP_NUMBER_FOR_ADVICE          ( (OM_type) 347 )
#define MH_T_REDIRECTION_HISTORY              ( (OM_type) 348 )
#define MH_T_REGISTRATION                     ( (OM_type) 349 )
#define MH_T_RENDITION_ATTRIBUTES             ( (OM_type) 350 )
#define MH_T_REPORT_ADDITIONAL_INFO           ( (OM_type) 351 )
```

```
#define MH_T_REPORT_DESTINATION                ( (OM_type) 352 )
#define MH_T_REPORTING_DL_NAME                 ( (OM_type) 353 )
#define MH_T_REPORTING_MTA_CERTIFICATE         ( (OM_type) 354 )
#define MH_T_SECRET_INFORMATION                ( (OM_type) 355 )
#define MH_T_SECURITY_CATEGORY_DATA            ( (OM_type) 356 )
#define MH_T_SECURITY_CATEGORY_IDS             ( (OM_type) 357 )
#define MH_T_SECURITY_CLASSIFICATION           ( (OM_type) 358 )
#define MH_T_SECURITY_LABEL                    ( (OM_type) 359 )
#define MH_T_SECURITY_POLICY_ID                ( (OM_type) 360 )
#define MH_T_SIGNATURE                         ( (OM_type) 361 )
#define MH_T_SUBJECT_EXT_TRACE_INFO            ( (OM_type) 362 )
#define MH_T_SUBJECT_MTS_IDENTIFIER            ( (OM_type) 363 )
#define MH_T_SUBMISSION_TIME                   ( (OM_type) 364 )
#define MH_T_SUPPLEMENTARY_INFO                ( (OM_type) 365 )
#define MH_T_SURNAME                           ( (OM_type) 366 )
#define MH_T_TELETEX_NBPS                      ( (OM_type) 367 )
#define MH_T_TEMPORARY                         ( (OM_type) 368 )
#define MH_T_TERMINAL_IDENTIFIER               ( (OM_type) 369 )
#define MH_T_TERMINAL_TYPE                     ( (OM_type) 370 )
#define MH_T_TIME                              ( (OM_type) 371 )
#define MH_T_TOKEN                             ( (OM_type) 372 )
#define MH_T_TWO_DIMENSIONAL                   ( (OM_type) 373 )
#define MH_T_UNCOMPRESSED                      ( (OM_type) 374 )
#define MH_T_UNLIMITED_LENGTH                  ( (OM_type) 375 )
#define MH_T_WORKSPACE                         ( (OM_type) 376 )
#define MH_T_X121_ADDRESS                      ( (OM_type) 377 )

/* Value Length */

#define MH_VL_ADMD_NAME                     ( (OM_value_length) 16 )
#define MH_VL_ATTEMPTED_ADMD_NAME           ( (OM_value_length) 16 )
#define MH_VL_ATTEMPTED_COUNTRY_NAME        ( (OM_value_length) 3 )
#define MH_VL_ATTEMPTED_PRMD_IDENTIFIER     ( (OM_value_length) 16 )
#define MH_VL_COMMON_NAME                   ( (OM_value_length) 64 )
#define MH_VL_CONTENT_CORRELATOR            ( (OM_value_length) 512 )
#define MH_VL_CONTENT_IDENTIFIER            ( (OM_value_length) 16 )
#define MH_VL_COUNTRY_NAME                  ( (OM_value_length) 3 )
#define MH_VL_DOMAIN_TYPE                   ( (OM_value_length) 8 )
#define MH_VL_DOMAIN_VALUE                  ( (OM_value_length) 128 )
#define MH_VL_GENERATION                    ( (OM_value_length) 3 )
```

```
#define MH_VL_GIVEN_NAME                    ( (OM_value_length) 16 )
#define MH_VL_INFORMATION                   ( (OM_value_length) 1024 )
#define MH_VL_INITIALS                      ( (OM_value_length) 5 )
#define MH_VL_ISDN_NUMBER                   ( (OM_value_length) 15 )
#define MH_VL_ISDN_SUBADDRESS               ( (OM_value_length) 40 )
#define MH_VL_LATEST_DELIVERY_TIME          ( (OM_value_length) 7 )
#define MH_VL_LOCAL_IDENTIFIER              ( (OM_value_length) 32 )
#define MH_VL_MSG_CONTENT_CORRELATOR        ( (OM_value_length) 16 )
#define MH_VL_MTA_NAME                      ( (OM_value_length) 32 )
#define MH_VL_NUMERIC_USER_IDENTIFIER       ( (OM_value_length) 32 )
#define MH_VL_ORGANIZATION_NAME             ( (OM_value_length) 64 )
#define MH_VL_ORGANIZATIONAL_UNIT_NAMES     ( (OM_value_length) 32 )
#define MH_VL_POSTAL_ADDRESS_DETAILS        ( (OM_value_length) 30 )
#define MH_VL_POSTAL_ADDRESS_IN_FULL        ( (OM_value_length) 185 )
#define MH_VL_POSTAL_CODE                   ( (OM_value_length) 16 )
#define MH_VL_POSTAL_COUNTRY_NAME           ( (OM_value_length) 32 )
#define MH_VL_POSTAL_DELIV_POINT_NAME       ( (OM_value_length) 30 )
#define MH_VL_POSTAL_DELIV_SYSTEM_NAME      ( (OM_value_length) 16 )
#define MH_VL_POSTAL_GENERAL_DELIV_ADDR     ( (OM_value_length) 30 )
#define MH_VL_POSTAL_LOCALE                 ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_BOX_NUMBER      ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_NAME            ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_NUMBER          ( (OM_value_length) 30 )
#define MH_VL_POSTAL_ORGANIZATION_NAME      ( (OM_value_length) 30 )
#define MH_VL_POSTAL_PATRON_DETAILS         ( (OM_value_length) 30 )
#define MH_VL_POSTAL_PATRON_NAME            ( (OM_value_length) 30 )
#define MH_VL_POSTAL_STREET_ADDRESS         ( (OM_value_length) 30 )
#define MH_VL_PRIVACY_MARK                  ( (OM_value_length) 128 )
#define MH_VL_PRIVATE_USE                   ( (OM_value_length) 126 )
#define MH_VL_PRMD_IDENTIFIER               ( (OM_value_length) 16 )
#define MH_VL_PRMD_NAME                     ( (OM_value_length) 16 )
#define MH_VL_RECIP_NUMBER_FOR_ADVICE       ( (OM_value_length) 32 )
#define MH_VL_REDIRECTION_TIME              ( (OM_value_length) 7 )
#define MH_VL_REPORT_ADDITIONAL_INFO        ( (OM_value_length) 1024 )
#define MH_VL_SUPPLEMENTARY_INFO            ( (OM_value_length) 64 )
#define MH_VL_SURNAME                       ( (OM_value_length) 40 )
#define MH_VL_TERMINAL_IDENTIFIER           ( (OM_value_length) 24 )
#define MH_VL_TIME                          ( (OM_value_length) 17 )
#define MH_VL_X121_ADDRESS                  ( (OM_value_length) 15 )
```

```
/* Value Number */

#define MH_VN_BILATERAL_INFORMATION          ( (OM_value_number) 8 )
#define MH_VN_ENCODED_INFORMATION_TYPES      ( (OM_value_number) 8 )
#define MH_VN_EXPANSION_HISTORY              ( (OM_value_number) 512 )
#define MH_VN_OTHER_RECIPIENT_NAMES          ( (OM_value_number) 32767 )
#define MH_VN_PREFERRED_DELIVERY_MODES       ( (OM_value_number) 10 )
#define MH_VN_RECIPIENT_DESCRIPTORS          ( (OM_value_number) 32767 )
#define MH_VN_REDIRECTION_HISTORY            ( (OM_value_number) 512 )
#define MH_VN_REPORT_SUBSTANCE               ( (OM_value_number) 32767 )
#define MH_VN_SECURITY_CATEGORY_DATA         ( (OM_value_number) 64 )
#define MH_VN_SECURITY_CATEGORY_IDS          ( (OM_value_number) 64 )
#define MH_VN_TRACE_INFO                     ( (OM_value_number) 512 )

/* END MH PORTION OF INTERFACE */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services, X/Open CAE Specification (November 1991), API to Electronic Mail (X.400).*

xmsga.h

Purpose

Definitions for the message store general attributes

Synopsis

#include <xom.h>
#include <xds.h>
#include <xdsmdup.h>
#include <xmhp.h>
#include <xmsga.h>

Description

The **xmsga.h** header declares the object identifiers for the message store general attributes. They are used with the directory message store object. This header must be included when use of the MHS directory user package (MDUP) has been negotiated.

All application programs that include this header must first include the **xom.h** object management header, the **xds.h** header, the **xdsmdup.h** and **xmhp.h** headers.

```
#ifndef XMSGA_HEADER
#define XMSGA_HEADER

/* MS General Attributes Package object identifier */

#define OMP_O_MS_GENERAL_ATTRIBUTES_PACKAGE  "\x56\x06\x01\x02\x06\x02"


/* MS General Attributes Types */
/*
 * Note: Every client program must explicitly import into
 * every compilation unit (C source program) the classes or
 * Object Identifiers that it uses.  Each of these classes or
```

```
 * Object Identifier names must then be explicitly exported from
 * just one compilation unit.
 * Importing and exporting can be done using the OM_IMPORT and
 * OM_EXPORT macros respectively (see [OM API]).
 * For instance, the client program uses
 *                OM_IMPORT( MS_A_CHILD_SEQUENCE_NUMBERS )
 * which in turn will make use of
 *                OMP_O_MS_A_CHILD_SEQUENCE_NUMBERS
 * defined below.
 */

#define OMP_O_MS_A_CHILD_SEQUENCE_NUMBERS           "\x56\x04\x03\x00"
#define OMP_O_MS_A_CONTENT                          "\x56\x04\x03\x01"
#define OMP_O_MS_A_CONTENT_CONFIDENTL_ALGM_ID       "\x56\x04\x03\x02"
#define OMP_O_MS_A_CONTENT_CORRELATOR               "\x56\x04\x03\x03"
#define OMP_O_MS_A_CONTENT_IDENTIFIER               "\x56\x04\x03\x04"
#define OMP_O_MS_A_CONTENT_INTEGRITY_CHECK          "\x56\x04\x03\x05"
#define OMP_O_MS_A_CONTENT_LENGTH                   "\x56\x04\x03\x06"
#define OMP_O_MS_A_CONTENT_RETURNED                 "\x56\x04\x03\x07"
#define OMP_O_MS_A_CONTENT_TYPE                     "\x56\x04\x03\x08"
#define OMP_O_MS_A_CONVERSION_LOSS_PROHIBITED       "\x56\x04\x03\x09"
#define OMP_O_MS_A_CONVERTED_EITS                   "\x56\x04\x03\x0A"
#define OMP_O_MS_A_CREATION_TIME                    "\x56\x04\x03\x0B"
#define OMP_O_MS_A_DELIVERED_EITS                   "\x56\x04\x03\x0C"
#define OMP_O_MS_A_DELIVERY_FLAGS                   "\x56\x04\x03\x0D"
#define OMP_O_MS_A_DL_EXPANSION_HISTORY             "\x56\x04\x03\x0E"
#define OMP_O_MS_A_ENTRY_STATUS                     "\x56\x04\x03\x0F"
#define OMP_O_MS_A_ENTRY_TYPE                       "\x56\x04\x03\x10"
#define OMP_O_MS_A_INTENDED_RECIPIENT_NAME          "\x56\x04\x03\x11"
#define OMP_O_MS_A_MESSAGE_DELIVERY_ENVELOPE        "\x56\x04\x03\x12"
#define OMP_O_MS_A_MESSAGE_DELIVERY_ID              "\x56\x04\x03\x13"
#define OMP_O_MS_A_MESSAGE_DELIVERY_TIME            "\x56\x04\x03\x14"
#define OMP_O_MS_A_MESSAGE_ORIGIN_AUTHEN_CHK        "\x56\x04\x03\x15"
#define OMP_O_MS_A_MESSAGE_SECURITY_LABEL           "\x56\x04\x03\x16"
#define OMP_O_MS_A_MESSAGE_SUBMISSION_TIME          "\x56\x04\x03\x17"
#define OMP_O_MS_A_MESSAGE_TOKEN                    "\x56\x04\x03\x18"
#define OMP_O_MS_A_ORIGINAL_EITS                    "\x56\x04\x03\x19"
#define OMP_O_MS_A_ORIGINATOR_CERTIFICATE           "\x56\x04\x03\x1A"
#define OMP_O_MS_A_ORIGINATOR_NAME                  "\x56\x04\x03\x1B"
#define OMP_O_MS_A_OTHER_RECIPIENT_NAMES            "\x56\x04\x03\x1C"
```

```
#define OMP_O_MS_A_PARENT_SEQUENCE_NUMBER           "\x56\x04\x03\x1D"
#define OMP_O_MS_A_PERRECIP_REPORT_DELIV_FLDS       "\x56\x04\x03\x1E"
#define OMP_O_MS_A_PRIORITY                         "\x56\x04\x03\x1F"
#define OMP_O_MS_A_PROOF_OF_DELIVERY_REQUEST        "\x56\x04\x03\x20"
#define OMP_O_MS_A_REDIRECTION_HISTORY              "\x56\x04\x03\x21"
#define OMP_O_MS_A_REPORT_DELIVERY_ENVELOPE         "\x56\x04\x03\x22"
#define OMP_O_MS_A_REPORT_ORIGIN_AUTHEN_CHK         "\x56\x04\x03\x23"
#define OMP_O_MS_A_REPORTING_DL_NAME                "\x56\x04\x03\x24"
#define OMP_O_MS_A_REPORTING_MTA_CERTIFICATE        "\x56\x04\x03\x25"
#define OMP_O_MS_A_SECURITY_CLASSIFICATION          "\x56\x04\x03\x26"
#define OMP_O_MS_A_SEQUENCE_NUMBER                  "\x56\x04\x03\x27"
#define OMP_O_MS_A_SUBJECT_SUBMISSION_ID            "\x56\x04\x03\x28"
#define OMP_O_MS_A_THIS_RECIPIENT_NAME              "\x56\x04\x03\x29"


/* Enumeration Constants */

/* for MS_A_ENTRY_STATUS */

#define MS_ES_NEW                       ((OM_enumeration) 0)
#define MS_ES_LISTED                    ((OM_enumeration) 1)
#define MS_ES_PROCESSED                 ((OM_enumeration) 2)


/* for MS_A_ENTRY_TYPE */

#define MS_ET_DELIVERED_MESSAGE         ((OM_enumeration) 0)
#define MS_ET_DELIVERED_REPORT          ((OM_enumeration) 1)
#define MS_ET_RETURNED_CONTENT          ((OM_enumeration) 2)


/* for MS_A_PRIORITY */

#define MS_PTY_NORMAL                   ((OM_enumeration) 0)
#define MS_PTY_LOW                      ((OM_enumeration) 1)
#define MS_PTY_URGENT                   ((OM_enumeration) 2)


/* for MS_A_SECURITY_CLASSIFICATION */
```

```
#define MS_SC_UNMARKED                  ((OM_enumeration) 0)
#define MS_SC_UNCLASSIFIED              ((OM_enumeration) 1)
#define MS_SC_RESTRICTED                ((OM_enumeration) 2)
#define MS_SC_CONFIDENTIAL              ((OM_enumeration) 3)
#define MS_SC_SECRET                    ((OM_enumeration) 4)
#define MS_SC_TOP_SECRET                ((OM_enumeration) 5)


#endif /* XMSGA_HEADER */
```

Related Information

X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services, X/Open CAE Specification (November 1991), API to Electronic Mail (X.400).

xom_intro

Purpose

Introduction to X/OPEN OSI-Abstract-Data Manipulation (XOM) functions

Synopsis

```
#include <xom.h>
#include <xomext.h>
```

Description

This **xom_intro** reference page defines the functions of the C interface. The following table lists the relevant functions.

Service Interface Functions—xom_intro(3xom)	
Function	**Description**
omX_extract()	Gets attribute values from specified object
omX_fill()	Initializes an OM_descriptor structure
omX_fill_oid()	Initializes an OM_descriptor with an OID value
omX_object_to_string()	Converts an OM_object to string format
omX_string_to_object()	Converts a string to OM_object
om_copy()	Copies a private object.
om_copy_value()	Copies a string between private objects.
om_create()	Creates a private object.
om_decode()	This function is not supported by the DCE XOM interface, and returns with an **OM_FUNCTION_DECLINED** error.

om_delete()	Deletes a private or service-generated object.
om_encode()	This function is not supported by the DCE XOM interface, and returns with an **OM_FUNCTION_DECLINED** error.
om_get()	Gets copies of attribute values from a private object.
om_instance()	Tests an object's class.
om_put()	Puts attribute values into a private object.
om_read()	Reads a segment of a string in a private object.
om_remove()	Removes attribute values from a private object.
om_write()	Writes a segment of a string into a private object.

As indicated in the table, the service interface comprises a number of functions whose purpose and range of capabilities are summarized as follows:

omX_extract()
: Creates a new public object that is an exact but independent copy of an existing subobject in a private object. This function is similiar to the **om_get()** function but includes an additional parameter *navigation_path* that contains directions to the required object to be extracted.

omX_fill() Initializes an OM descriptor structure with user supplied values for its type, syntax and value.

omX_fill_oid()
: Initializes an OM descriptor structure with user supplied values for its type and value. The syntax of the descriptor is always set to **OM_S_OBJECT_IDENTIFIER_STRING**.

omX_object_to_string()
: Converts an OM object into a string format.

omX_string_to_object()
Creates a new private object, which is build from the *string* and *class* input parameters.

om_copy() Creates an independent copy of an existing private object and all its subobjects. The copy is placed in the original's workspace, or in another specified by the XOM application.

om_copy_value()
Replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.

om_create() Creates a new private object that is an instance of a particular class. The object can be initialized with the attribute values specified as initial in the class definition.

The service does not permit the API user to explicitly create instances of all classes, but rather only those indicated by a package's definition as having this property.

om_delete() Deletes a service-generated public object, or makes a private object inaccessible.

om_get() Creates a new public object that is an exact but independent copy of an existing private object. The client can request certain exclusions, each of which reduces the copy to a part of the original. The client can also request that values be converted from one syntax to another before they are returned.

The copy can exclude: attributes of types other than those specified, values at positions other than those specified within an attribute, the values of multivalued attributes, copies of (not handles for) subobjects, or all attribute values (revealing only an attribute's presence).

om_instance()
Determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. This function is useful since it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.

om_put() Places or replaces in one private object copies of the attribute values of another public or private object.

The source values can be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values can be substituted for any existing destination values or for the values at specified positions in the destination attribute.

om_read() Reads a segment of a value of an attribute of a private object. The value must be a string. The value can first be converted from one syntax to another. The function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

om_remove()
Removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.

om_write() Writes a segment of a value of an attribute to a private object. The value must be a string. The segment can first be converted from one syntax to another. The written segment becomes the value's last segment since any elements beyond it are discarded. The function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

In the C interface, the functions are realized by macros. The function prototype in the synopsis of a function's specification simply shows the client's view of the function.

The intent of the interface definition is that each function be atomic; that is, either it carries out its assigned task in full and reports success, or it fails to carry out even a part of the task and reports an exception. However, the service does not guarantee that a task is always carried out in full.

Errors

Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages. The possible error return values are described in the function reference pages.

XOM functions check for NULL pointers and return an error, except for workspace pointers. Pointers are only checked at the function interface. The check is only for NULL and not for validity. If NULL or invalid pointers are passed this may result in an undetermined behaviour.

omX_extract

Purpose

Extracts the first occurrence of the requested OM type from an object

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code omX_extract(
        OM_private_object object,
        OM_type_list navigation_path,
        OM_exclusions exclusions,
        OM_type_list included_types,
        OM_boolean local_strings,
        OM_value_position initial_value,
        OM_value_position limiting_value,
        OM_public_object *values,
        OM_value_position *total_number);
```

Parameters

Input

object The object from which data is to be extracted.

navigation_path
Contains a NULL-terminated list of OM types that lead to the target object to be extracted. It does not include the OM type of the target object.

exclusions Explicit requests for zero or more exclusions, each of which reduces the copy to a prescribed portion of the original. The exclusions apply to the attributes of the target object, but not to those of its subobjects.

Apart from **OM_NO_EXCLUSIONS**, each value is chosen from the following list. When multiple exclusions are specified, each is applied in the order in which it is displayed in the list with lower-numbered

exclusions having precedence over higher-numbered exclusions. If, after the application of an exclusion, that portion of the object is not returned, no further exclusions need be applied to that portion.

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**

  The copy includes descriptors comprising only attributes of specified types. Note that this exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

- **OM_EXCLUDE_MULTIPLES**

  The copy includes a single descriptor for each attribute that has two or more values, rather than one descriptor for each value. None of these descriptors contains an attribute value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  If the attribute has values of two or more syntaxes, the descriptor identifies one of those syntaxes; however, the syntax identified is not specified.

  Note that this exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**

  The copy includes descriptors comprising only values at specified positions within an attribute. Note that, when this exclusion is used in conjunction with the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion, it provides a means for determining the values of a specified attribute, as well as the syntaxes of those values, one or more but not all attributes at a time.

- **OM_EXCLUDE_VALUES**

  The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  Note that this exclusion provides a means for determining an object's composition; that is, the type and syntax of each of its attribute values.

- **OM_EXCLUDE_SUBOBJECTS**

The copy includes, for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls.

Note that this exclusion provides a means for examining an object one level at a time.

- **OM_EXCLUDE_DESCRIPTORS**

  When this exclusion is specified, no descriptors are returned and the copy result is not present. The *total_number* parameter reflects the number of descriptors that would be returned by applying the other inclusion and exclusion specifications.

  Note that this exclusion provides an attribute analysis capability. For instance, the total number of values in a multivalued attribute can be determined by specifying an inclusion of the specific attribute type, and exclusions of **OM_EXCLUDE_DESCRIPTORS**, **OM_EXCLUDE_SUBOBJECTS**, and **OM_EXCLUDE_ALL_BUT_THESE_TYPES**.

The **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion affects the choice of descriptors, while the **OM_EXCLUDE_VALUES** exclusion affects the composition of descriptors.

included_types
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion is requested; it identifies the types of the attributes to be included in the copy (provided that they are displayed in the original).

local_strings
: This Boolean parameter indicates whether conversion to local string format should be carried out or not.

initial_value
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute of the first value to be included in the copy.

  If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

limiting_value

This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute one beyond that of the last value to be included in the copy. If this parameter is not greater than the *initial_value* parameter, no values are included (and no descriptors are returned).

If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Output

values The *values* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS** and the **OM_EXCLUDE_DESCRIPTORS** exclusion is not specified. It contains the array of OM descriptors extracted.

The memory space for *values* is provided by **omX_extract()**. It is the responsibility of the calling function to subsequently release this space through a call to **om_delete()**.

total_number

The number of attribute descriptors returned in the public object, but not in any of its subobjects, based on the inclusion and exclusion parameters specified. If the **OM_EXCLUDE_DESCRIPTORS** exclusion is specified, no *values* result is returned and the *total_number* result reflects the actual number of attribute descriptors that would be returned based on the remaining inclusion and exclusion values.

Note that the total includes only the attribute descriptors in the *values* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **omX_extract()** function creates a new public object that is an exact, but independent, copy of an existing subobject in a private object. It is similiar to the **om_get()** function but includes an additional parameter, *navigation_path* which contains directions to the required object to be extracted. The client can request certain exclusions, each of which reduces the copy to a part of the original.

One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the *values* parameter includes a descriptor that omits the elements (but not the length) of the string. The *elements* component of the *string* component in the descriptor's *value* component is **OM_ELEMENTS_UNSPECIFIED**, and the **OM_S_LONG_STRING** bit of the *syntax* component is set to **OM_TRUE**.

The parameters *exclusions*, *included_types*, *local_strings*, *initial_value*, and *limiting_value* only apply to the target object being extracted.

Note that the client can access long values by means of **om_read()**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

omX_fill

Purpose

Initializes an **OM_descriptor** structure

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code omX_fill(
        OM_type type,
        OM_syntax syntax,
        OM_uint32 length,
        void *elements,
        OM_descriptor *destination);
```

Parameters

Input

type The type of OM descriptor structure.

syntax The syntax value for this OM descriptor.

length The data length for values of string syntax. Zero is entered for values of type **OM_object**. When initializing an **OM_descriptor** with an **OM_type** that has an **OM_syntax** of either **OM_S_INTEGER**, **OM_S_BOOLEAN** or **OM_S_ENUMERATION**, then the associated value must be entered in the *length* parameter.

elements The string contents.

Output

destination Contains the filled descriptor.

Description

The **omX_fill()** function is used to initialize an OM descriptor structure with user supplied values for its type, syntax, and value.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages. Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*.

omX_fill_oid

Purpose

Initializes an **OM_descriptor** structure with an object identifier value

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code omX_fill_oid(
        OM_type type,
        OM_object_identifier object_id,
        OM_descriptor *destination);
```

Parameters

Input

type The type of **OM_descriptor** structure.

object_id The object identifier value.

Output

destination Contains the filled descriptor.

Description

The **omX_fill_oid()** function is used to initialize an OM descriptor structure with user-supplied values for its type and value. The syntax of the descriptor is always set to **OM_S_OBJECT_IDENTIFIER_STRING**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

omX_object_to_string

Purpose

Converts an OM object from descriptor to string format

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code omX_object_to_string(
        OM_object object,
        OM_boolean local_strings,
        OM_string *string);
```

Parameters

Input

object Contains the OM object to be converted.

local_strings This Boolean value indicates if the *string* return value should be converted to a local string format. For further information on local strings please refer to the *DCE 1.2.2 Application Development Guide—Directory Services*.

Output

string Contains the converted object in string format.

The calling function should provide the memory for *string*. The string's contents are initially unspecified. The string's length becomes the number of octets required to contain the segment that the function is to read. The service modifies this parameter. The string's elements become the elements actually read. The string's length becomes the number of octets required to hold the segment actually read.

Description

The **omX_object_to_string()** function converts an OM object into a string format. The object can either be a client-generated or a service-generated public or private object.

The objects that can be handled by this function are restricted to those defined in the schema file, **xoischema**. Additionally, the OM objects **DS_C_ATTRIBUTE_ERROR** and **DS_C_ERROR** are also handled. For these, a message string containing the error message is returned.

For the syntax of the output strings, please refer to the *DCE 1.2.2 Application Development Guide—Directory Services*.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** and **xomext.h** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

omX_string_to_object

Purpose

Converts an OM object specified in string format to descriptor format

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code omX_string_to_object(
        OM_workspace workspace,
        OM_string *string,
        OM_object_identifier class,
        OM_boolean local_strings,
        OM_private_object *object,
        OM_integer *error_position,
        OM_integer *error_type);
```

Parameters

Input

workspace The workspace pointer obtained from a **ds_initialize()** call.

string The string to be converted. Refer to the *DCE 1.2.2 Application Development Guide—Directory Services* for details of the string syntaxes allowed.

class The OM class of the object to be created.

local_strings Indicates if the attribute values are to be converted from their local string format.

Output

object The converted object.

error_position
: If there is a syntax error in the input string, then *error_position* indicates the position in the string where the error was detected.

error_type
: Indicates the type of error. Refer to the **xomext.h** header file for explanations of the error types.

Description

The **omX_string_to_object()** function creates a new private object, which is built from the *string* and *class* input parameters.

The objects that can be created by this function are restricted to those defined in the schema file, **xoischema**.

Notes

The memory space for the *object* return parameter is allocated by **omX_string_to_object()**. The calling application is responsible for releasing this memory with the **om_delete()** function call.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

  If there is a syntax error in the input string, *OM_return_code* is set to **OM_WRONG_VALUE_MAKEUP** and the type of error is returned in *error_type*.

Errors

Refer to **xom.h(4xom)** and **xomext.h** for a list of the possible error values that can be returned in *OM_return_code* and *error_type*. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

om_copy

Purpose

Creates a new private object that is an exact, but independent, copy of an existing private object

Synopsis

```
#include <xom.h>

OM_return_code om_copy(
        OM_private_object original,
        OM_workspace workspace,
        OM_private_object *copy);
```

Parameters

Input

original The original that remains accessible.

workspace The workspace in which the copy is to be created. The original's class must be in a package associated with this workspace.

Output

copy The new copy of the private object. This result is present if and only if the return value for *OM_return_code* is **OM_SUCCESS**.

Description

The **om_copy()** function creates a new private object (the copy) that is an exact but independent copy of an existing private object (the original). The function is recursive in that copying the original also copies its subobjects.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_WORKSPACE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_TOO_MANY_VALUES**

om_copy_value

Purpose

Places or replaces a string in one private object with a copy of a string in another private object

Synopsis

```
#include <xom.h>

OM_return_code om_copy_value(
        OM_private_object source,
        OM_type source_type,
        OM_value_position source_value_position,
        OM_private_object destination,
        OM_type destination_type,
        OM_value_position destination_value_position);
```

Parameters

Input

source The source that remains accessible.

source_type Identifies the type of an attribute. One of the attribute values is copied.

source_value_position
The position within the *source* attribute of the value copied.

destination The destination that remains accessible.

destination_type
Identifies the type of the attribute. One of the attribute values is placed or replaced.

destination_value_position
The position within the *destination* attribute of the value placed or replaced. If the value position exceeds the number of values present in the *destination* attribute, the parameter is taken to be equal to that number.

Description

The **om_copy_value()** function places or replaces an attribute value in one private object (the destination) with a copy of an attribute value in another private object (the source). The source value is a string. The copy's syntax is that of the original.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**

- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_create

Purpose

Creates a new private object that is an instance of a particular class

Synopsis

```
#include <xom.h>

OM_return_code om_create(
        OM_object_identifier class,
        OM_boolean initialize,
        OM_workspace workspace,
        OM_private_object *object);
```

Parameters

Input

class Identifies the class of the object to be created. The specified class shall be concrete.

initialize Determines whether the object created is initialized as specified in the definition of its class. If this parameter is **OM_TRUE**, the object is made to comprise the attribute values specified as initial values in the tabular definitions of the object's class and its superclasses. If this parameter is **OM_FALSE**, the object is made to comprise the **OM_CLASS** attribute alone.

workspace The workspace in which the object is created. The specified class is in a package associated with this workspace.

Output

object The created object. This result is present if and only if the return value for *OM_return_code* is **OM_SUCCESS**.

Description

The **om_create()** function creates a new private object that is an instance of a particular class.

Notes

By subsequently adding new values to the object and replacing and removing existing values, the client can create all conceivable instances of the object's class.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_WORKSPACE**

- **OM_NOT_CONCRETE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_delete

Purpose

Deletes a private or service-generated object

Synopsis

```
#include <xom.h>

OM_return_code om_delete(
        OM_object subject);
```

Parameters

Input

subject The object to be deleted.

Description

The **om_delete()** function deletes a service-generated public object or makes a private object inaccessible. It is not intended for use on client-generated public objects.

If applied to a service-generated public object, the function deletes the object and releases any resources associated with the object, including the space occupied by descriptors and attribute values. The function is applied recursively to any public subobjects. This does not affect any private subobjects.

If applied to a private object, the function makes the object inaccessible. Any existing object handles for the object are invalidated. The function is applied recursively to any private subobjects.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_THE_SERVICES**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_get

Purpose

Creates a public copy of all or particular parts of a private object

Synopsis

```
#include <xom.h>

OM_return_code om_get(
        OM_private_object original,
        OM_exclusions exclusions,
        OM_type_list included_types,
        OM_boolean local_strings,
        OM_value_position initial_value,
        OM_value_position limiting_value,
        OM_public_object *copy,
        OM_value_position *total_number);
```

Parameters

Input

original The original that remains accessible.

exclusions Explicit requests for zero or more exclusions, each of which reduces the copy to a prescribed portion of the original. The exclusions apply to the attributes of the object, but not to those of its subobjects.

Apart from **OM_NO_EXCLUSIONS**, each value is chosen from the following list. When multiple exclusions are specified, each is applied in the order in which it is displayed in the list with lower-numbered exclusions having precedence over higher-numbered exclusions. If, after the application of an exclusion, that portion of the object is not returned, no further exclusions need be applied to that portion.

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**

The copy includes descriptors comprising only attributes of specified types. Note that this exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

- **OM_EXCLUDE_MULTIPLES**

  The copy includes a single descriptor for each attribute that has two or more values, rather than one descriptor for each value. None of these descriptors contains an attribute value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  If the attribute has values of two or more syntaxes, the descriptor identifies one of those syntaxes; however, the syntax identified is not specified.

  Note that this exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**

  The copy includes descriptors comprising only values at specified positions within an attribute. Note that, when this exclusion is used in conjunction with the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion, it provides a means for determining the values of a specified attribute, as well as the syntaxes of those values, one or more but not all attributes at a time.

- **OM_EXCLUDE_VALUES**

  The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  Note that this exclusion provides a means for determining an object's composition; that is, the type and syntax of each of its attribute values.

- **OM_EXCLUDE_SUBOBJECTS**

  The copy includes, for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject, rather than a public copy of it.

This handle makes that subobject accessible for use in subsequent function calls.

Note that this exclusion provides a means for examining an object one level at a time.

- **OM_EXCLUDE_DESCRIPTORS**

  When this exclusion is specified, no descriptors are returned and the copy result is not present. The *total_number* parameter reflects the number of descriptors that would be returned by applying the other inclusion and exclusion specifications.

  Note that this exclusion provides an attribute analysis capability. For instance, the total number of values in a multivalued attribute can be determined by specifying an inclusion of the specific attribute type, and exclusions of **OM_EXCLUDE_DESCRIPTORS**, **OM_EXCLUDE_SUBOBJECTS**, and **OM_EXCLUDE_ALL_BUT_THESE_TYPES**.

The **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion affects the choice of descriptors, while the **OM_EXCLUDE_VALUES** exclusion affects the composition of descriptors.

included_types
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion is requested; it identifies the types of the attributes to be included in the copy (provided that they are displayed in the original).

local_strings
: This Boolean parameter indicates whether conversion to local string format should be carried out or not. For further information on local strings please refer the *DCE 1.2.2 Application Development Guide—Directory Services*.

initial_value
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute of the first value to be included in the copy.

  If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

limiting_value

This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute one beyond that of the last value to be included in the copy. If this parameter is not greater than the *initial_value* parameter, no values are included (and no descriptors are returned).

If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Output

copy

The *copy* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS** and the **OM_EXCLUDE_DESCRIPTORS** exclusion is not specified.

The space occupied by the public object and every attribute value that is a string is service provided. If the client alters any part of that space, the effect upon the service's subsequent behavior is unspecified.

total_number

The number of attribute descriptors returned in the public object, but not in any of its subobjects, based on the inclusion and exclusion parameters specified. If the **OM_EXCLUDE_DESCRIPTORS** exclusion is specified, no *copy* result is returned and the *total_number* result reflects the actual number of attribute descriptors that would be returned based on the remaining inclusion and exclusion values.

Note that the total includes only the attribute descriptors in the *copy* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **om_get()** function creates a new public object (the *copy*) that is an exact, but independent, copy of an existing private object, the *original* parameter. The client can request certain exclusions, each of which reduces the copy to a part of the original.

One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the *copy* parameter includes a descriptor that omits the elements (but not the length) of the string. The *elements* component of the *string* component in the descriptor's *value*

component is **OM_ELEMENTS_UNSPECIFIED**, and the **OM_S_LONG_STRING** bit of the *syntax* component is set to **OM_TRUE**.

Note that the client can access long values by means of **om_read()**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_EXCLUSION**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**

- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_instance

Purpose

Determines whether an object is an instance of a particular class or any of its subclasses

Synopsis

```
#include <xom.h>

OM_return_code om_instance(
        OM_object subject,
        OM_object_identifier class,
        OM_boolean *instance);
```

Parameters

Input

subject The subject that remains accessible.

class Identifies the class in question.

Output

instance Indicates whether the subject is an instance of the specified class or any of its subclasses. This result is present if and only if the value of the *OM_return_code* is set to **OM_SUCCESS**.

Description

The **om_instance()** function determines whether a service-generated public or private object (the subject) is an instance of a particular class or any of its subclasses.

Notes

The client can determine an object's class (**C**) by simply inspecting the object, using programming language constructs if the object is public or **om_get()** if it is private.

This function is useful in that it reveals that an object is an instance of the specified class, even if **C** is a subclass of that class.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NOT_THE_SERVICES**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_put

Purpose

Places or replaces in one private object copies of the attribute values of another public or private object

Synopsis

```
#include <xom.h>

OM_return_code om_put(
        OM_private_object destination,
        OM_modification modification,
        OM_object source,
        OM_type_list included_types,
        OM_value_position initial_value,
        OM_value_position limiting_value);
```

Parameters

Input

destination The destination that remains accessible. The destination's class is unaffected.

modification The nature of the requested modification. The modification determines how **om_put()** uses the attribute values in the source to modify the object. In all cases, for each attribute present in the source, copies of its values are placed in the object's destination attribute of the same type. The data value is chosen from among the following:

- **OM_INSERT_AT_BEGINNING**

  The source values are inserted before any existing destination values. (The latter are retained.)

- **OM_INSERT_AT_CERTAIN_POINT**

  The source values are inserted before the value at a specified position in the destination attribute. (The latter are retained.)

- **OM_INSERT_AT_END**

  The source values are inserted after any existing destination values. (The latter are retained.)

- **OM_REPLACE_ALL**

  The source values are placed in the *destination* attribute. The existing destination values, if any, are discarded.

- **OM_REPLACE_CERTAIN_VALUES**

  The source values are substituted for the values at specified positions in the destination attribute. (The latter are discarded.)

source The source that remains accessible. The source's class is ignored. However, the attributes being copied from the source must be compatible with the destination's class definition.

included_types
If present, this parameter identifies the types of the attributes to be included in the destination (provided that they are displayed in the source); otherwise, all attributes are to be included.

initial_value This parameter is present if and only if the *modification* parameter is **OM_INSERT_AT_CERTAIN_POINT** or **OM_REPLACE_CERTAIN_VALUES**. It specifies the position within each destination attribute at which source values are inserted, or of the first value replaced, respectively.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in a destination attribute, the parameter is taken to be equal to that number.

limiting_value
Present if and only if the *modification* parameter is **OM_REPLACE_CERTAIN_VALUES**. It specifies the position within each destination attribute one beyond that of the last value replaced. If this parameter is present, it must be greater than the *initial_value* parameter.

If the *limiting_value* parameter is **OM_ALL_VALUES** or exceeds the number of values present in a destination attribute, the parameter is taken to be equal to that number.

Description

The **om_put()** function places or replaces in one private object (that is, the destination) copies of the attribute values of another public or private object (that is, the source). The client can specify that the source's values replace all or particular values in the destination, or are inserted at a particular position within each attribute. All string values being copied that are in the local representation are first converted into the nonlocal representation for that syntax (which may entail the loss of some information).

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_MODIFICATION**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**

- **OM_NO_SUCH_TYPE**
- **OM_NOT_CONCRETE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_TOO_MANY_VALUES**
- **OM_VALUES_NOT_ADJACENT**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_MAKEUP**
- **OM_WRONG_VALUE_NUMBER**
- **OM_WRONG_VALUE_POSITION**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_read

Purpose

Reads a segment of a string in a private object

Synopsis

```
#include <xom.h>

OM_return_code om_read(
        OM_private_object subject,
        OM_type type,
        OM_value_position value_position,
        OM_boolean local_string,
        OM_string_length *string_offset,
        OM_string *elements);
```

Parameters

Input

subject The subject that remains accessible.

type Identifies the type of the attribute, one of whose values is read.

value_position
The position within the attribute of the value read.

local_string This Boolean parameter indicates whether conversion to local string format should be carried out or not. For further information on local strings please refer to the *DCE 1.2.2 Application Development Guide—Directory Services*.

Input/Output

string_offset On input this parameter contains the offset, in octets, of the start of the string segment to be read. If it exceeds the total length of the string, the parameter is equal to the string length.

On output it contains the offset, in octets, of the start of the next string segment to be read, or 0 (zero) if the value's final segment is read. The result is present if, and only if, the *OM_return_code* is **OM_SUCCESS**. The value returned can be used as the input *string_offset* parameter in the next call of this function. This enables sequential reading of a value of a long string.

elements On input, the space the client provides for the segment to be read. The string's contents are initially unspecified. The string's length is initially the number of octets required to contain the segment that the function is to read.

On output, the string's elements become the elements actually read. The string's length becomes the number of octets required to hold the segment actually read. This can be less than the initial length if the segment is the last in a long string.

Description

The **om_read()** function reads a segment of an attribute value in a private object, namely the subject.

The segment returned is a segment of the string value that is returned if the complete value is read in a single call.

Note that this function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_SYNTAX**

om_remove

Purpose

Removes and discards values of an attribute of a private object

Synopsis

```
#include <xom.h>

OM_return_code om_remove(
        OM_private_object subject,
        OM_type type,
        OM_value_position initial_value,
        OM_value_position limiting_value);
```

Parameters

Input

subject The subject that remains accessible. The subject's class is unaffected.

type Identifies the type of the attribute, some of whose values are removed. The type is not **OM_CLASS**.

initial_value The position within the attribute of the first value removed.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in the attribute, the parameter is taken to be equal to that number.

limiting_value

The position within the attribute one beyond that of the last value removed. If this parameter is not greater than the *initial_value* parameter, no values are removed.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Description

The **om_remove()** function removes and discards particular values of an attribute of a private object, the subject. If no values remain, the attribute itself is also removed. If the value is a subobject, the value is first removed and then **om_delete()** is applied to it, thus destroying the object.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_write

Purpose

Writes a segment of a string into a private object

Synopsis

```
#include <xom.h>

OM_return_code om_write(
        OM_private_object subject,
        OM_type type,
        OM_value_position value_position,
        OM_syntax syntax,
        OM_string_length *string_offset,
        OM_string elements);
```

Parameters

Input

subject The subject that remains accessible.

type Identifies the type of the attribute, one of whose values is written.

value_position
The position within the above attribute of the value to be written. The value position can neither be negative nor exceed the number of values present. If it equals the number of values present, the segment is inserted into the attribute as a new value.

syntax If the value being written is not already present in the subject, this identifies the syntax that the value has. It must be a permissible syntax for the attribute of which this is a value. If the value being written is already present in the subject, then that value's syntax is preserved and this parameter is ignored.

elements The string segment to be written. A copy of this segment occupies a position within the string value being written, starting at the offset given

by the *string_offset* input parameter. Any values already at or beyond this offset are discarded.

Input/Output

string_offset On input this parameter contains the offset, in octets, of the start of the string segment to be written. If it exceeds the current length of the string value being written, the parameter is taken to be equal to that current length.

On output it contains the offset, in octets, after the last string segment written. This result is present if, and only if, the *OM_return_code* result is **OM_SUCCESS**. The value returned in *string_offset* can be used as the input *string_offset* parameter the next time this function is called. This enables sequential writing of the value of a long string.

Description

The **om_write()** function writes a segment of an attribute value in a private object, the *subject* parameter.

The segment supplied is a segment of the string value that is supplied if the complete value is written in a single call.

The written segment is made the value's last. The function discards any values whose offset equals or exceeds the *string_offset* result. If the value being written is in the local representation, it is converted to the nonlocal representation (which may entail the loss of information and which may yield a different number of elements than that provided).

Note that this function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code

Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to

OM_SUCCESS; whereas, if the function fails, it has one of the values listed under **ERRORS**.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_MAKEUP**
- **OM_WRONG_VALUE_POSITION**
- **OM_WRONG_VALUE_SYNTAX**

xom.h

Purpose

Header file for XOM

Synopsis

#include <xom.h>

Description

The declarations, as assembled here, constitute the contents of a header file made accessible to client programmers. The header file includes by reference a second header file (**xomi.h**) comprising the declarations defining the C workspace interface. The **xomi.h** header file and the workspace interface are only used internally by the service interface, and are not visible to the client programmer.

```
#ifndef XOM_HEADER
#define XOM_HEADER

/* BEGIN SERVICE INTERFACE */

/* INTERMEDIATE DATA TYPES */

typedef  int            OM_sint;
typedef  short          OM_sint16;
typedef  long int       OM_sint32;
typedef  unsigned       OM_uint;
typedef  unsigned short OM_uint16;
typedef  long unsigned  OM_uint32;

/* PRIMARY DATA TYPES */

/* Boolean */
```

```
typedef OM_uint32 OM_boolean;

/* String Length */

typedef OM_uint32 OM_string_length;

/* Enumeration */

typedef OM_sint32 OM_enumeration;

/* Exclusions */

typedef OM_uint OM_exclusions;

/* Integer */

typedef OM_sint32 OM_integer;

/* Modification */

typedef OM_uint OM_modification;

/* Object */

typedef struct OM_descriptor_struct *OM_object;

/* String */

typedef struct {
        OM_string_length     length;
        void   *elements;
} OM_string;

#define OM_STRING(string)       \
         { (OM_string_length)(sizeof(string)-1), string }

/* Workspace */

typedef void *OM_workspace;
```

```
/* SECONDARY DATA TYPES */

/* Object Identifier */

typedef OM_string OM_object_identifier;

/* Private Object */

typedef OM_object OM_private_object;

/* Public Object */

typedef OM_object OM_public_object;

/* Return Code */

typedef OM_uint OM_return_code;

/* Syntax */

typedef OM_uint16 OM_syntax;

/* Type */

typedef OM_uint16 OM_type;

/* Type List */

typedef OM_type *OM_type_list;

/* Value */

typedef struct {
        OM_uint32       padding;
        OM_object       object;
} OM_padded_object;

typedef union OM_value_union {
        OM_string       string;
```

```
        OM_boolean      boolean;
        OM_enumeration  enumeration;
        OM_integer      integer;
        OM_padded_object        object;
} OM_value;

/* Value Length */

typedef OM_uint32 OM_value_length;

/* Value Position */

typedef OM_uint32 OM_value_position;

/* TERTIARY DATA TYPES */

/* Descriptor */

typedef struct OM_descriptor_struct {
        OM_type                 type;
        OM_syntax               syntax;
        union OM_value_union    value;
} OM_descriptor;


/* SYMBOLIC CONSTANTS */

/* Boolean */

#define OM_FALSE        ((OM_boolean) 0)
#define OM_TRUE         ((OM_boolean) 1)

/* Element Position */

#define OM_LENGTH_UNSPECIFIED   ((OM_string_length) 0xFFFFFFFF)

/* Exclusions */

#define OM_NO_EXCLUSIONS                    ((OM_exclusions) 0)
#define OM_EXCLUDE_ALL_BUT_THESE_TYPES      ((OM_exclusions) 1)
```

```
#define OM_EXCLUDE_ALL_BUT_THESE_VALUES    ((OM_exclusions) 2)
#define OM_EXCLUDE_MULTIPLES               ((OM_exclusions) 4)
#define OM_EXCLUDE_SUBOBJECTS              ((OM_exclusions) 8)
#define OM_EXCLUDE_VALUES                  ((OM_exclusions) 16)
#define OM_EXCLUDE_DESCRIPTORS             ((OM_exclusions) 32)

/* Modification */

#define OM_INSERT_AT_BEGINNING             ((OM_modification) 1)
#define OM_INSERT_AT_CERTAIN_POINT         ((OM_modification) 2)
#define OM_INSERT_AT_END                   ((OM_modification) 3)
#define OM_REPLACE_ALL                     ((OM_modification) 4)
#define OM_REPLACE_CERTAIN_VALUES          ((OM_modification) 5)

/* Object Identifiers */

/*  NOTE: These macros rely on the ## token-pasting operator of
 *  ANSI C. On many pre-ANSI compilers the same effect can be
 *  obtained by replacing ## with /**/

/* Private macro to calculate length of an object identifier
 */
#define OMP_LENGTH(oid_string)  (sizeof(OMP_O_##oid_string)-1)


/* Macro to initialize the syntax and value of an object identifier
 */
#define OM_OID_DESC(type, oid_name)                        \
        { (type), OM_S_OBJECT_IDENTIFIER_STRING,           \
        { { OMP_LENGTH(oid_name) , OMP_D_##oid_name } } }


/* Macro to mark the end of a client-allocated public object
 */
#define OM_NULL_DESCRIPTOR      \
  { OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, \
  { { 0, OM_ELEMENTS_UNSPECIFIED } } }


/* Macro to make class constants available
```

```
/* within a compilation unit
 */
#define OM_IMPORT(class_name)                                   \
                extern char OMP_D_##class_name [];              \
                extern OM_string class_name;

/* Macro to allocate memory for class constants
/* within a compilation unit
 */
#define OM_EXPORT(class_name)                                   \
        char OMP_D_##class_name[] = OMP_O_##class_name ;        \
        OM_string class_name =                                  \
            { OMP_LENGTH(class_name), OMP_D_##class_name } ;


/* Constant for the OM package
 */
/* { joint-iso-ccitt(2) mhs-motis(6) group(6) white(1)
      api(2) om(4) } */

#define OMP_O_OM_OM             "\x56\x06\x01\x02\x04"

/* Constant for the Encoding class
 */
#define OMP_O_OM_C_ENCODING     "\x56\x06\x01\x02\x04\x01"

/* Constant for the External class
 */
#define OMP_O_OM_C_EXTERNAL     "\x56\x06\x01\x02\x04\x02"

/* Constant for the Object class
 */
#define OMP_O_OM_C_OBJECT       "\x56\x06\x01\x02\x04\x03"

/* Constant for the BER Object Identifier
 */
#define OMP_O_OM_BER            "\x51\x01"

/* Constant for the Canonical-BER Object Identifier
 */
```

```
#define OMP_O_OM_CANONICAL_BER  "\x56\x06\x01\x02\x04\x04"

/* Return Code */

#define OM_SUCCESS                  ((OM_return_code) 0)
#define OM_ENCODING_INVALID         ((OM_return_code) 1)
#define OM_FUNCTION_DECLINED        ((OM_return_code) 2)
#define OM_FUNCTION_INTERRUPTED     ((OM_return_code) 3)
#define OM_MEMORY_INSUFFICIENT      ((OM_return_code) 4)
#define OM_NETWORK_ERROR            ((OM_return_code) 5)
#define OM_NO_SUCH_CLASS            ((OM_return_code) 6)
#define OM_NO_SUCH_EXCLUSION        ((OM_return_code) 7)
#define OM_NO_SUCH_MODIFICATION     ((OM_return_code) 8)
#define OM_NO_SUCH_OBJECT           ((OM_return_code) 9)
#define OM_NO_SUCH_RULES            ((OM_return_code) 10)
#define OM_NO_SUCH_SYNTAX           ((OM_return_code) 11)
#define OM_NO_SUCH_TYPE             ((OM_return_code) 12)
#define OM_NO_SUCH_WORKSPACE        ((OM_return_code) 13)
#define OM_NOT_AN_ENCODING          ((OM_return_code) 14)
#define OM_NOT_CONCRETE             ((OM_return_code) 15)
#define OM_NOT_PRESENT              ((OM_return_code) 16)
#define OM_NOT_PRIVATE              ((OM_return_code) 17)
#define OM_NOT_THE_SERVICES         ((OM_return_code) 18)
#define OM_PERMANENT_ERROR          ((OM_return_code) 19)
#define OM_POINTER_INVALID          ((OM_return_code) 20)
#define OM_SYSTEM_ERROR             ((OM_return_code) 21)
#define OM_TEMPORARY_ERROR          ((OM_return_code) 22)
#define OM_TOO_MANY_VALUES          ((OM_return_code) 23)
#define OM_VALUES_NOT_ADJACENT      ((OM_return_code) 24)
#define OM_WRONG_VALUE_LENGTH       ((OM_return_code) 25)
#define OM_WRONG_VALUE_MAKEUP       ((OM_return_code) 26)
#define OM_WRONG_VALUE_NUMBER       ((OM_return_code) 27)
#define OM_WRONG_VALUE_POSITION     ((OM_return_code) 28)
#define OM_WRONG_VALUE_SYNTAX       ((OM_return_code) 29)
#define OM_WRONG_VALUE_TYPE         ((OM_return_code) 30)

/* String (Elements component) */

#define OM_ELEMENTS_UNSPECIFIED     ((void *) 0)
```

```
/* Syntax */

#define OM_S_NO_MORE_SYNTAXES         ((OM_syntax) 0)
#define OM_S_BIT_STRING               ((OM_syntax) 3)
#define OM_S_BOOLEAN                  ((OM_syntax) 1)
#define OM_S_ENCODING_STRING          ((OM_syntax) 8)
#define OM_S_ENUMERATION              ((OM_syntax) 10)
#define OM_S_GENERAL_STRING           ((OM_syntax) 27)
#define OM_S_GENERALISED_TIME_STRING  ((OM_syntax) 24)
#define OM_S_GRAPHIC_STRING           ((OM_syntax) 25)
#define OM_S_IA5_STRING               ((OM_syntax) 22)
#define OM_S_INTEGER                  ((OM_syntax) 2)
#define OM_S_NULL                     ((OM_syntax) 5)
#define OM_S_NUMERIC_STRING           ((OM_syntax) 18)
#define OM_S_OBJECT                   ((OM_syntax) 127)
#define OM_S_OBJECT_DESCRIPTOR_STRING ((OM_syntax) 7)
#define OM_S_OBJECT_IDENTIFIER_STRING ((OM_syntax) 6)
#define OM_S_OCTET_STRING             ((OM_syntax) 4)
#define OM_S_PRINTABLE_STRING         ((OM_syntax) 19)
#define OM_S_TELETEX_STRING           ((OM_syntax) 20)
#define OM_S_UTC_TIME_STRING          ((OM_syntax) 23)
#define OM_S_VIDEOTEX_STRING          ((OM_syntax) 21)
#define OM_S_VISIBLE_STRING           ((OM_syntax) 26)

#define OM_S_LONG_STRING              ((OM_syntax) 0x8000)
#define OM_S_NO_VALUE                 ((OM_syntax) 0x4000)
#define OM_S_LOCAL_STRING             ((OM_syntax) 0x2000)
#define OM_S_SERVICE_GENERATED        ((OM_syntax) 0x1000)
#define OM_S_PRIVATE                  ((OM_syntax) 0x0800)
#define OM_S_SYNTAX                   ((OM_syntax) 0x03FF)

/* Type */

#define OM_NO_MORE_TYPES              ((OM_type) 0)
#define OM_ARBITRARY_ENCODING         ((OM_type) 1)
#define OM_ASN1_ENCODING              ((OM_type) 2)
#define OM_CLASS                      ((OM_type) 3)
#define OM_DATA_VALUE_DESCRIPTOR      ((OM_type) 4)
#define OM_DIRECT_REFERENCE           ((OM_type) 5)
#define OM_INDIRECT_REFERENCE         ((OM_type) 6)
```

```
#define OM_OBJECT_CLASS              ((OM_type) 7)
#define OM_OBJECT_ENCODING           ((OM_type) 8)
#define OM_OCTET_ALIGNED_ENCODING    ((OM_type) 9)
#define OM_PRIVATE_OBJECT            ((OM_type) 10)
#define OM_RULES                     ((OM_type) 11)

/* Value Position */

#define OM_ALL_VALUES         ((OM_value_position) 0xFFFFFFFF)

/* WORKSPACE INTERFACE */

#include <xomi.h>          /* Only for internal use by interface */

/* END SERVICE INTERFACE */
#endif   /* XOM_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS), X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM), DCE 1.2.2 Application Development Guide—Directory Services.*

Chapter 5

DCE Distributed Time Service

dts_intro

Purpose

Introduction to DCE Distributed Time Service (DTS)

Description

The DCE Distributed Time Service programming routines can obtain timestamps that are based on Coordinated Universal Time (UTC), translate between different timestamp formats, and perform calculations on timestamps. Applications can call the DTS routines from server or clerk systems and use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling.

The DTS routines can perform the following basic functions:

- Retrieve the current (UTC-based) time from DTS.
- Convert binary timestamps expressed in the **utc** time structure to or from **tm** structure components.
- Convert the binary timestamps expressed in the **utc** time structure to or from **timespec** structure components.
- Convert the binary timestamps expressed in the **utc** time structure to or from ASCII strings.
- Compare two binary time values.
- Calculate binary time values.
- Obtain time zone information.

DTS can convert between several types of binary time structures that are based on different calendars and time unit measurements. DTS uses UTC-based time structures, and can convert other types of time structures to its own presentation of UTC-based time.

Absolute time is an interval on a time scale; absolute time measurements are derived from system clocks or external time-providers. For DTS, absolute times reference the UTC standard and include the inaccuracy and other information. When you display an absolute time, DTS converts the time to ASCII text, as shown in the following display:

```
1992-11-21-13:30:25.785-04:00I000.082
```

Relative time is a discrete time interval that is often added to or subtracted from an absolute time. A TDF associated with an absolute time is one example of a relative time. Note that a relative time does not use the calendar date fields, since these fields concern absolute time.

UTC is the international time standard that DTS uses. The zero hour of UTC is based on the zero hour of Greenwich Mean Time (GMT). The documentation consistently refers to the time zone of the Greenwich Meridian as GMT. However, this time zone is also sometimes referred to as UTC.

The Time Differential Factor (TDF) is the difference between UTC and the time in a particular time zone.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime( )** system call, which is described in the **ctime(3)** reference page, provides additional information.)

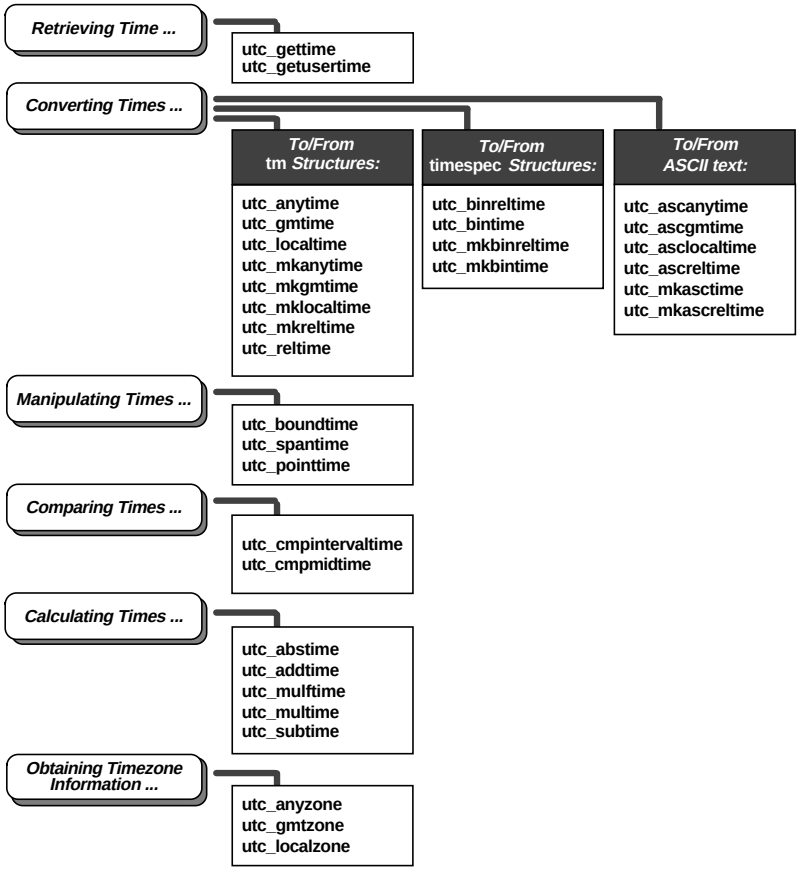
If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

The *DCE 1.2.2 Application Development Guide* provides additional information about UTC and GMT, TDF and time zones, and relative and absolute times.

Unless otherwise specified, the default input and output parameters are as follows:

- If NULL is specified for a *utc* input parameter, the current time is used.
- If NULL is specified for any output parameter, no result is returned.

The following illustration categorizes the DTS portable interface routines by function.



An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

utc_abstime( )
Computes the absolute value of a relative binary timestamp.

utc_addtime( )
Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time.

utc_anytime( )
Converts a binary timestamp to a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure.

utc_anyzone( )
Gets the time zone label and offset from GMT by using the TDF contained in the *utc* input parameter.

utc_ascanytime( )
Converts a binary timestamp to an ASCII string that represents an arbitrary time zone.

utc_ascgmtime( )
Converts a binary timestamp to an ASCII string that expresses a GMT time.

utc_asclocaltime( )
Converts a binary timestamp to an ASCII string that represents a local time.

utc_ascreltime( )
Converts a relative binary timestamp to an ASCII string that represents the time.

utc_binreltime( )
Converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy.

utc_bintime( )
Converts a binary timestamp to a **timespec** structure.

utc_boundtime( )
Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.

utc_cmpintervaltime( )
Compares two binary timestamps or two relative binary timestamps.

utc_cmpmidtime( )
Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.

utc_gettime( )
Returns the current system time and inaccuracy as a binary timestamp.

utc_getusertime( )
Returns the time and process-specific TDF, rather than the system-specific TDF.

utc_gmtime( )
Converts a binary timestamp to a **tm** structure that expresses GMT or the equivalent UTC.

utc_gmtzone( )
Gets the time zone label for GMT.

utc_localtime( )
Converts a binary timestamp to a **tm** structure that expresses local time.

utc_localzone( )
Gets the local time zone label and offset from GMT, given **utc**.

utc_mkanytime( )
Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.

utc_mkascreltime( )
Converts a NULL-terminated character string that represents a relative timestamp to a binary timestamp.

utc_mkasctime( )
Converts a NULL-terminated character string that represents an absolute timestamp to a binary timestamp.

utc_mkbinreltime( )
Converts a **timespec** structure expressing a relative time to a binary timestamp.

utc_mkbintime( )
Converts a **timespec** structure to a binary timestamp.

utc_mkgmtime( )
Converts a **tm** structure that expresses GMT or UTC to a binary timestamp.

utc_mklocaltime( )
Converts a **tm** structure that expresses local time to a binary timestamp.

utc_mkreltime( )
Converts a **tm** structure that expresses relative time to a relative binary timestamp.

utc_mulftime( )
Multiplies a relative binary timestamp by a floating-point value.

utc_multime( )
Multiplies a relative binary timestamp by an integer factor.

utc_pointtime( )
Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.

utc_reltime( )
Converts a relative binary timestamp to a **tm** structure.

utc_spantime( )
Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps.

utc_subtime( )
Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times.

Related Information

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

utc_abstime

Purpose

Computes the absolute value of a relative binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_abstime(
        utc_t* result,
        utc_t *utc);
```

Parameters

Input

utc — Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result — Absolute value of the input relative binary timestamp.

Description

The **utc_abstime()** routine computes the absolute value of a relative binary timestamp. The input timestamp represents a relative (delta) time.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time parameter or invalid results.

Examples

The following example scales a relative time, computes its absolute value, and prints the result.

```
utc_t       relutc, scaledutc;
char        timstr[UTC_MAX_STR_LEN];

/*
 *   Make sure relative timestamp represents a positive interval...
 */

utc_abstime(&relutc,        /* Out: Abs-value of rel time  */
            &relutc);       /* In:  Relative time to scale */

/*
 *   Scale it by a factor of 17...
 */

utc_multime(&scaledutc,     /* Out: Scaled relative time   */
            &relutc,        /* In:  Relative time to scale */
            17L);               /* In:  Scale factor           */

utc_ascreltime(timstr,          /* Out: ASCII relative time    */
               UTC_MAX_STR_LEN, /* In:  Length of input string */
               &scaledutc); /* In:  Relative time to       */
                                /* convert                     */

printf("%s\n",timstr);

/*
 *   Scale it by a factor of 17.65...
 */

utc_mulftime(&scaledutc,    /* Out: Scaled relative time   */
             &relutc,       /* In:  Relative time to scale */
             17.65);            /* In:  Scale factor           */
```

```
utc_ascreltime(timstr,          /* Out: ASCII relative time    */
               UTC_MAX_STR_LEN, /* In:  Length of input string */
               &scaledutc); /* In:  Relative time to       */
                                /* convert                     */

printf("%s\n",timstr);
```

utc_addtime

Purpose

Computes the sum of two binary timestamps

Synopsis

```
#include <dce/utc.h>

int utc_addtime(
        utc_t* result,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Resulting binary timestamp or relative binary timestamp, depending upon the operation performed:

- *relative time+relative time=relative time*
- *absolute time+relative time=absolute time*
- *relative time+absolute time=absolute time*
- *absolute time+absolute time* is undefined. (See the note later in this reference page.)

Description

The **utc_addtime()** routine adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies. One or both of the input timestamps typically represents a relative (delta) time. The TDF in the first input timestamp is copied to the output. The timestamps can be two relative times or a relative time and an absolute time.

Notes

Although no error is returned, the combination *absolute time+absolute time* should *not* be used.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time parameter or invalid results.

Examples

The following example shows how to compute a timestamp that represents a time at least 5 seconds in the future.

```
utc_t               now, future, fivesec;
reltimespec_t       tfivesec;
timespec_t          tzero;

/*   Construct a timestamp that represents 5 seconds...
 */
tfivesec.tv_sec = 5;
tfivesec.tv_nsec = 0;
tzero.tv_sec = 0;
tzero.tv_nsec = 0;
utc_mkbinreltime(&fivesec,  /* Out: 5 secs in binary timestamp    */
                 &tfivesec, /* In:  5 secs in timespec            */
```

```
                &tzero);   /* In:  0 secs inaccuracy in timespec */

/*    Get the maximum possible current time...
 *    (The NULL input parameter is used to specify the current time.)
 */
utc_pointtime((utc_t *)0,     /* Out: Earliest possible current time  */
              (utc_t *)0,     /* Out: Midpoint of current time        */
              &now,       /* Out: Latest possible current time    */
              (utc_t *)0);    /* In:  Use current time                */

/*   Add 5 seconds to get future timestamp...
 */
utc_addtime(&future,      /* Out: Future binary timestamp      */
            &now,         /* In:  Latest possible time now     */
            &fivesec);    /* In:  5 secs                       */
```

Related Information

Functions: **utc_subtime(3dts)**.

utc_anytime

Purpose

Converts a binary timestamp to a **tm** structure

Synopsis

```
#include <dce/utc.h>

int utc_anytime(
        struct tm *timetm,
        long *tns,
        struct tm *inacctm,
        long *ins,
        long *tdf,
        utc_t *utc);
```

Parameters

Input

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp expressed in the timestamp's local time.

tns Nanoseconds since the time component of the binary timestamp.

inacctm Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp.

tdf TDF component of the binary timestamp in units of seconds east of GMT.

Description

The **utc_anytime()** routine converts a binary timestamp to a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure. The TDF information contained in the timestamp is returned with the time and inaccuracy components; the TDF component determines the offset from GMT and the local time value of the **tm** structure. Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

The following example converts a timestamp by using the TDF information in the timestamp, and then prints the result.

```
utc_t               evnt;
struct tm           tmevnt;
timespec_t          tevnt, ievnt;
char                tznam[80];

/*   Assume evnt contains the timestamp to convert...
 *
 *   Get time as a tm structure, using the time zone information in
 *   the timestamp...
 */
utc_anytime(&tmevnt,        /* Out: tm struct of time of evnt    */
            (long *)0,          /* Out: nanosec of time of evnt      */
```

```
            (struct tm *)0,     /* Out: tm struct of inacc of evnt   */
            (long *)0,          /* Out: nanosec of inacc of evnt     */
            (int *)0,           /* Out: tdf of evnt                  */
            &evnt);         /* In:  binary timestamp of evnt     */

/*   Get the time and inaccuracy as timespec structures...
 */
utc_bintime(&tevnt,         /* Out: timespec of time of evnt     */
            &ievnt,         /* Out: timespec of inacc of evnt    */
            (int *)0,           /* Out: tdf of evnt                  */
            &evnt);         /* In:  Binary timestamp of evnt     */

/*   Construct the time zone name from time zone information in the
 *   timestamp...
 */
utc_anyzone(tznam,  /* Out: Time zone name               */
            80,                 /* In:  Size of time zone name       */
            (long *)0,          /* Out: tdf of event                 */
            (long *)0,          /* Out: Daylight saving flag         */
            &evnt);         /* In:  Binary timestamp of evnt     */

/*   Print timestamp in the format:
 *
 *           1991-03-05-21:27:50.023I0.140 (GMT-5:00)
 *           1992-04-02-12:37:24.003Iinf (GMT+7:00)
 */

printf("%d-%02d-%02d-%02d:%02d:%02d.%03d",
        tmevnt.tm_year+1900, tmevnt.tm_mon+1, tmevnt.tm_mday,
        tmevnt.tm_hour, tmevnt.tm_min, tmevnt.tm_sec,
        (tevnt.tv_nsec/1000000));

if ((long)ievnt.tv_sec == -1)
    printf("Iinf");
else
    printf("I%d.%03d", ievnt.tv_sec, (ievnt.tv_nsec/1000000));

printf(" (%s)\n", tznam);
```

Related Information

Functions: **utc_anyzone(3dts), utc_gettime(3dts), utc_getusertime(3dts), utc_gmtime(3dts), utc_localtime(3dts),utc_mkanytime(3dts).**

utc_anyzone

Purpose

Gets the time zone label and offset from GMT

Synopsis

```
#include <dce/utc.h>

int utc_anyzone(
        char *tzname,
        size_t tzlen,
        long *tdf,
        int *isdst,
        const utc_t *utc);
```

Parameters

Input

tzlen Length of the *tzname* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

tzname Character string that is long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT.

isdst Integer with a value of −1, indicating that no information is supplied as to whether it is standard time or daylight saving time. A value of −1 is always returned.

Description

The **utc_anyzone()** routine gets the time zone label and offset from GMT by using the TDF contained in the *utc* input parameter. The label returned is always of the

form GMT+*n* or GMT-*n* where *n* is the *tdf* expressed in *hours*:*minutes*. (The label associated with an arbitrary time zone is not known; only the offset is known.)

Notes

All of the output parameters are optional. No value is returned and no error occurs if the pointer is NULL.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or an insufficient buffer.

Examples

See the sample program in the **utc_anytime(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts)**, **utc_gmtzone(3dts)**, **utc_localzone(3dts)**.

utc_ascanytime

Purpose

Converts a binary timestamp to an ASCII string that represents an arbitrary time zone

Synopsis

```
#include <dce/utc.h>

int utc_ascanytime(
        char *cp,
        size_t stringlen,
        utc_t *utc);
```

Parameters

Input

stringlen The length of the *cp* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

cp ASCII string that represents the time.

Description

The **utc_ascanytime()** routine converts a binary timestamp to an ASCII string that expresses a time. The TDF component in the timestamp determines the local time used in the conversion.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts a time to an ASCII string that expresses the time in the time zone where the timestamp was generated.

```
utc_t      evnt;
char       localTime[UTC_MAX_STR_LEN];

/*
 *  Assuming that evnt contains the timestamp to convert, convert
 *  the time to ASCII in the following format:
 *
 *          1991-04-01-12:27:38.37-8:00I2.00
 */

utc_ascanytime(localtime,         /* Out: Converted time    */
               UTC_MAX_STR_LEN,   /* In:  Length of string  */
               &evnt);        /* In:  Time to convert   */
```

Related Information

Functions: **utc_ascgmtime(3dts)**, **utc_asclocaltime(3dts)**.

utc_ascgmtime

Purpose

Converts a binary timestamp to an ASCII string that expresses a GMT time

Synopsis

```
#include <dce/utc.h>

int utc_ascgmtime(
        char *cp,
        size_t stringlen,
        utc_t *utc);
```

Parameters

Input

stringlen Length of the *cp* buffer.

utc Binary timestamp.

Output

cp ASCII string that represents the time.

Description

The **utc_ascgmtime()** routine converts a binary timestamp to an ASCII string that expresses a time in GMT.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts the current time to GMT format.

```
char   gmTime[UTC_MAX_STR_LEN];

/*   Convert the current time to ASCII in the following format:
 *          1991-04-01-12:27:38.37I2.00
 */
utc_ascgmtime(gmTime,            /* Out: Converted time      */
              UTC_MAX_STR_LEN,   /* In:  Length of string    */
              (utc_t*) NULL);    /* In:  Time to convert     */
                                 /* Default is current time  */
```

Related Information

Functions: **utc_ascanytime(3dts), utc_asclocaltime(3dts)**.

utc_asclocaltime

Purpose

Converts a binary timestamp to an ASCII string that represents a local time

Synopsis

```
#include <dce/utc.h>

int utc_asclocaltime(
        char *cp,
        size_t stringlen,
        utc_t *utc);
```

Parameters

Input

stringlen Length of the *cp* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

cp ASCII string that represents the time.

Description

The **utc_asclocaltime()** routine converts a binary timestamp to an ASCII string that expresses local time.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime**() system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time parameter or invalid results.

Examples

The following example converts the current time to local time.

```
char   localTime[UTC_MAX_STR_LEN];

/*  Convert the current time...
 */

utc_asclocaltime(localTime,        /* Out:  Converted time      */
                 UTC_MAX_STR_LEN,  /* In:   Length of string    */
                 (utc_t*) NULL);   /* In:   Time to convert     */
                                   /* Default is current time   */
```

Related Information

Functions: **utc_ascanytime(3dts)**, **utc_ascgmtime(3dts)**.

utc_ascreltime

Purpose

Converts a relative binary timestamp to an ASCII string that represents the time

Synopsis

```
#include <dce/utc.h>

int utc_ascreltime(
        char *cp,
        const size_t stringlen,
        utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp.

stringlen Length of the *cp* buffer.

Output

cp ASCII string that represents the time.

Description

The **utc_ascreltime()** routine converts a relative binary timestamp to an ASCII string that represents the time.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

See the sample program in the **utc_abstime(3dts)** reference page.

Related Information

Functions: **utc_mkascreltime(3dts)**.

utc_binreltime

Purpose

Converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy

Synopsis

```
#include <dce/utc.h>

int utc_binreltime(
        reltimespec_t *timesp,
        timespec_t *inaccsp,
        utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timesp Time component of the relative binary timestamp, in the form of seconds and nanoseconds since the base time (1970–01–01:00:00:00.0+00:00I0).

inaccsp Inaccuracy component of the relative binary timestamp, in the form of seconds and nanoseconds.

Description

The **utc_binreltime()** routine converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy. These **timespec** structures describe a time interval.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

The following example measures the duration of a process, then prints the resulting relative time and inaccuracy.

```
utc_t               before, duration;
reltimespec_t       tduration;
timespec_t          iduration;

/*   Get the time before the start of the operation...
 */
utc_gettime(&before);           /* Out: Before binary timestamp     */

/*    ...Later...
 *    Subtract, getting the duration as a relative time.
 *
 *         NOTE: The NULL argument is used to obtain the current time.
 */

utc_subtime(&duration,          /* Out: Duration rel bin timestamp  */
            (utc_t *)0,         /* In:  After binary timestamp      */
            &before);           /* In:  Before binary timestamp     */

/*   Convert the relative times to timespec structures...
 */

utc_binreltime(&tduration,      /* Out: Duration time timespec      */
               &iduration,      /* Out: Duration inacc timespec     */
               &duration);      /* In:  Duration rel bin timestamp  */

/*   Print the duration...
 */
```

```
printf("%d.%04d", tduration.tv_sec, (tduration.tv_nsec/10000));

if ((long)iduration.tv_sec == -1)
    printf("Iinf\n");
else
    printf("I%d.%04d\n", iduration.tv_sec, (iduration.tv_nsec/100000));
```

Related Information

Functions: **utc_mkbinreltime(3dts)**.

utc_bintime

Purpose

Converts a binary timestamp to a **timespec** structure

Synopsis

```
#include <dce/utc.h>

int utc_bintime(
        timespec_t *timesp,
        timespec_t *inaccsp,
        long *tdf,
        utc_t *utc);
```

Parameters

Input

utc — Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timesp — Time component of the binary timestamp, in the form of seconds and nanoseconds since the base time.

inaccsp — Inaccuracy component of the binary timestamp, in the form of seconds and nanoseconds.

tdf — TDF component of the binary timestamp in the form of signed number of seconds east of GMT.

Description

The **utc_bintime()** routine converts a binary timestamp to a **timespec** structure. The TDF information contained in the timestamp is returned.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_anytime(3dts)** reference page.

Related Information

Functions: **utc_binreltime(3dts)**, **utc_mkbintime(3dts)**.

utc_boundtime

Purpose

Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event

Synopsis

```
#include <dce/utc.h>

int utc_boundtime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Before binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 After binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Spanning timestamp.

Description

Given two UTC times, the **utc_boundtime()** routine returns a single UTC time whose inaccuracy bounds the two input times. This is useful for timestamping events: the routine gets the **utc** values before and after the event, then calls **utc_boundtime()** to build a timestamp that includes the event.

Notes

The TDF in the output UTC value is copied from the *utc2* input parameter. If one or both input values have unspecified inaccuracies, the returned time value also has an unspecified inaccuracy and is the average of the two input values.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time parameter or invalid parameter order.

Examples

The following example records the time of an event and constructs a single timestamp, which includes the time of the event. Note that the **utc_getusertime()** routine is called so the time zone information that is included in the timestamp references the user's environment rather than the system's default time zone.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime**() system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

```
utc_t               before, after, evnt;

/*  Get the time before the event...
 */
utc_getusertime(&before);       /* Out: Before binary timestamp     */

/*  Get the time after the event...
 */
utc_getusertime(&after);        /* Out: After binary timestamp      */
```

```
/*  Construct a single timestamp that describes the time of the
 *   event...
 */
utc_boundtime(&evnt,           /* Out: Timestamp that bounds event */
              &before,         /* In:  Before binary timestamp     */
              &after);         /* In:  After binary timestamp      */
```

Related Information

Functions: **utc_gettime(3dts)**, **utc_pointtime(3dts)**, **utc_spantime(3dts)**.

utc_cmpintervaltime

Purpose

Compares two binary timestamps or two relative binary timestamps

Synopsis

```
#include <dce/utc.h>

int utc_cmpintervaltime(
        enum utc_cmptype *relation,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

relation Receives the result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

- **utc_equalTo**
- **utc_lessThan**
- **utc_greaterThan**
- **utc_indeterminate**

Description

The **utc_cmpintervaltime()** routine compares two binary timestamps and returns a flag indicating that the first time is greater than, less than, equal to, or overlapping with the second time. Two times overlap if the intervals (*time* − *inaccuracy*, *time* + *inaccuracy*) of the two times intersect.

The input binary timestamps express two absolute or two relative times. Do *not* compare relative binary timestamps to absolute binary timestamps. If you do, no meaningful results and no errors are returned.

The following routine does a temporal ordering of the time intervals.

```
utc1 is utc_lessThan utc2 iff
        utc1.time + utc1.inacc < utc2.time - utc2.inacc

utc1 is utc_greaterThan utc2 iff
        utc1.time - utc1.inacc > utc2.time + utc2.inacc

utc1 utc_equalTo utc2 iff
        utc1.time == utc2.time and
        utc1.inacc == 0 and
        utc2.inacc == 0
```

utc1 is **utc_indeterminate** with respect to **utc2** if the intervals overlap.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument.

Examples

The following example checks to see if the current time is definitely after 13:00 local time.

```
struct tm           tmtime, tmzero;
enum utc_cmptype    relation;
utc_t               testtime;

/*   Zero the tm structure for inaccuracy...
 */
memset(&tmzero, 0, sizeof(tmzero));

/*  Get the current time, mapped to a tm structure...
 *
 *        NOTE: The NULL argument is used to get the current time.
 */
utc_gmtime(&tmtime,         /* Out: Current GMT time in tm struct   */
           (long *)0,       /* Out: Nanoseconds of time             */
           (struct tm *)0,  /* Out: Current inaccuracy in tm struct */
           (long *)0,       /* Out: Nanoseconds of inaccuracy       */
           (utc_t *)0);     /* In:  Current timestamp               */

/*   Alter the tm structure to correspond to 13:00 local time       */
 */

tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

/*  Convert to a binary timestamp...
 */
utc_mkgmtime(&testtime,     /* Out: Binary timestamp of 13:00       */
             &tmtime,       /* In:  1:00 PM in tm struct            */
             0,             /* In:  Nanoseconds of time             */
             &tmzero,       /* In:  Zero inaccuracy in tm struct    */
             0);            /* In:  Nanoseconds of inaccuracy       */

/*   Compare to the current time. Note the use of the NULL argument */
 */
utc_cmpintervaltime(&relation,    /* Out: Comparison relation       */
                    (utc_t *)0,   /* In:  Current timestamp         */
                    &testtime);   /* In:  13:00 PM timestamp        */
```

```
/*   If it is not later - wait, print a message, etc.
 */

if (relation != utc_greaterThan) {

/*
 *     Note: It could be earlier than 13:00 local time or it could be
 *           indeterminate.  If indeterminate, for some applications
 *           it might be worth waiting.
 */
}
```

Related Information

Functions: **utc_cmpmidtime(3dts)**.

utc_cmpmidtime

Purpose

Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies

Synopsis

```
#include <dce/utc.h>

int utc_cmpmidtime(
        enum utc_cmptype *relation,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

relation Result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

- **utc_equalTo**
- **utc_lessThan**
- **utc_greaterThan**

Description

The **utc_cmpmidtime()** routine compares two binary timestamps and returns a flag indicating that the first timestamp is greater than, less than, or equal to the second timestamp. Inaccuracy information is ignored for this comparison; the input values are therefore equivalent to the midpoints of the time intervals described by the input binary timestamps.

The input binary timestamps express two absolute or two relative times. Do *not* compare relative binary timestamps to absolute binary timestamps. If you do, no meaningful results and no errors are returned.

The following routine does a lexical ordering on the time interval midpoints.

```
utc1 is utc_lessThan utc2 iff
        utc1.time < utc2.time

utc1 is utc_greaterThan utc2 iff
        utc1.time > utc2.time

utc1 is utc_equalTo utc2 iff
        utc1.time == utc2.time
```

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument.

Examples

The following example checks if the current time (ignoring inaccuracies) is after 13:00 local time.

```
struct tm           tmtime, tmzero;
enum utc_cmptype    relation;
```

```
utc_t               testtime;

/*   Zero the tm structure for inaccuracy...
 */
memset(&tmzero, 0, sizeof(tmzero));

/*   Get the current time, mapped to a tm structure...
 *
 *        NOTE:  The NULL argument is used to get the current time.
 */
utc_localtime(&tmtime,        /* Out: Current local time in tm struct */
              (long *)0,      /* Out: Nanoseconds of time             */
              (struct tm *)0, /* Out: Current inacc in tm struct      */
              (long *)0,      /* Out: Nanoseconds of inaccuracy       */
              (utc_t *)0);    /* In:  Current timestamp               */

 /*   Alter the tm structure to correspond to 13:00 local time.
  */
tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

 /*   Convert to a binary timestamp...
  */
utc_mklocaltime(&testtime,    /* Out: Binary timestamp of 13:00       */
                &tmtime,      /* In:  13:00 in tm struct              */
                0,            /* In:  Nanoseconds of time             */
                &tmzero,      /* In:  Zero inaccuracy in tm struct    */
                0);           /* In:  Nanoseconds of inaccuracy       */

 /*   Compare to the current time.  Note the use of the NULL argument
  */
utc_cmpmidtime(&relation,     /* Out: Comparison relation             */
               (utc_t *)0,    /* In:  Current timestamp               */
               &testtime);    /* In:  13:00 local time timestamp      */

 /*   If the time is not later - wait, print a message, etc.
  */
if (relation != utc_greaterThan) {
```

```
 /*          It is not later then 13:00 local time.  Note that
  *          this depends on the setting of the user's environment.
  */
}
```

Related Information

Functions: **utc_cmpintervaltime(3dts)**.

utc_gettime

Purpose

Returns the current system time and inaccuracy as a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_gettime(
        utc_t *utc);
```

Parameters

Input

None.

Output

utc System time as a binary timestamp.

Description

The **utc_gettime()** routine returns the current system time and inaccuracy in a binary timestamp. The routine takes the TDF from the operating system's kernel; the TDF is specified in a system-dependent manner.

Return Values

0	Indicates that the routine executed successfully.
–1	Generic error that indicates the time service cannot be accessed.

Examples

See the sample program in the **utc_binreltime(3dts)** reference page.

utc_getusertime

Purpose

Returns the time and process-specific TDF, rather than the system-specific TDF

Synopsis

```
#include <dce/utc.h>

int utc_getusertime(
        utc_t *utc);
```

Parameters

Input

None.

Output

utc System time as a binary timestamp.

Description

The **utc_getusertime()** routine returns the system time and inaccuracy in a binary timestamp. The routine takes the TDF from the user's environment, which determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime()** system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user environment does not specify a TDF, the system's TDF is used. The system's time zone rule is applied (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Return Values

0	Indicates that the routine executed successfully.
−1	Generic error that indicates the time service cannot be accessed.

Examples

See the sample program in the **utc_boundtime(3dts)** reference page.

Related Information

Functions: **utc_gettime(3dts)**.

utc_gmtime

Purpose

Converts a binary timestamp to a **tm** structure that expresses GMT or the equivalent UTC

Synopsis

```
#include <dce/utc.h>

int utc_gmtime(
        struct tm *timetm,
        long *tns,
        struct tm *inacctm,
        long *ins,
        utc_t *utc);
```

Parameters

Input

utc Binary timestamp to be converted to **tm** structure components. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp.

tns Nanoseconds since the time component of the binary timestamp.

inacctm Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp. If the inaccuracy is unspecified, *ins* returns a value of −1.

Description

The **utc_gmtime()** routine converts a binary timestamp to a **tm** structure that expresses GMT (or the equivalent UTC). Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpintervaltime(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts), utc_gmtzone(3dts)**, **utc_localtime(3dts)**, **utc_mkgmtime(3dts)**.

utc_gmtzone

Purpose

Gets the time zone label for GMT

Synopsis

```
#include <dce/utc.h>

int utc_gmtzone(
        char *tzname,
        size_t tzlen,
        long *tdf,
        int *isdst,
        utc_t *utc);
```

Parameters

Input

tzlen Length of buffer *tzname*.

utc Binary timestamp. This parameter is ignored.

Output

tzname Character string long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT. A value of 0 (zero) is always returned.

isdst Integer with a value of 0 (zero), indicating that daylight saving time is not in effect. A value of 0 (zero) is always returned.

Description

The **utc_gmtzone()** routine gets the time zone label and zero offset from GMT. Outputs are always *tdf*=**0** and *tzname*=**GMT**. This routine exists for symmetry with the

utc_anyzone() and the **utc_localzone()** routines. Use NULL if you want this routine to use the current time for this parameter.

Notes

All of the output parameters are optional. No value is returned and no error occurs if the *tzname* pointer is NULL.

Return Values

0 Indicates that the routine executed successfully (always returned).

Examples

The following example prints out the current time in both local time and GMT time.

```
utc_t       now;
struct tm   tmlocal, tmgmt;
long        tzoffset;
int         tzdaylight;
char        tzlocal[80], tzgmt[80];

/*  Get the current time once, so both conversions use the same
 *   time...
 */
utc_gettime(&now);

/*  Convert to local time, using the process TZ environment
 *   variable...
 */
utc_localtime(&tmlocal,       /* Out: Local time tm structure       */
              (long *)0,      /* Out: Nanosec of time               */
              (struct tm *)0, /* Out: Inaccuracy tm structure       */
              (long *)0,      /* Out: Nanosec of inaccuracy         */
              (int *)0,       /* Out: TDF of local time             */
```

```
                &now);          /* In:  Current timestamp (ignore)    */



/*   Get the local time zone name, offset from GMT, and current
 *   daylight savings flag...
 */
utc_localzone(tzlocal,          /* Out: Local time zone name          */
              80,               /* In:  Length of loc time zone name  */
              &tzoffset,        /* Out: Loc time zone offset in secs  */
              &tzdaylight,      /* Out: Local time zone daylight flag */
              &now);            /* In:  Current binary timestamp      */

/*   Convert to GMT...
 */
utc_gmtime(&tmgmt,              /* Out: GMT tm structure              */
           (long *)0,           /* Out: Nanoseconds of time           */
           (struct tm *)0,      /* Out: Inaccuracy tm structure       */
           (long *)0,           /* Out: Nanoseconds of inaccuracy     */
           &now);               /* In:  Current binary timestamp      */

/*   Get the GMT time zone name...
 */
utc_gmtzone(tzgmt,              /* Out: GMT time zone name            */
            80,                 /* In:  Size of GMT time zone name    */
            (long *)0,          /* Out: GMT time zone offset in secs  */
            (int *)0,           /* Out: GMT time zone daylight flag   */
            &now);              /* In:  Current binary timestamp     */
         /*      (ignore)        */

/*   Print out times and time zone information in the following
 *   format:
 *
 *          12:00:37 (EDT) = 16:00:37 (GMT)
 *          EDT is -240 minutes ahead of Greenwich Mean Time.
 *          Daylight savings time is in effect.
 */
printf("%d:%02d:%02d (%s) = %d:%02d:%02d (%s)\n",
       tmlocal.tm_hour, tmlocal.tm_min, tmlocal.tm_sec, tzlocal,
       tmgmt.tm_hour, tmgmt.tm_min, tmgmt.tm_sec, tzgmt);
```

```
printf("%s is %d minutes ahead of Greenwich Mean Time\n", tzlocal,
       tzoffset/60);
if (tzdaylight != 0)
    printf("Daylight savings time is in effect\n");
```

Related Information

Functions: **utc_anyzone(3dts)**, **utc_gmtime(3dts)**, **utc_localzone(3dts)**.

utc_localtime

Purpose

Converts a binary timestamp to a **tm** structure that expresses local time

Synopsis

```
#include <dce/utc.h>

int utc_localtime(
        struct tm *timetm,
        long *tns,
        struct tm *inacctm,
        long *ins,
        utc_t *utc);
```

Parameters

Input

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp, expressing local time.

tns Nanoseconds since the time component of the binary timestamp.

inacctm Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp. If the inaccuracy is unspecified, *ins* returns a value of −1.

Description

The **utc_localtime()** routine converts a binary timestamp to a **tm** structure that expresses local time.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime**() system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_gmtzone(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts)**, **utc_gmtime(3dts)**, **utc_localzone(3dts)**, **utc_mklocaltime(3dts)**.

utc_localzone

Purpose

Gets the local time zone label and offset from GMT, given **utc**

Synopsis

```
#include <dce/utc.h>

int utc_localzone(
        char *tzname,
        size_t tzlen,
        long *tdf,
        int *isdst,
        utc_t *utc);
```

Parameters

Input

tzlen Length of the *tzname* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

tzname Character string long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT.

isdst Integer with a value of 0 (zero) if standard time is in effect or a value of 1 if daylight saving time is in effect.

Description

The **utc_localzone()** routine gets the local time zone label and offset from GMT, given **utc**.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime**() system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Notes

All of the output parameters are optional. No value is returned and no error occurs if the pointer is NULL.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or an insufficient buffer.

Examples

See the sample program in the **utc_gmtzone(3dts)** reference page.

Related Information

Functions: **utc_anyzone(3dts)**, **utc_gmtzone(3dts)**, **utc_localtime(3dts)**.

utc_mkanytime

Purpose

Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkanytime(
        utc_t *utc,
        struct tm *timetm,
        long tns,
        struct tm *inacctm,
        long ins,
        long tdf);
```

Parameters

Input

timetm A **tm** structure that expresses the local time; **tm_wday** and **tm_yday** are ignored on input; the value of **tm_isdt** should be −1.

tns Nanoseconds since the time component.

inacctm A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified; **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored on input.

ins Nanoseconds of the inaccuracy component.

tdf Time differential factor to use in conversion.

Output

utc Resulting binary timestamp.

Description

The **utc_mkanytime()** routine converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp. Required inputs include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

The following example converts a string ISO format time in an arbitrary time zone to a binary timestamp. This may be part of an input timestamp routine, although a real implementation will include range checking.

```
utc_t       utc;
struct tm   tmtime, tminacc;
float       tsec, isec;
double      tmp;
long        tnsec, insec;
int         i, offset, tzhour, tzmin, year, mon;
char        *string;

/*  Try to convert the string...                              */

if(sscanf(string, "%d-%d-%d:%d:%e+%d:%dI%e",
          &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
          &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {

/*  Try again with a negative TDF...                          */

if (sscanf(string, "%d-%d-%d-%d:%d:%e-%d:%dI%e",
           &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
           &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {
```

```
/*  ERROR                                                       */

        exit(1);
    }

/*  TDF is negative                                             */

    tzhour = -tzhour;
    tzmin = -tzmin;

}

/*  Fill in the fields...                                       */

tmtime.tm_year = year - 1900;
tmtime.tm_mon = --mon;
tmtime.tm_sec = tsec;
tnsec = (modf(tsec, &tmp)*1.0E9);
offset = tzhour*3600 + tzmin*60;
tminacc.tm_sec = isec;
insec = (modf(isec, &tmp)*1.0E9);

/* Convert to a binary timestamp...                             */

utc_mkanytime(&utc,       /* Out: Resultant binary timestamp       */
              &tmtime,    /* In:  tm struct that represents input  */
              tnsec,      /* In:  Nanoseconds from input           */
              &tminacc,   /* In:  tm struct that represents inacc  */
              insec,      /* In:  Nanoseconds from input           */
              offset);    /* In:  TDF from input                   */
```

Related Information

Functions: **utc_anytime(3dts)**, **utc_anyzone(3dts)**.

utc_mkascreltime

Purpose

Converts a NULL-terminated character string that represents a relative timestamp to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkascreltime(
        utc_t *utc,
        char *string);
```

Parameters

Input

string A NULL-terminated string that expresses a relative timestamp in its ISO format.

Output

utc Resulting binary timestamp.

Description

The **utc_mkascreltime()** routine converts a NULL-terminated string, which represents a relative timestamp, to a binary timestamp.

Notes

The ASCII string must be NULL-terminated.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time parameter or invalid results.

Examples

The following example converts an ASCII relative time string to its binary equivalent.

```
utc_t      utc;
char       str[UTC_MAX_STR_LEN];

/*   Relative time of -333 days, 12 hours, 1 minute, 37.223 seconds
 *   Inaccuracy of 50.22 seconds in the format:  -333-12:01:37.223I50.22
 */
(void)strcpy((void *)str,
             "-333-12:01:37.223I50.22");

utc_mkascreltime(&utc,   /* Out: Binary utc                    */
                 str);   /* In:  String                        */
```

Related Information

Functions: **utc_ascreltime(3dts)**.

utc_mkasctime

Purpose

Converts a NULL-terminated character string that represents an absolute timestamp to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkasctime(
        utc_t *utc,
        char *string);
```

Parameters

Input

string A NULL-terminated string that expresses an absolute time.

Output

utc Resulting binary timestamp.

Description

The **utc_mkasctime()** routine converts a NULL-terminated string that represents an absolute time to a binary timestamp.

Notes

The ASCII string must be NULL-terminated.

Return Values

0 Indicates that the routine executed successfully.

−1	Indicates an invalid time parameter or invalid results.

Examples

The following example converts an ASCII time string to its binary equivalent.

```
utc_t     utc;
char      str[UTC_MAX_STR_LEN];

/*   July 4, 1776, 12:01:37.223 local time
 *   TDF of -5:00 hours
 *   Inaccuracy of 3600.32 seconds
 */
(void)strcpy((void *)str,
             "1776-07-04-12:01:37.223-5:00I3600.32");

utc_mkasctime(&utc,    /* Out: Binary utc            */
              str);    /* In:  String                */
```

Related Information

Functions: **utc_ascanytime(3dts)**, **utc_ascgmtime(3dts)**, **utc_asclocaltime(3dts)**.

utc_mkbinreltime

Purpose

Converts a **timespec** structure expressing a relative time to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkbinreltime(
        utc_t *utc,
        reltimespec_t *timesp,
        timespec_t *inaccsp);
```

Parameters

Input

timesp A **reltimespec** structure that expresses a relative time.

inaccsp A **timespec** structure that expresses inaccuracy. If a null pointer is passed, or if **tv_sec** is set to a value of –1, the inaccuracy is considered to be unspecified.

Output

utc Resulting relative binary timestamp.

Description

The **utc_mkbinreltime()** routine converts a **timespec** structure that expresses relative time to a binary timestamp.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_addtime(3dts)** reference page.

Related Information

Functions: **utc_binreltime(3dts)**, **utc_mkbintime(3dts)**.

utc_mkbintime

Purpose

Converts a **timespec** structure to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkbintime(
        utc_t *utc,
        timespec_t *timesp,
        timespec_t *inaccsp,
        long tdf);
```

Parameters

Input

timesp A **timespec** structure that expresses time since 1970–01–01:00:00:00.0+00:00I0.

inaccsp A **timespec** structure that expresses inaccuracy. If a null pointer is passed, or if **tv_sec** is set to a value of –1, the inaccuracy is considered to be unspecified.

tdf TDF component of the binary timestamp.

Output

utc Resulting binary timestamp.

Description

The **utc_mkbintime()** routine converts a **timespec** structure time to a binary timestamp. The TDF input is used as the TDF of the binary timestamp.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or invalid results.

Examples

The following example obtains the current time from **time(3)**, converts it to a binary timestamp with an inaccuracy of 5.2 seconds, and specifies GMT.

```
timespec_t   ttime, tinacc;
utc_t        utc;

/*   Obtain the current time (without the inaccuracy)...
 */

ttime.tv_sec = time((time_t *)0);
ttime.tv_nsec = 0;

/*   Specify the inaccuracy...
 */

tinacc.tv_sec = 5;
tinacc.tv_nsec = 200000000;

/*   Convert to a binary timestamp...
 */

utc_mkbintime(&utc,       /* Out: Binary timestamp          */
              &ttime,     /* In:  Current time in timespec  */
              &tinacc,    /* In:  5.2 seconds in timespec   */
              0);         /* In:  TDF of GMT                */
```

Related Information

Functions: **utc_bintime(3dts)**, **utc_mkbinreltime(3dts)**.

utc_mkgmtime

Purpose

Converts a **tm** structure that expresses GMT or UTC to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkgmtime(
        utc_t *utc,
        struct tm *timetm,
        long tns,
        struct tm *inacctm,
        long ins);
```

Parameters

Input

timetm — A **tm** structure that expresses GMT. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdt** should be −1.

tns — Nanoseconds since the time component.

inacctm — A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored.

ins — Nanoseconds of the inaccuracy component.

Output

utc — Resulting binary timestamp.

Description

The **utc_mkgmtime()** routine converts a **tm** structure that expresses GMT or UTC to a binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpintervaltime(3dts)** reference page.

Related Information

Functions: **utc_gmtime(3dts)**.

utc_mklocaltime

Purpose

Converts a **tm** structure that expresses local time to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mklocaltime(
        utc_t *utc,
        struct tm *timetm,
        long tns,
        struct tm *inacctm,
        long ins);
```

Parameters

Input

timetm A **tm** structure that expresses the local time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** should be −1.

tns Nanoseconds since the time component.

inacctm A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored.

ins Nanoseconds of the inaccuracy component.

Output

utc Resulting binary timestamp.

Description

The **utc_mklocaltime()** routine converts a **tm** structure that expresses local time to a binary timestamp.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime**() system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpmidtime(3dts)** reference page.

Related Information

Functions: **utc_localtime(3dts)**.

utc_mkreltime

Purpose

Converts a **tm** structure that expresses relative time to a relative binary timestamp

Synopsis

```
#include <dce/utc.h>

int utc_mkreltime(
        utc_t *utc,
        struct tm *timetm,
        long tns,
        struct tm *inacctm,
        long ins);
```

Parameters

Input

timetm — A **tm** structure that expresses a relative time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** should be −1.

tns — Nanoseconds since the time component.

inacctm — A **tm** structure that expresses seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_year**, **tm_wday**, **tm_isdst**, and **tm_zone** are ignored.

ins — Nanoseconds of the inaccuracy component.

Output

utc — Resulting relative binary timestamp.

Description

The **utc_mkreltime()** routine converts a **tm** structure that expresses relative time to a relative binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

The following example converts the relative time **125-03:12:30.1I120.25** to a relative binary timestamp.

```
utc_t       utc;
struct tm   tmtime,tminacc;
long        tnsec,insec;

    /* Fill in the fields
     */
    memset((void *)&tmtime,0,sizeof(tmtime));
    tmtime.tm_mday = 125;
    tmtime.tm_hour = 3;
    tmtime.tm_min  = 12;
    tmtime.tm_sec  = 30;
    tnsec = 100000000;     /* .1 * 1.0E9 */

    memset((void *)&tminacc,0,sizeof(tminacc));
    tminacc.tm_sec = 120;
    tnsec = 250000000;     /* .25 * 1.0E9 */

/* Convert to a relative binary timestamp...
 */
utc_mkreltime(&utc,       /* Out: Resultant relative binary timestamp */
```

```
&tmtime,    /* In:  tm struct that represents input     */
tnsec,      /* In:  Nanoseconds from input              */
&tminacc,   /* In:  tm struct that represents inacc     */
insec);     /* In:  Nanoseconds from input              */
```

utc_mulftime

Purpose

Multiplies a relative binary timestamp by a floating-point value

Synopsis

```
#include <dce/utc.h>

int utc_mulftime(
        utc_t *result,
        utc_t *utc1,
        double factor);
```

Parameters

Input

utc1 Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

factor Real scale factor (double-precision, floating-point value).

Output

result Resulting relative binary timestamp.

Description

The **utc_mulftime()** routine multiplies a relative binary timestamp by a floating-point value. Either or both may be negative; the resulting relative binary timestamp has the appropriate sign. The unsigned inaccuracy in the relative binary timestamp is also multiplied by the absolute value of the floating-point value.

Return Values

0 Indicates that the routine executed successfully.

−1	Indicates an invalid time argument or invalid results.

Examples

The following example scales a relative time by a floating-point factor and prints the result.

```
utc_t       relutc, scaledutc;
struct tm   scaledreltm;
char        timstr[UTC_MAX_STR_LEN];

/*   Assume relutc contains the time to scale.
 */
utc_mulftime(&scaledutc,           /* Out: Scaled rel time     */
             &relutc,              /* In:  Rel time to scale   */
             17.65);                   /* In:  Scale factor        */

utc_ascreltime(timstr,                 /* Out: ASCII rel time      */
               UTC_MAX_STR_LEN,        /* In:  Input buffer length */
               &scaledutc);        /* In:  Rel time to convert */

printf("%s\n",timstr);

/*    Convert it to a tm structure and print it.
 */
utc_reltime(&scaledreltm,          /* Out: Scaled rel tm       */
            (long *)0,                 /* Out: Scaled rel nano-sec */
            (struct tm *)0,            /* Out: Scaled rel inacc tm */
            (long *)0,                 /* Out: Scd rel inacc nanos */
            &scaledutc);           /* In:  Rel time to convert */

printf("Approximately %d days, %d hours and %d minutes\n",
       scaledreltm.tm_yday, scaledreltm.tm_hour, scaledreltm.tm_min);
```

Related Information

Functions: **utc_multime(3dts)**.

utc_multime

Purpose

Multiplies a relative binary timestamp by an integer factor

Synopsis

```
#include <dce/utc.h>

int utc_multime(
        utc_t *result,
        utc_t *utc1,
        long factor);
```

Parameters

Input

utc1 Relative binary timestamp.

factor Integer scale factor.

Output

result Resulting relative binary timestamp.

Description

The **utc_multime()** routine multiplies a relative binary timestamp by an integer. Either or both may be negative; the resulting binary timestamp has the appropriate sign. The unsigned inaccuracy in the binary timestamp is also multiplied by the absolute value of the integer.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

Examples

The following example scales a relative time by an integral value and prints the result.

```
utc_t       relutc, scaledutc;

char        timstr[UTC_MAX_STR_LEN];

/*   Assume relutc contains the time to scale.
 *   Scale it by a factor of 17 ...
 */
utc_multime(&scaledutc,     /* Out: Scaled rel time       */
        &relutc,     /*  In: Rel time to scale     */
            17L);        /*  In: Scale factor          */

utc_ascreltime(timstr,          /* Out: ASCII rel time        */
      UTC_MAX_STR_LEN,          /*  In: Input buffer length   */
           &scakedutc);     /*  In: Rel time to convert   */

printf("Scaled result is %s, timstr);
```

Related Information

Functions: **utc_mulftime(3dts)**.

utc_pointtime

Purpose

Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time

Synopsis

```
#include <dce/utc.h>

int utc_pointtime(
        utc_t *utclp,
        utc_t *utcmp,
        utc_t *utchp,
        utc_t *utc);
```

Parameters

Input

utc Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

utclp Lowest (earliest) possible absolute time or shortest possible relative time that the input timestamp can represent.

utcmp Midpoint of the input timestamp.

utchp Highest (latest) possible absolute time or longest possible relative time that the input timestamp can represent.

Description

The **utc_pointtime()** routine converts a binary timestamp to three binary timestamps that represent the earliest, latest, and most likely (midpoint) times. If the input is a relative binary time, the outputs represent relative binary times.

Notes

All outputs have zero inaccuracy. An error is returned if the input binary timestamp has an unspecified inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument.

Examples

See the sample program in the **utc_addtime(3dts)** reference page.

Related Information

Functions: **utc_boundtime(3dts)**, **utc_spantime(3dts)**.

utc_reltime

Purpose

Converts a relative binary timestamp to a **tm** structure

Synopsis

```
#include <dce/utc.h>

int utc_reltime(
        struct tm *timetm,
        long *tns,
        struct tm *inacctm,
        long *ins,
        utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp.

Output

timetm Relative time component of the relative binary timestamp. The field **tm_mday** returns a value of −1 and the fields **tm_year** and **tm_mon** return values of 0 (zero). The field **tm_yday** contains the number of days of relative time.

tns Nanoseconds since the time component of the relative binary timestamp.

inacctm Seconds of the inaccuracy component of the relative binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the relative binary timestamp.

Description

The **utc_reltime()** routine converts a relative binary timestamp to a **tm** structure. Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_mulftime(3dts)** reference page.

Related Information

Functions: **utc_mkreltime(3dts)**.

utc_spantime

Purpose

Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps

Synopsis

```
#include <dce/utc.h>

int utc_spantime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Spanning timestamp.

Description

Given two binary timestamps, the **utc_spantime()** routine returns a single UTC time interval whose inaccuracy spans the two input timestamps (that is, the interval resulting from the earliest possible time of either timestamp to the latest possible time of either timestamp).

Notes

The *tdf* parameter in the output UTC value is copied from the *utc2* input. If either input binary timestamp has an unspecified inaccuracy, an error is returned.

Return Values

0	Indicates that the routine executed successfully.
−1	Indicates an invalid time argument.

Examples

The following example computes the earliest and latest times for an array of 10 timestamps.

```
utc_t               time_array[10], testtime, earliest, latest;
int                 i;

/*   Set the running timestamp to the first entry...
 */
testtime = time_array[0];

for (i=1; i<10; i++) {

   /*   Compute the minimum and the maximum against the next
    *   element...
    */
utc_spantime(&testtime,        /* Out: Resultant interval          */
             &testtime,        /* In:  Largest previous interval   */
             &time_array[i]);  /* In:  Element under test          */
}

/*   Compute the earliest and latest possible times
 */

utc_pointtime(&earliest,       /* Out: Earliest poss time in array */
             (utc_t *)0,       /* Out: Midpoint                    */
```

```
    &latest,          /* Out: Latest poss time in array   */
    &testtime);       /* In:  Spanning interval           */
```

Related Information

Functions: **utc_boundtime(3dts)**, **utc_gettime(3dts)**, **utc_pointtime(3dts)**.

utc_subtime

Purpose

Computes the difference between two binary timestamps

Synopsis

```
#include <dce/utc.h>

int utc_subtime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Resulting binary timestamp or relative binary timestamp, depending upon the operation performed:

- *absolute time – absolute time = relative time*
- *relative time – relative time = relative time*
- *absolute time – relative time = absolute time*
- *relative time – absolute time* is undefined. (See the note later in this reference page.)

Description

The **utc_subtime()** routine subtracts one binary timestamp from another. The two binary timestamps express either an absolute time and a relative time, two relative times, or two absolute times. The resulting timestamp is *utc1* minus *utc2*. The inaccuracies of the two input timestamps are combined and included in the output timestamp. The TDF in the first timestamp is copied to the output.

Notes

Although no error is returned, the combination *relative time–absolute time* should *not* be used.

Return Values

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_binreltime(3dts)** reference page.

Related Information

Functions: **utc_addtime(3dts)**.

Chapter 6

DCE Security Service

sec_intro

Purpose

Application program interface to the DCE Security Service

Description

The DCE Security Service application program interface (API) allows developers to create network services with complete access to all the authentication and authorization capabilities of DCE Security Service and facilities.

The transaction of a network service generally consists of a client process requesting some action from a server process. The client may itself be a server, or a user, and the server may also be a client of other servers. Before the targeted server executes the specified action, it must be sure of the client's identity, and it must know whether the client is authorized to request the service.

The security service API consists of the following sets of remote procedure calls (RPCs) used to communicate with various security-related services and facilities:

rgy Maintains the network registry of principal identities.

era Maintains extended registry attributes.

login Validates a principal's network identity and establish delegated identities.

epa Extracts privilege attributes from an opaque binding handle.

acl Implements an access control list (ACL) protocol for the authorization of a principal to network access and services.

key Provides facilities for the maintenance of account keys for daemon principals.

id Maps file system names to universal unique IDs (UUIDs).

pwd_mgmt Provides facilities for password management.

pk Provides facilities for public key authentication.

All the calls in this API have names beginning with the **sec_** prefix. These are the same calls used by various user-level tools provided as part of the DCE. For example,

the **sec_create_db(1)** tool is written with **sec_rgy** calls, **acl_edit(1)** is written with **sec_acl** calls, and the **login(1)** program, with which a user logs in to a DCE system, is written using **sec_login** calls. Most sites will find the user-level tools adequate for their needs, and only must use the security service API to customize or replace the functionality of these tools.

Though most of the calls in the security service API represent RPC transactions, code has been provided on the client side to handle much of the overhead involved with making remote calls. These *stubs* handle binding to the requested security server site, the marshalling of data into whatever form is needed for transmission, and other bookkeeping involved with these remote calls. An application programmer can use the security service interfaces as if they were composed of simple C functions.

This reference page introduces each of the following APIs:

- Registry APIs
- Login APIs
- Extended privilege attributes APIs
- Extended registry attributes APIs
- ACL APIs
- Key management APIs
- ID mapping APIs
- Password management APIs
- Public Key APIs

The section for each API is organized as follows:

- Synopsis
- Data Types
- Constants
- Files

Registry API Data Types

Synopsis

#include <dce/rgybase.h>

Data Types

The following data types are used in **sec_rgy_*** calls:

sec_rgy_handle_t
A pointer to the registry server handle. The registry server is bound to a handle with the **sec_rgy_site_open()** routine.

sec_rgy_bind_auth_info_type_t
A enumeration that defines whether or not the binding is authenticated. This data type is used in conjunction with the **sec_rgy_bind_auth_info_t** data type to set up the authorization method and parameters for a binding. The **sec_rgy_bind_auth_info_type_t** type consists of the following elements:

sec_rgy_bind_auth_none
The binding is not authenticated.

sec_rgy_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_rgy_bind_auth_info_t
A discriminated union that defines authorization and authentication parameters for a binding. This data type is used in conjunction with the **sec_rgy_bind_auth_info_type_t** data type to set up the authorization method and parameters for a binding. The **sec_rgy_bind_auth_info_t** data type consists of the following elements:

info_type A **sec_rgy_bind_auth_info_type_t** data type that specifies whether or not the binding is authenticated. The contents of the union depend on the value of **sec_rgy_bind_auth_info_type_t**.

For unauthenticated bindings (**sec_rgy_bind_auth_info_type_t** = **sec_rgy_bind_auth_none**), no parameters are supplied.

For authenticated bindings (**sec_rgy_bind_auth_info_type_t** = **sec_rgy_bind_auth_dce**), the **dce_info** structure is supplied.

dce_info A structure that consists of the following elements:

authn_level An unsigned 32-bit integer indicating the protection level for RPC calls made using the server binding handle. The protection level determines the degree to which authenticated communications between the client and the server are protected by the authentication service specified by **authn_svc**.

If the RPC runtime or the RPC protocol in the bound protocol sequence does not support a specified level, the level is automatically upgraded to the next higher supported level. The possible protection levels are as follows:

Protection Level	Description
rpc_c_protect_level_default	Uses the default protection level for the specified authentication service. The default protection level for DCE shared-secret key authentication is **rpc_c_protect_level_pkt_value**.
rpc_c_protect_level_none	Performs no authentication: tickets are not exchanged, session keys are not established, client PACs or names are not certified, and transmissions are in the clear. Note that although uncertified PACs should not be trusted, they may be useful for debugging, tracing, and measurement purposes.

rpc_c_protect_level_connect	Authenticates only when the client establishes a relationship with the server.
rpc_c_protect_level_call	Authenticates only at the beginning of each remote procedure call when the server receives the request. This level does not apply to remote procedure calls made over a connection-based protocol sequence (that is, **ncacn_ip_tcp**). If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.
rpc_c_protect_level_pkt	Ensures that all data received is from the expected client.

Protection Level	**Description**
rpc_c_protect_level_pkt_integ	Ensures and verifies that none of the data transferred between client and server has been modified. This is the highest protection level that is guaranteed to be present in the RPC runtime.
rpc_c_protect_level_pkt_privacy	Authenticates as specified by all of the previous levels and also encrypts each RPC argument value. This is the highest protection level, but is not guaranteed to be present in the RPC runtime.

authn_svc Specifies the authentication service to use. The exact level of protection provided by the authentication service is specified by *protect_level*. The supported authentication services are as follows:

Authentication Service	Description
rpc_c_authn_none	No authentication: no tickets are exchanged, no session keys established, client PACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for remote procedure calls made using this binding.
rpc_c_authn_dce_secret	DCE shared-secret key authentication.
rpc_c_authn_default	Default authentication service. The current default authentication service is DCE shared-secret key; therefore, specifying **rpc_c_authn_default** is equivalent to specifying **rpc_c_authn_dce_secret**.
rpc_c_authn_dce_public	DCE public key authentication (reserved for future use).

authz_svc Specifies the authorization service implemented by the server for the interface. The validity and trustworthiness of authorization data, like any application data, is dependent on the authentication service and protection level specified. The supported authorization services are as follows:

Authentication Service	Description
rpc_c_authz_none	Server performs no authorization. This is valid only if **authn_svc** is set to **rpc_c_authn_none**, specifying that no authentication is being performed.
rpc_c_authz_name	Server performs authorization based on the client principal name. This value cannot be used if **authn_svc** is **rpc_c_authn_none**.
rpc_c_authz_dce	Server performs authorization using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with this binding. Generally, access is checked against DCE access control lists (ACLs).

identity A value of type **sec_login_handle_t** that represents a complete login context.

sec_timeval_sec_t
A 32-bit integer containing the seconds portion of a UNIX **timeval_t**, to be used when expressing absolute dates.

sec_timeval_t
A structure containing the full UNIX time. The structure contains two 32-bit integers that indicate seconds (**sec**) and microseconds (**usec**) since 0:00, January 1, 1970.

sec_timeval_period_t
A 32-bit integer expressing seconds relative to some well-known time.

sec_rgy_acct_key_t
Specifies how many parts (person, group, organization) of an account login name will be enough to specify a unique abbreviation for that account.

sec_rgy_cursor_t
A structure providing a pointer into a registry database. This type is used for iterative operations on the registry information. For example, a call to **sec_rgy_pgo_get_members()** might return the 10 account names following the input **sec_rgy_cursor_t** position. Upon return, the cursor position will have been updated, so the next call to that routine will

return the next 10 names. The components of this structure are not used by application programs.

sec_rgy_pname_t
A character string of length **sec_rgy_pname_t_size**.

sec_rgy_name_t
A character string of length **sec_rgy_name_t_size**.

sec_rgy_login_name_t
A structure representing an account login name. It contains three strings of type **sec_rgy_name_t**:

pname The person name for the account.

gname The group name for the account.

oname The organization name for the account.

sec_rgy_member_t
A character string of length **sec_rgy_name_t_size**.

sec_rgy_foreign_id_t
The representation of a foreign ID. This structure contains two components:

cell A string of type **uuid_t** representing the UUID of the foreign cell.

principal A string of type **uuid_t** representing the UUID of the principal.

sec_rgy_sid_t
A structure identifying an account. It contains three fields:

person The UUID of the person part of the account.

group The UUID of the group part of the account.

org The UUID of the organization part of the account.

sec_rgy_unix_sid_t
A structure identifying an account with UNIX ID numbers. It contains three fields:

person The UNIX ID of the person part of the account.

group The UNIX ID of the group part of the account.

org The UNIX ID of the organization part of the account.

sec_rgy_domain_t
This 32-bit integer specifies which naming domain a character string refers to: person, group, or organization.

sec_rgy_pgo_flags_t
A 32-bit bitset containing flags pertaining to registry entries. This type contains the following three flags:

sec_rgy_pgo_is_an_alias
If set, indicates the registry entry is an alias of another entry.

sec_rgy_pgo_is_required
If set, the registry item is required and cannot be deleted. An example of a required account is the one for the registry server itself.

sec_rgy_pgo_projlist_ok
If the accompanying item is a person entry, this flag indicates the person may have concurrent group sets. If the item is a group entry, the flag means this group can appear in a concurrent group set. The flag is undefined for organization items.

sec_rgy_pgo_item_t
The structure identifying a registry item. It contains five components:

id The UUID of the registry item, in **uuid_t** form.

unix_num A 32-bit integer containing the UNIX ID number of the registry item.

quota A 32-bit integer representing the maximum number of user-defined groups the account owner can create.

flags A **sec_rgy_pgo_flags_t** bitset containing information about the entry.

fullname A **sec_rgy_pname_t** character string containing a full name for the registry entry. For a person entry, this field might contain the real name of the account owner. For a group, it might contain a description of the group. This is

just a data field, and registry queries cannot search on the **fullname** entry.

sec_rgy_acct_admin_flags_t

A 32-bit bitset containing administration flags used as part of the administrator's information for any registry account. The set contains three flags:

sec_rgy_acct_admin_valid
Specifies that the account is valid for login.

sec_rgy_acct_admin_server
If set, the account's name can be used as a server name in a ticket-granting ticket.

sec_rgy_acct_admin_client
If set, the account's name can be used as a client name in a ticket-granting ticket.

Note that you can prevent the principal from being authenticated, by turning off both the **sec_rgy_acct_admin_server** and the **sec_rgy_acct_admin_client flags**.

sec_rgy_acct_auth_flags_t

A 32-bit bitset containing account authorization flags used to implement authentication policy as defined by the Kerberos Version 5 protocol. The set contains the following flags:

sec_rgy_acct_auth_user_to_user
Forces the use of user-to-user server authentication on a server principal.

sec_rgy_acct_auth_post_dated
Allows issuance of post-dated certificates.

sec_rgy_acct_auth_forwardable
Allows issuance of forwardable certificates.

sec_rgy_acct_auth_tgt
Allows issuance of certificates based on ticket-granting ticket (TGT) authentication. If this flag is not set, a client requesting a service may have to supply a password directly to the server.

sec_rgy_acct_auth_renewable
Allows issuance of renewable certificates.

sec_rgy_acct_auth_proxiable
Allows issuance of proxiable certificates.

sec_rgy_acct_auth_dup_session_key
Allows issuance of duplicate session keys.

sec_rgy_acct_admin_t
The portion of a registry account item containing components relevant to administrators. This structure consists of the fields listed below. Note that only *expiration_date*, *good_since_date*, *flags*, and *authentication_flags* can be modified by an administrator; the remaining fields are set by the security server.

creator This field, in **foreign_id_t format**, identifies the administrator who created the registry account.

creation_date
Specifies the creation date of the account, in **sec_timeval_sec_t** format.

last_changer
Identifies the last person to change any of the account information, in **foreign_id_t** format.

change_date
Specifies the date of the last modification of the account information, in **sec_timeval_sec_t** format.

expiration_date
The date after which the account will no longer be valid. In **sec_timeval_sec_t** format.

good_since_date
The Kerberos Version 5 TGT revocation date. TGTs issued before this date will not be honored. In **sec_timeval_sec_t** format.

flags Administrative flags in **sec_rgy_acct_admin_flags_t** format.

authentication_flags
Authentication flags in **sec_rgy_acct_auth_flags_t** format.

sec_rgy_acct_user_flags_t
A 32-bit bitset containing flags controlling user-modifiable information. There is only one flag currently implemented. If **sec_rgy_acct_user_passwd_valid** is set, it indicates the user password is valid. If it is not set, this flag prompts the user to change the password on the next login attempt.

sec_rgy_acct_user_t
A structure containing registry account information. The structure consists of the fields listed below. Note that only the **gecos**, **homedir**, **shell**, and **flags** fields can be modified by the account owner or other authorized user; the remaining fields are set by the security server.

gecos This is a character string (in **sec_rgy_pname_t** format) containing information about the account user. It generally consists of everything after the full name in the UNIX **gecos** format.

homedir The login directory for the account user, in **sec_rgy_pname_t** format.

shell The default shell for the account user, in **sec_rgy_pname_t** format.

passwd_version_number
An unsigned 32-bit integer, indicating the password version number. This value is used as output only.

passwd The UNIX encrypted account password, in **sec_rgy_unix_passwd_buf_t** format. This value is used as output only.

passwd_dtm
The date the password was established, in **sec_timeval_sec_t** format.

flags Account user flags, in **sec_rgy_acct_user_flags_t** format.

sec_rgy_plcy_pwd_flags_t
A 32-bit bitset containing two flags about password policy:

sec_rgy_plcy_pwd_no_spaces
If set, will not allow spaces in a password.

sec_rgy_plcy_pwd_non_alpha
If set, requires at least one nonalphanumeric character in the password.

sec_rgy_plcy_t
A structure defining aspects of registry account policy. It contains five components:

passwd_min_len
A 32-bit integer describing the minimum number of characters in the account password.

passwd_lifetime
The number of seconds after a password's creation until it expires, in **sec_timeval_period_t** format.

passwd_exp_date
The expiration date of the account password, in **sec_timeval_sec_t** format.

acct_lifespan
The number of seconds after the creation of an account before it expires, in **sec_timeval_period_t** format.

passwd_flags
Account password policy flags, in **sec_rgy_plcy_pwd_flags_t** format.

sec_rgy_plcy_auth_t
This type describes authentication policy. It is a structure containing two time periods, in **sec_timeval_period_t** format. One, **max_ticket_lifetime**, specifies the maximum length of the period during which a ticket-granting ticket (TGT) will be valid. The other, **max_renewable_lifetime**, specifies the maximum length of time for which such a ticket may be renewed. This authentication policy applies both to the registry as a whole as well as individual accounts. The effective policy for a given account is defined to be the more restrictive of the site and principal authentication policy.

sec_rgy_properties_t
A structure describing some registry properties. It contains the following:

read_version
A 32-bit integer describing the earliest version of the **secd** software that can read this registry.

write_version
A 32-bit integer describing the version of the **secd** software that wrote this registry.

minimum_ticket_lifetime
The minimum lifetime of an authentication certificate, in **sec_timeval_period_t** format.

default_certificate_lifetime
The normal lifetime of an an authentication certificate (ticket-granting ticket in Kerberos parlance), in **sec_timeval_period_t** format. Processes may request authentication certificates with longer lifetimes up to, but not in excess of, the maximum allowable lifetime as determined by the effective policy for the account.

low_unix_id_person
The lowest UNIX number permissible for a person item in the registry.

low_unix_id_group
The lowest UNIX number permissible for a group item in the registry.

low_unix_id_org
The lowest UNIX number permissible for an organization item in the registry.

max_unix_id
The largest UNIX number permissible for any registry entry.

flags Property flags, in **sec_rgy_properties_flags_t** format.

realm The name of the cell, in **sec_rgy_name_t** form, for which this registry is the authentication service.

realm_uuid The UUID of the same cell.

sec_rgy_properties_flags_t
A 32-bit bitset, containing flags concerning registry properties:

sec_rgy_prop_readonly
If set (TRUE), indicates that this registry is a query site.

sec_rgy_prop_auth_cert_unbound
If set (TRUE), the registry server will accept requests from any site.

sec_rgy_prop_shadow_passwd
If the shadow password flag is set (TRUE), the registry server will not include the account password when responding to a request for the user data from a specified account. This helps minimize the risk of an account password being intercepted while traveling over the network.

sec_rgy_prop_embedded_unix_id
Indicates that all UUIDs in this registry contain a UNIX number embedded. This implies that the UNIX numbers of objects in the registry cannot be changed, since UUIDs are immutable.

sec_rgy_override_t
A 32-bit integer used as a flag for registry override mode. Possible values are the constants **sec_rgy_no_override** and **sec_rgy_override**. When this mode is enabled, override data supplied by the node administrator will replace some of the data gotten from the registry for a given person/account under certain conditions. These conditions are as follows:

1. The registry permits the requested overrides to be set for this machine.
2. The override data is intended for person/account at hand.

When the mode is override off, data from the registry is returned to the end user or the application remains untouched.

sec_rgy_mode_resolve_t
A 32-bit integer used as a flag for resolve mode. Possible values are the constants **sec_rgy_no_resolve_pname** and **sec_rgy_resolve_pname**. When the mode is enabled, pathnames containing leading // (slashes) will be translated into a form understandable by the local machine's NFS.

sec_rgy_unix_passwd_buf_t
A character array of UNIX password strings.

Constants

The following constants are used in **sec_rgy_** calls:

sec_rgy_default_handle
The value of an unbound registry server handle.

sec_rgy_acct_key_t
The following 32-bit integer constants are used with the **sec_rgy_acct_key_t** data type:

sec_rgy_acct_key_none
Invalid key.

sec_rgy_acct_key_person
The person name alone is enough.

sec_rgy_acct_key_group
The person and group names are both necessary for the account abbreviation.

sec_rgy_acct_key_org
The person, group, and organization names are all necessary.

sec_rgy_acct_key_last
Key values must be less than this constant.

sec_rgy_pname_t_size
The maximum number of characters in a **sec_rgy_pname_t**.

sec_rgy_name_t_size
The maximum number of characters in a **sec_rgy_name_t**.

sec_rgy_domain_t
The following 32-bit integer constants are the possible values of the **sec_rgy_domain_t** data type:

sec_rgy_domain_person
The name in question refers to a person.

sec_rgy_domain_group
The name in question refers to a group.

sec_rgy_domain_org
The name in question refers to an organization.

sec_rgy_pgo_flags_t
A 32-bit constant equal to a variable of type **sec_rgy_pgo_flags_t** with no flags set.

sec_rgy_quota_unlimited
A 32-bit integer. Set the *quota* field of the **sec_rgy_pgo_item_t** type to this constant to override the registry quota limitation.

sec_rgy_acct_admin_flags_t
A 32-bit integer. This is the value of the **sec_rgy_acct_admin_flags_t** bitset when none of its flags are set.

sec_rgy_acct_auth_flags_none
A 32-bit integer. This is the value of the **sec_rgy_acct_auth_flags_t** bitset when none of its flags are set.

sec_rgy_acct_user_flags_t
A 16-bit integer. This is the value of the **sec_rgy_acct_user_flags_t** bitset when none of its flags are set.

sec_rgy_plcy_pwd_flags_t
A 16-bit integer. This is the value of the **sec_rgy_policy_pwd_flags_t** bitset when none of its flags are set.

sec_rgy_properties_flags_t
A 16-bit integer. This is the value of the **sec_rgy_properties_flags_t** bitset when none of its flags are set.

sec_rgy_override
A 32-bit integer, which turns registry override mode on. When this mode is enabled, override data supplied by the node administrator will replace some of the data gotten from the registry for a given person/account under certain conditions.

sec_rgy_no_override
A 32-bit integer, which turns off registry override mode.

sec_rgy_resolve_pname
A 32-bit integer, which turns on registry resolve mode. When the mode is enabled, pathnames containing leading // (slashes) will be translated into a form understandable by the local machine's NFS.

sec_rgy_no_resolve_pname
A 32-bit integer, which turns off registry resolve mode.

Files

/usr/include/dce/rgybase.idl
The **idl** file from which **rgybase.h** was derived.

Extended Registry Attribute Data Types

Synopsis

#include <dce/sec_attr_base.h>

Data Types

The following data types are used in **sec_rgy_attr** calls:

sec_attr_twr_ref_t
A pointer to a tower. This data type is used with the **sec_attr_twr_set_t** data type to allow a client to pass an unallocated array of towers, which the server must allocate. Both data types are used in conjunction with the **sec_attr_bind_type_t** data type.

sec_attr_twr_set_t
A structure that defines an array of towers. This data type is used with the **sec_attr_twr_ref_t** data type to allow a client to pass an unallocated array of towers, which the server must allocate. Both data types are used in conjunction with the **sec_attr_bind_type_t** data type. The **sec_attr_twr_set_t** structure consists of the following elements:

count
An unsigned 32-bit integer specifying the number of towers in the array.

towers[ ]
An array of pointers (of type **sec_attr_twr_ref_t**) to towers.

sec_attr_bind_type_t
A 32-bit integer that specifies the type of binding used by an attribute interface. The data type (which is used in conjunction with the **sec_attr_binding_t** data type) uses the following constants:

sec_attr_bind_type_string
An RPC string binding.

sec_attr_bind_type_twrs
A DCE protocol tower representation of a bindings.

sec_attr_bind_type_svrname
A name in **rpc_c_ns_syntax** format that identifies a CDS entry containing the server's binding information. This constant has the following structure:

name_syntax
Must be **rpc_c_ns_syntax_dce** to specify that DCE naming rules are used to specify **name**.

name
A pointer to a name of a CDS entry in **rpc_c_ns_syntax_dce** syntax.

sec_attr_binding_t
A discriminated union that supplies information to generate a binding handle for a attribute trigger. This data type, which is used in conjunction with the **sec_attr_bind_info_t** data type, is composed of the following elements:

bind_type A value of type **sec_attr_bind_type_t** that defines the type of binding used by an attribute interface. The contents of **tagged union** (see table) depend on the value of **sec_attr_bind_type_t**.

tagged_union
A tagged union specifying the binding handle. The contents of the tagged union depend on the value of **bind_type** as follows:

If bind_type is...	Then tagged_union is...
sec_attr_bind_type_string	A pointer to an unsigned 32-bit character string specifying an attribute's RPC string binding.
sec_attr_bind type_twrs	An attribute's tower binding representation of type **sec_attr_twr_set_t**.
sec_attr_bind_svrname	A pointer to a name of type **sec_attr_bind_type_t** that specifies a Cell Directory Service entry containing a attribute trigger's binding information.

sec_attr_binding_p_t
A pointer to a **sec_attr_binding_t** union.

sec_attr_bind_auth_info_type_t
An enumeration that defines whether or not the binding is authenticated. This data type is used in conjunction with the **sec_attr_bind_auth_info_t** data type to set up the authorization method and parameters for an RPC binding. The **sec_attr_bind_auth_info_type_t** type consists of the following elements:

sec_attr_bind_auth_none
The binding is not authenticated.

sec_attr_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_attr_bind_auth_info_t
A discriminated union that defines authorization and authentication parameters for a binding. This data type is used in conjunction with the **sec_attr_bind_auth_info_type_t** data type to set up the authorization method and parameters for an RPC binding. The

sec_attr_bind_auth_info_t data type consists of the following elements:

info_type A **sec_attr_bind_auth_info_type_t** data type that specifies whether or not the binding is authenticated. The contents of **tagged union** (below) depend on the value of **sec_attr_bind_auth_info_type_t**.

tagged_union

A tagged union specifying the method of authorization and the authorization parameters. For unauthenticated bindings (**sec_attr_bind_auth_info_type_t** = **sec_attr_bind_auth_none**), no parameters are supplied. For authenticated bindings (**sec_attr_bind_auth_info_type_t** = **sec_attr_bind_auth_dce**), the following union is supplied:

svr_princ_name

A pointer to a character string that specifies the principal name of the server referenced by the binding handle.

protect_level

An unsigned 32-bit integer indicating the protection level for RPC calls made using the server binding handle. The protection level determines the degree to which authenticated communications between the client and the server are protected by the authentication service specified by **authn_svc**.

If the RPC runtime or the RPC protocol in the bound protocol sequence does not support a specified level, the level is automatically upgraded to the next higher supported level. The possible protection levels are as follows:

Protection Level	Description
rpc_c_protect_level_default	Uses the default protection level for the specified authentication service. The default protection level for DCE shared-secret key authentication is **rpc_c_protect_level_pkt_value**
rpc_c_protect_level_none	Performs no authentication: tickets are not exchanged, session keys are not established, client PACs or names are not certified, and transmissions are in the clear. Note that although uncertified PACs should not be trusted, they may be useful for debugging, tracing, and measurement purposes.
rpc_c_protect_level_connect	Authenticates only when the client establishes a relationship with the server.
rpc_c_protect_level_call	Authenticates only at the beginning of each remote procedure call when the server receives the request. This level does not apply to remote procedure calls made over a connection-based protocol sequence (that is, **ncacn_ip_tcp**). If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.
rpc_c_protect_level_pkt	Ensures that all data received is from the expected client.

Protection Level	Description
rpc_c_protect_level_pkt_integ	Ensures and verifies that none of the data transferred between client and server has been modified. This is the highest protection level that is guaranteed to be present in the RPC runtime.
rpc_c_protect_level_pkt_privacy	Authenticates as specified by all of the previous levels and also encrypts each RPC argument value. This is the highest protection level, but is not guaranteed to be present in the RPC runtime.

authn_svc Specifies the authentication service to use. The exact level of protection provided by the authentication service is specified by *protect_level*. The supported authentication services are as follows:

Authentication Service	Description
rpc_c_authn_none	No authentication: no tickets are exchanged, no session keys established, client PACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for remote procedure calls made using this binding.
rpc_c_authn_dce_secret	DCE shared-secret key authentication.
rpc_c_authn_default	Default authentication service. The current default authentication service is DCE shared-secret key; therefore, specifying **rpc_c_authn_default** is equivalent to specifying **rpc_c_authn_dce_secret**.
rpc_c_authn_dce_public	DCE public key authentication (reserved for future use).

authz_svc
Specifies the authorization service implemented by the server for the interface.

The validity and trustworthiness of authorization data, like any application data, is dependent on the authentication service and protection level specified. The supported authorization services are as follows:

Authentication Service	Description
rpc_c_authz_none	Server performs no authorization. This is valid only if **authn_svc** is set to **rpc_c_authn_none**, specifying that no authentication is being performed.
rpc_c_authz_name	Server performs authorization based on the client principal name. This value cannot be used if **authn_svc** is **rpc_c_authn_none**.
rpc_c_authz_dce	Server performs authorization using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with this binding. Generally, access is checked against DCE ACLs.

sec_attr_bind_info_t

A structure that specifies attribute trigger binding information. This data type, which is used in conjunction with the **sec_attr_schema_entry_t** data type, contains of the following elements:

auth_info The binding authorization information of type **sec_attr_bind_auth_info_t**.

num_bindings

An unsigned 32-bit integer specifying the number of binding handles in **bindings**.

bindings An array of **sec_attr_binding_t** data types that specify binding handles.

sec_attr_bind_info_p_t

A pointer to a **sec_attr_bind_info_t** union.

sec_attr_encoding_t

An enumerator that contains attribute encoding tags used to define the legal encodings for attribute values. The data type, which is used in conjunction with the **sec_attr_value_t** and **sec_attr_schema_entry_t** data types, consists of the following elements:

sec_attr_enc_any
The attribute value can be of any legal encoding type. This encoding tag is legal only in a schema entry. An attribute entry must contain a concrete encoding type.

sec_attr_enc_void
The attribute has no value. It is simple a marker that is either present or absent.

sec_attr_enc_printstring
The attribute value is a printable IDL string in DCE portable character set.

sec_attr_enc_printstring_array
The attribute value is an array of printstrings.

sec_attr_enc_integer
The attribute value is a signed 32-bit integer.

sec_attr_enc_bytes
The attribute value is a string of bytes. The string is assumed to be a pickle or some other self describing type. (See also the **sec_attr_enc_bytes_t** data type.)

sec_attr_enc_confidential_bytes
The attribute value is a string of bytes that have been encrypted in the key of the principal object to which the attribute is attached. The string is assumed to be a pickle or some other self describing type. This encoding type is useful only when attached to a principal object, where it is decrypted and encrypted each time the principal's password changes. (See also the **sec_attr_enc_bytes_t** data type.)

sec_attr_enc_i18n_data
The attribute value is an internationalized string of bytes with a tag identifying the OSF registered codeset used to

encode the data. (See also the **sec_attr_i18n_data_t** data type.)

sec_attr_enc_uuid
The attribute is a value of type **uuid_t**, a DCE UUID.

sec_attr_enc_attr_set
The attribute value is an attribute set, a vector of attribute UUIDs used to associate multiple related attribute instances which are members of the set. (See also the **sec_attr_enc_attr_set_t** data type.)

sec_attr_enc_binding
The attribute value is a **sec_attr_bind_info_t** data type that specifies DCE server binding information.

sec_attr_enc_trig_binding
This encoding type is returned by **rs_attr_lookup** call. It informs the client agent of the trigger binding information of an attribute with a query trigger.

Unless **sec_attr_enc_void** or **sec_attr_enc_any** is specified, the attribute values must conform to the attribute's encoding type.

sec_attr_enc_bytes_t
A structure that defines the length of attribute encoding values for attributes encoded as **sec_attr_enc_bytes** and **sec_attr_enc_confidential_bytes**. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

length An unsigned 32-bit integer that defines the data length.

data[ ] An array of bytes specifying the length of attribute encoding data.

sec_attr_i18n_data_t
A structure that defines the codeset used for attributes encoded as **sec_attr_enc_il8n_data** and the length of the attribute encoding values. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

codeset An unsigned 32-bit identifier of a codeset registered with the Open Software Foundation.

length An unsigned 32-bit integer that defines the data length.

data[ ] An array of bytes specifying the length of attribute encoding data.

sec_attr_enc_attr_set_t
A structure that that supplies the UUIDs of each member of an attribute set. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

num_members
An unsigned 32-bit integer specifying the total number of attribute's in the set.

members[ ]
An array containing values of type **uuid_t**, the UUID of each member in the set.

sec_attr_enc_printstring_t
A structure that contains a printstring.

sec_attr_enc_printstring_p_t
A pointer to a **sec_attr_enc_printstring_t** structure.

sec_attr_enc_str_array_t
A structure that defines a printstring array. It consists of

num_strings
An unsigned 32-bit integer specifying the number of strings in the array.

strings[ ] An array of pointers (of type **sec_attr_enc_print_string_p_t**) to printstrings.

sec_attr_value_t
A discriminated union that defines attribute values. The union, which is used in conjunction with the **sec_attr_t** data type, consists of the following elements:

attr_encoding
A **sec_attr_encoding_t** data type that defines attribute encoding. The contents of **tagged union** depend on the value of **sec_attr_encoding_t**.

tagged_union
A tagged union whose contents depend on **attr_encoding** as follows:

If attr_encoding is...	Then tagged_union is...
sec_attr_enc_void	NULL
sec_attr_enc_printstring	A pointer to **printstring**
sec_attr_enc_printstring_array	A pointer to an array of **printstring**s
sec_attr_enc_integer	**signed_int**, a 32-bit signed integer
sec_attr_enc_bytes	**bytes**, a pointer to a structure of type **sec_attr_enc_bytes_t**
sec_attr_enc_confidential_bytes	**bytes**, a pointer to a structure of type **sec_attr_enc_bytes_t**
sec_attr_enc_i18n_data	**idata**, a pointer to a structure of type **sec_attr_i18n_data_t**
sec_attr_end_uuid	**uuid**, a value of type **uuid_t**
sec_attr_enc_attr_set	**attr_set**, a pointer to a structure of type **sec_attr_enc_attr_set_t**
sec_attr_enc_binding	**binding**, a pointer to a structure of type **sec_attr_binding_info_t**

sec_attr_t

A structure that defines an attribute. The structure consists of

attr_id A value of type **uuid_t**, the UUID of the attribute.

attr_value A value of type **sec_attr_value_t**.

sec_attr_acl_mgr_info_t

A structure that contains the access control information defined in a schema entry for an attribute. The structure, which is used in conjunction with the **sec_attr_schema_entry_t** data type, consists of the following elements:

acl_mgr_type

The value of type **uuid_t** that specifies the UUID of the ACL manager type that supports the object type to which the attribute can be attached. This field provides a well-defined context for evaluating the permission bits needed to operate on the attribute. The following table lists the ACL manager types for registry objects.

Registry Object Type	ACL Manager Type	Valid Permissions
principal	06ab9320-0191-11ca-a9e8-08001e039d7d	rcDnfmaug
group	06ab9640-0191-11ca-a9e8-08001e039d7d	rctDnfmM
organization	06ab9960-0191-11ca-a9e8-08001e039d7d	rctDnfmM
directory	06ab9c80-0191-11ca-a9e8-08001e039d7d	rcidDn
policy	06ab8f10-0191-11ca-a9e8-08001e039d7d	rcma
replist	2ac24970-60c3-11cb-b261-08001e039d7d	cidmAI

query_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to access the attribute's value.

update_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to update the attribute's value.

test_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to test the attribute's value.

delete_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to delete an attribute instance.

sec_attr_acl_mgr_info_p_t
A pointer to a **sec_attr_acl_mgr_info_t** structure.

sec_attr_acl_mgr_info_set_t
A structure that defines an attribute's ACL manager set. The structure consists of the following elements:

num_acl_mgrs
An unsigned 32-bit integer that specifies the number of ACL managers in the ACL manager set.

mgr_info[ ] An array of pointers of type **sec_attr_mgr_info_p_t** that define the ACL manager types in the ACL manager set and the permission sets associated with the ACL manager type.

sec_attr_intercell_action_t

An enumerator that specifies the action that should be taken by the privilege service when it reads acceptable attributes from a foreign cell. A foreign attribute is acceptable only if there is either a schema entry for the foreign cell or if **sec_attr_intercell_act_accept** is set to **true**.

This enumerator, which is used in conjunction with the **sec_attr_schema_entry_t** data type, is composed of the following elements:

sec_attr_intercell_act_accept

If the **unique** flag in the **sec_attr_schema_entry_t** data type is not set on, retain the attribute. If the **unique** flag is set on, retain the attribute only if its value is unique among all attribute instances of the same attribute type within the cell.

sec_attr_intercell_act_reject

Discard the input attribute.

sec_attr_intercell_act_evaluate

Use the binding information in the *trig_binding* field of this **sec_attr_schema_entry_t** data type to make a **sec_attr_trig_query** call to a trigger server. That server determines whether to retain the attribute value, discard the attribute value, or map the attribute to another value(s).

sec_attr_trig_type_t

Specifies the trigger type, a flag that determines whether an attribute trigger should be invoked for query operations. The data type, which is used in conjunction with the **sec_attr_schema_entry_t** data type, uses the following constants:

sec_attr_trig_type_query

The attribute trigger server is invoked for query operations.

sec_attr_trig_type_query

The attribute trigger server is invoked for update operations.

sec_attr_schema_entry_t

A structure that defines a complete attribute entry for the schema catalog. The entry is identified by both a unique string name and a unique attribute UUID. Although either can either can be used as a retrieval

key, the string name should be used for interactive access to the attribute and the UUID for programmatic access. The attribute UUID is used to identify the semantics defined for the attribute type in the schema.

The **sec_attr_schema_entry_t** data type consists of the following elements:

attr_name A pointer to the attribute name.

attr_id A value of type **uuid_t** that identifies the attribute type.

attr_encoding
An enumerator of type **sec_attr_encoding_t** that specifies the attribute's encoding.

acl_mgr_set A structure of type **sec_attr_acl_mgr_info_set_t** that specifies the ACL manager types that support the objects on which attributes of this type can be created and the permission bits supported by that ACL manager type.

schema_entry_flags
An unsigned integer of type **sec_attr_sch_entry_flags_t** that defines bitsets for the following flags:

unique When set on, this flag indicates that each instance of this attribute type must have a unique value within the cell for the object type implied by the ACL manager type. If this flag is not set on, uniqueness checks are not performed for attribute writes.

multi_valued
When set on, this flag indicates that this attribute type may be multivalued; in other words, multiple instances of the same attribute type can be attached to a single registry object. If this flag is not set on, only one instance of this attribute type can be attached to an object.

reserved When set on, this flag prevents the schema entry from being deleted through any interface or by any user. If this flag is not

set on, the entry can be deleted by any authorized principal.

use_defaults
When set on, the system-defined default attribute value will be returned on a client query if an instance of this attribute does not exist on the queried object. If this flag is not set on, system defaults are not used.

intercell_action
An enumerator of type **sec_attr_intercell_action_t** that specifies how the privilege service will handle attributes from a foreign cell.

trig_types A flag of type **sec_attr_trig_type_t** that specifies whether whether a trigger can perform update or query operations.

trig_binding
A pointer to a structure of type **sec_attr_bind_info_t** that supplies the attribute trigger binding handle.

scope A pointer to a string that defines the objects to which the attribute can be attached.

comment A pointer to a string that contains general comments about the attribute.

sec_attr_schema_entry_parts_t
A 32-bit bitset containing flags that specify the schema entry fields that can be modified on a schema entry update operation. This data type contains the following flags:

sec_attr_schema_part_name
If set, indicates that the attribute name (**attr_name**) can be changed.

sec_attr_schema_part_reserved
If set, indicates that the setting of the flag that determines whether or not the schema entry can be deleted (**reserved**) can be changed.

sec_attr_schema_part_defaults
If set, indicates that the flag that determines whether or not a query for a nonexistent attribute will not result

in a search for a system default (**apply_default**) can be changed.

sec_attr_schema_part_trig_bind
If set, indicates that the trigger's binding information (**trig_binding**) can be changed.

sec_attr_schema_part_comment
If set, indicates whether or not comments associated with the schema entry (**comment**) can be changed.

sec_attr_component_name_t
A pointer to a character string used to further specify the object to which the attribute is attached. (Note that this data type is analogous to the **sec_acl_component_name_t** data type in the ACL interface.)

sec_attr_cursor_t
A structure that provides a pointer into a registry database and is used for multiple database operations.

This cursor must minimally represent the object indicated by **xattrschema** in the schema interfaces, or *component_name* in the attribute interfaces. The cursor may additionally represent an entry within that schema or an attribute instance on that component.

sec_attr_srch_cursor_t
A structure that provides a pointer into a registry database and is used for multiple database operations. The cursor must minimally represent the list of all objects managed by this server that possess the search attributes specified in the **sec_attr_srch_cursor_init** routine. It may additionally represent a given object within this list as well as attribute instance(s) possessed by that object.

sec_attr_trig_cursor_t
A structure that provides an attribute trigger cursor for interactive operations. The structure consists of the following elements:

source A value of type **uuid_t** that provides a UUID to identify the server that initialized the cursor.

object_handle
A signed 32-bit integer that identifies the object (specified by **xattrschema** in the schema interface or

component_name in the attribute interface) upon which the operation is being performed.

entry_handle
A signed 32-bit integer that identifies the current entry (*schema_entry* in the schema interface or *attribute instance* in the attribute interface) for the operation.

valid A Boolean field with the following values:

true (1) Indicates an initialized cursor.

false (0) Indicates an uninitialized cursor.

sec_attr_trig_timeval_sec_t
A 32-bit integer containing the seconds portion of a UNIX **timeval_t**, to be used when expressing absolute dates.

Files

/usr/include/dce/sec_attr_base.idl
The **idl** file from which **sec_attr_base.h** was derived.

Constants

The following constants are used in **sec_attr** calls:

sec_attr_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_attr_bind_auth_none
The binding is not authenticated.

sec_attr_bind_type_string
The attribute uses an RPC string binding.

sec_attr_bind_type_svrname
The attribute uses a name in **rpc_c_ns_syntax** format that identifies a CDS entry containing the server's binding information. This constant has the following structure:

name_syntax
Must be **rpc_c_ns_syntax_dce** to specify that DCE naming rules are used to specify **name**.

name A pointer to a name of a CDS entry in **rpc_c_ns_syntax_dce** syntax.

sec_attr_bind_type_twr
The attribute uses a DCE protocol tower binding representation.

sec_attr_trig_type_t
The following 32-bit constants are used with the **sec_attr_trig_type_t** data type:

sec_attr_trig_type_query The trigger server can perform only query operations.

sec_attr_trig_type_update The trigger server can perform only update operations.

sec_attr_intercell_action_t
The following constants are used with the **sec_attr_intercell_action_t** data type:

sec_attr_intercell_act_accept
If the **unique** flag in the **sec_attr_schema_entry_t** data type is not set on, retain attributes from a foreign cell. If the **unique** flag is set on, retain the foreign attribute only if its value is unique among all attribute instances of the same attribute type within the cell.

sec_attr_intercell_act_reject
Discard attributes from a foreign cell.

sec_attr_intercell_act_evaluate
A trigger server determines whether to retain foreign attributes, discard foreign attributes, or map foreign attribute to another value(s).

sec_attr_schema_entry_parts_t
The following constants are used with the **sec_attr_schema_entry_parts_t** data type:

sec_attr_schema_part_name
Indicates that the attribute name can be changed in an schema update operation.

sec_attr_schema_part_reserved
Indicates that the setting of the **reserved** flag can be changed in a schema entry update.

sec_attr_schema_part_defaults
Indicates that the **apply_default** flag can be changed in a schema entry update operation.

sec_attr_schema_part_trig_bind
Indicates that trigger binding information can be changed in a schema entry update operation.

sec_attr_schema_part_comment
Indicates that comments associated with the schema entry can be changed in a schema entry update.

Login API Data Types

Synopsis

#include <dce/sec_login.h>

Data Types

The following data types are used in **sec_login_** calls:

sec_login_handle_t
This is an opaque pointer to a data structure representing a complete login context. The context includes a principal's network credentials, as well as other account information. The network credentials are also referred to as the principal's ticket-granting ticket.

sec_login_flags_t

A 32-bit set of flags describing restrictions on the use of a principal's validated network credentials. Currently, only one flag is implemented. Possible values are:

sec_login_no_flags
No special flags are set.

sec_login_credentials_private
Restricts the validated network credentials to the current process. If this flag is not set, it is permissible to share credentials with descendents of current process.

sec_login_auth_src_t

An enumerated set describing how the login context was authorized. The possible values are:

sec_login_auth_src_network
Authentication accomplished through the normal network authority. A login context authenticated this way will have all the network credentials it ought to have.

sec_login_auth_src_local
Authentication accomplished via local data. Authentication occurs locally if a principal's account is tailored for the local machine, or if the network authority is unavailable. Since login contexts authenticated locally have no network credentials, they may not be used for network operations.

sec_login_auth_src_overridden
Authentication accomplished via the override facility.

sec_login_passwd_t

The **sec_login_get_pwent()** call will return a pointer to a password structure, which depends on the underlying registry structure.

In most cases, the structure will look like that supported by Berkeley 4.4BSD and OSF/1, which looks like this:

```
struct passwd {
 char  *pw_name;          * user name *
 char  *pw_passwd;        * encrypted password *
```

```
int    pw_uid;          * user uid *
int    pw_gid;          * user gid *
time_t pw_change;         * password change time *
char  *pw_class;         * user access class *
char  *pw_gecos;          * Honeywell login info *
char  *pw_dir;           * home directory *
char  *pw_shell;         * default shell *
time_t pw_expire;         * account expiration *
};
```

sec_passwd_rec_t

A structure containing either a plaintext password or a preencrypted buffer of password data. The **sec_passwd_rec_t** structure consists of three components:

version_number
The version number of the password.

pepper A character string combined with the password before an encryption key is derived from the password.

key A structure consists of the following components:

key_type The key type can be the following:

sec_passwd_plain
Indicates that a printable string of data is stored in **plain**.

sec_passwd_des
Indicates that an array of data is stored in **des_key.**

tagged_union
A structure specifying the password. The value of the structure depends on **key_type**. If **key_type** is **sec_passwd_plain**, structure contains **plain**, a character string. If **key_type** is **sec_passwd_des**, the structure contains **des_key**, a DES key of type **sec_passwd_des_key_t**.

Constants

The following constants are used in **sec_login_** calls:

sec_login_default_handle
The value of a login context handle before setup or validation.

sec_login_flags_t
The following two constants are used with the **sec_login_flags_t** type:

sec_login_no_flags
No special flags are set.

sec_login_credentials_private
Restricts the validated network credentials to the current process. If this flag is not set, it is permissible to share credentials with descendents of current process.

sec_login_remote_uid
Used in the **sec_login_passwd_t** structure for users from remote cells.

sec_login_remote_gid
Used in the **sec_login_passwd_t** structure for users from remote cells.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **sec_login.h** was derived.

Extended Privilege Attribute API Data Types

Synopsis

```
#include <dce/id_epac.h>
#include <dce/nbase.h>
```

Data Types

The following data types are used in extended privilege attribute calls and in the **sec_login_cred** calls that implement extended privilege attributes.

sec_cred_cursor_t
A structure that provides an input/output cursor used to iterate through a set of delegates in the **sec_cred_get_delegate()** or **sec_login_cred_get_delegate()** calls. This cursor is initialized by the **sec_cred_initialize_cursor()** or **sec_login_cred_init_cursor()** call.

sec_cred_attr_cursor_t
A structure that provides an input/output cursor used to iterate through a set of extended attributes in the **sec_cred_get_extended_attributes()** call. This cursor is initialized by the **sec_cred_initialize_attr_cursor()** call.

sec_id_opt_req_t
A structure that specifies application-defined optional restrictions. The **sec_id_opt_req_t** data type is composed of the following elements:

restriction_len
An unsigned 16-bit integer that defines the size of the restriction data.

restrictions
A pointer to a **byte_t** that contains the restriction data.

sec_rstr_entry_type_t
An enumerator that specifies the entry types for delegate and target restrictions. This data type is used in conjunction with the **sec_id_restriction_t** data type where the specific UUID(s), if appropriate, are supplied. It consists of the following components:

sec_rstr_e_type_user
The target is a local principal identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_group
The target is a local group identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_foreign_user
The target is a foreign principal identified by principal and cell UUID.

sec_rstr_e_type_foreign_group
The target is a foreign group identified by group and cell UUID.

sec_rstr_e_type_foreign_other
The target is any principal that can authenticate to the foreign cell identified by UUID.

sec_rstr_e_type_any_other
The target is any principal that can authenticate to any cell, but is not identified in any other type entry.

sec_rstr_e_type_no_other
No pincipal can act as a target or delegate.

sec_id_restriction_t
A discriminated union that defines delegate and target restrictions. The union, which is used in conjunction with the **sec_restriction_set_t** data type, consists of the following elements:

entry_type A **sec_rstr_entry_type_t** that defines the ACL entry types for delegate and target restrictions. The value of **tagged_union** depends on the value of **entry_type**.

tagged_union
A tagged union whose contents depend on **entry_type** as follows:

If entry_type is...	**Then tagged_union is...**
sec_rstr_e_type_any_other	NULL
sec_rstr_e_type_foreign_other	**foreign_id** that identifies the foreign cell.
sec_rstr_e_type_user **Sec_rstr_e_type_group**	**id**, a **sec_id_t** that identifies the user or group.
sec_rstr_e_type_foreign_user **sec_rstr_e_type_foreign_group**	**foreign_id**, a **sec_id_foreign_t** that identifies the foreign user or group.

sec_id_restriction_set_t
A structure that that supplies delegate and target restrictions. The structure consists of

num_restrictions
A 16-bit unsigned integer that defines the number of restrictions in **restrictions**.

restrictions A pointer to a **sec_id_restriction_t** that contains the restrictions.

sec_id_compatibility_mode_t
A unsigned 16 bit integer that defines the compatibility between current and pre-1.1 servers. The data type uses the following constants:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The 1.0 PAC data extracted from the EPAC of the chain initiator.

sec_id_compat_mode_caller
Compatibility mode is on. The 1.0 PAC data extracted from the last delegate in the delegation chain.

sec_id_delegation_type_t
An unsigned 16 bit integer that defines the delegation type. The data type uses the following constants:

sec_id_deleg_type_none
Delegation is not allowed.

sec_id_deleg_type_traced
Traced delegation is allowed.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation is allowed.

sec_id_pa_t An structure that contains pre-1.1 PAC data extracted from an EPAC of a current version server. This data type, which is used for compatibility with pre-1.1 servers, consists of the following elements:

realm A value of type **sec_id_t** that contains the UUID that identifies the cell in which the principal associated with the PAC exists.

principal A value of type **sec_id_t** that contains the UUID of the principal.

group A value of type **sec_id_t** that contains the UUID of the principal's primary group.

num_groups
An unsigned 16-bit integer that specifies the number of groups in the principal's groupset.

groups An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

num_foreign_groupsets
An unsigned 16-bit integer that specifies the number of foreign groups for the principal's groupset.

foreign_groupsets
An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

sec_id_pac_t
An structure that contains a pre-1.1 PAC. This data type, which is used as output of the **sec_cred_get_v1_pac** call, consists of the following elements:

pac_type
A value of type **sec_id_pac_format_t** that can be used to describe the PAC format.

authenticated
A boolean field that indicates whether or not the PAC is authenticated (obtained from an authenticated source). FALSE indicates that the PAC is not authenticated. No authentication protocol was used in the rpc that transmitted the identity of the caller. TRUE indicates that the PAC is authenticated.

realm A value of type **sec_id_t** that contains the UUID that identifies the cell in which the principal associated with the PAC exists.

principal A value of type **sec_id_t** that contains the UUID of the principal.

group For local principals, a value of type **sec_id_t** that contains the UUID of the principal's primary group.

num_groups
An unsigned 16-bit integer that specifies the number of groups in the principal's groupset.

groups
An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

num_foreign_groups
An unsigned 16-bit integer that specifies the number of foreign groups in the principal's groupset.

foreign_groups
An array of pointers to **sec_id_t**s that contain the UUIDs of the each foreign group in the principal's groupset.

sec_id_pac_format_t
An enumerator that can be used to describe the PAC format.

sec_id_t A structure that contains UUIDs for principals, groups, or organizations and an optional printstring name. Since a UUID is an handle for the object's identity, the **sec_id_t** data type is the basic unit for identifying principals, groups, and organizations.

Because the printstring name is dynamically allocated, this datatype requires a destructor function. Generally, however, the **sec_id_t** is embedded in other data types (ACLs, for example), and these datatypes have a destructor function to release the printstring storage.

The **sec_id_t** data type is composed of the following elements:

uuid A value of type **uuid_t**, the UUID of the principal, group, or organization.

name A pointer to a character string containing the name of the principal, group, or organization.

sec_id_foreign_t
A structure that contains UUIDs for principals, groups, or organizations for objects in a foreign cell and the UUID that identifies the foreign cell. The **sec_id_foreign_t** data type is composed of the following elements:

id A value of type **sec_id_t** that contains the UUIDs of the objects from the foreign cell.

realm A value of type **sec_id_t** that contains the UUID of the foreign cell.

sec_id_foreign_groupset_t

A structure that contains UUIDs for set of groups in a foreign cell and the UUID that identifies the foreign cell. The **sec_id_foreign_groupset_t** data type is composed of the following elements:

realm

A value of type **sec_id_t** that contain the UUID of the foreign cell.

num_groups

An unsigned 16-bit integer specifying the number of group UUIDs in **groups**.

groups

A printer to a **sec_id_t** that contains the UUIDs of the groupset from the foreign cell.

Constants

The following constants are used in the extended privilege attribute calls and in the the **sec_login** calls that implement extended privilege attributes:

sec_id_compat_mode_none

Compatibility mode is off.

sec_id_compat_mode_initiator

Compatibility mode is on. The 1.0 PAC data extracted from the EPAC of the chain initiator.

sec_id_compat_mode_caller

Compatibility mode is on. The 1.0 PAC data extracted from the last delegate in the delegation chain.

sec_id_deleg_type_none

Delegation is not allowed.

sec_id_deleg_type_traced

Traced delegation is allowed.

sec_id_deleg_type_impersonation

Simple (impersonation) delegation is allowed.

sec_rstr_e_type_user
The delegation target is a local principal identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_group
The delegation target is a local group identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_foreign_user
The delegation target is a foreign principal identified by principal and cell UUID.

sec_rstr_e_type_foreign_group
The delegation target is a foreign group identified by group and cell UUID.

sec_rstr_e_type_foreign_other
The delegation target is any principal that can authenticate to the foreign cell identified by UUID.

sec_rstr_e_type_any_other
The delegation target is any principal that can authenticate to any cell, but is not identified in any other type entry.

sec_rstr_e_type_no_other
No pincipal can act as a target or delegate.

Files

/usr/include/dce/sec_cred.idl
The **idl** file from which **sec_cred.h** was derived.

/usr/include/dce/sec_epac.idl
The **idl** file from which **sec_epac.h** was derived.

/usr/include/dce/sec_nbase.idl
The **idl** file from which **sec_nbase.h** was derived.

ACL API Data Types

Synopsis

#include <dce/aclbase.h>

Data Types

The following data types are used in **sec_acl_** calls:

sec_acl_handle_t
A pointer to an opaque handle bound to an ACL that is the subject of a test or examination. The handle is bound to the ACL with **sec_acl_bind()**. An unbound handle has the value **sec_acl_default_handle**.

sec_acl_posix_semantics_t
A flag that indicates which, if any, POSIX ACL semantics an ACL manager supports. The following constants are defined for use with the **sec_acl_posix_semantics_t** data type:

sec_acl_posix_no_semantics
The manager type does not support POSIX semantics.

sec_acl_posix_mask_obj
The manager type supports the **mask_obj** entry type and POSIX 1003.6 Draft 12 ACL mask entry semantics.

sec_acl_t This data type is the fundamental type for the ACL manager interfaces. The **sec_acl_t** type contains a complete access control list, made up of a list of entry fields (type **sec_acl_entry_t**). The default cell identifies the authentication authority for simple ACL entries (foreign entries identify their own foreign cells). The **sec_acl_manager_type** identifies the manager to interpret this ACL.

The **sec_acl_t** type is a structure containing the following fields:

default_realm
A structure of type **sec_acl_id_t**, this identifies the UUID and (optionally) the name of the default cell.

sec_acl_manager_type
Contains the UUID of the ACL manager type.

num_entries
An unsigned 32-bit integer containing the number of ACL entries in this ACL.

sec_acl_entries
An array containing **num_entries** pointers to different ACL entries, each of type **sec_acl_entry_t**.

sec_acl_p_t
This data type, simply a pointer to a **sec_acl_t**, is for use with the **sec_acl_list_t** data type.

sec_acl_list_t
This data type is a structure containing an unsigned 32-bit integer **num_acls** that describes the number of ACLs indicated by its companion array of pointers, **sec_acls**, of type **sec_acl_p_t**.

sec_acl_entry_t
The **sec_acl_entry_t** type is a structure made up of the following components:

perms A set of flags of type **sec_acl_permset_t** that describe the permissions granted for the principals identified by this ACL entry. Note that if a principal matches more than one ACL entry, the effective permissions will be the most restrictive combination of all the entries.

entry_info A structure containing two members:

entry_type A flag of type **sec_acl_entry_type_t**, indicating the type of ACL entry.

tagged_union
A tagged union whose contents depend on the type of the entry.

The types of entries indicated by **entry_type** can be the following:

sec_acl_e_type_user_obj
The entry contains permissions for the implied user object. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_group_obj
The entry contains permissions for the implied group object. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_other_obj
The entry contains permissions for principals not otherwise named through user or group entries. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_user
The entry contains a key that identifies a user. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_group
The entry contains a key that identifies a group. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_mask_obj
The entry contains the maximum permissions for all entries other than **mask_obj**, **unauthenticated**, **user_obj**, **other_obj**.

sec_acl_e_type_foreign_user
The entry contains a key that identifies a user and the foreign realm.

sec_acl_e_type_foreign_group
The entry contains a key that identifies a group and the foreign realm.

sec_acl_e_type_foreign_other
The entry contains a key that identifies a foreign realm. Any user that can authenticate to the foreign realm will be allowed access.

sec_acl_e_type_any_other
The entry contains permissions to be applied to any accessor who can authenticate to any realm, but is not identified in any other entry (except **sec_acl_e_type_unauthenticated**).

sec_acl_e_type_unauthenticated

The entry contains permissions to be applied when the accessor does not pass authentication procedures. A privilege attribute certificate will indicate that the caller's identity is not authenticated. The identity is used to match against the standard entries, but the access rights are masked by this mask. If this mask does not exist in an ACL, the ACL is assumed to grant no access and all unauthenticated access attempts will be denied.

Great care should be exercised when allowing unauthenticated access to an object. Almost by definition, unauthenticated access is very easy to spoof. The presence of this mask on an ACL essentially means that anyone can get at least as much access as allowed by the mask.

sec_acl_e_type_extended

The entry contains additional pickled data. This kind of entry cannot be interpreted, but can be used by an out-of-date client when copying an ACL from one manager to another (assuming that the two managers each understand the data).

The contents of the tagged union depend on the entry type.

For the following entry types, the union contains a UUID and an optional print string (called **entry_info.tagged_union.id** with type **sec_id_t**) for an identified local principal, or for an identified foreign realm.

- **sec_acl_e_type_user**
- **sec_acl_e_type_group**
- **sec_acl_type_foreign_other**

For the following entry types, the union contains two UUIDs and optional print strings (called **entry_info.tagged_union.foreign_id** with type **sec_id_foreign_t**) for an identified foreign principal and its realm.

- **sec_acl_e_type_foreign_user**
- **sec_acl_e_type_foreign_group**

For an extended entry (**sec_acl_e_type_extended**), the union contains **entry_info.tagged_union.extended_info**, a pointer to an information block of type **sec_acl_extend_info_t**.

sec_acl_permset_t

A 32-bit set of permission flags. The flags currently represent the conventional file system permissions (read, write, execute) and the extended DFS permissions (owner, insert, delete).

The unused flags represent permissions that can only be interpreted by the manager for the object. For example, **sec_acl_perm_unused_00000080** may mean to one ACL manager that withdrawals are allowed, and to another ACL manager that rebooting is allowed.

The following constants are defined for use with the **sec_acl_permset_t** data type:

sec_acl_perm_read
The ACL allows read access to the protected object.

sec_acl_perm_write
The ACL allows write access to the protected object.

sec_acl_perm_execute
The ACL allows execute access to the protected object.

sec_acl_perm_control
The ACL allows the ACL itself to be modified.

sec_acl_perm_insert
The ACL allows insert access to the protected object.

sec_acl_perm_delete
The ACL allows delete access to the protected object.

sec_acl_perm_test
The ACL allows access to the protected object only to the extent of being able to test for existence.

The bits from 0x00000080 to 0x80000000 are not used by the conventional ACL permission set. Constants of the form **sec_acl_perm_unused_00000080** have been defined so application programs can easily use these bits for extended ACLs.

sec_acl_extend_info_t

This is an extended information block, provided for future extensibility. Primarily, this allows an out-of-date client to read an ACL from a newer manager and apply it to another (up-to-date) manager. The data cannot be interpreted by the out-of-date client without access to the appropriate pickling routines (that presumably are unavailable to such a client).

In general, ACL managers should not accept ACLs that contain entries the manager does not understand. The manager clearly cannot perform the security service requested by an uninterpretable entry, and it is considered a security breach to lead a client to believe that the manager is performing a particular class of service if the manager cannot do so.

The data structure is made up of the following components:

extension_type
The UUID of the extension type.

format_label
The format of the label, in **ndr_format_t** form.

num_bytes An unsigned 32-bit integer indicating the number of bytes containing the pickled data.

pickled_data
The byte array containing the pickled data.

sec_acl_type_t

The **sec_acl_type_t** type differentiates among the various types of ACLs an object can possess. Most file system objects will only have one ACL controlling the access to that object, but objects that control the creation of other objects (sometimes referred to as *containers*) may have more. For example, a directory can have three different ACLs: the directory ACL, controlling access to the directory; the initial object (or default object) ACL, which serves as a mask when creating new objects in the directory; and the initial directory (or default directory) ACL, which serves as a mask when creating new directories (containers).

The **sec_acl_type_t** is an enumerated set containing one of the following values:

sec_acl_type_object
The ACL refers to the specified object.

sec_acl_type_default_object
The ACL is to be used when creating objects in the container.

sec_acl_type_default_container
The ACL is to be used when creating nested containers.

The following values are defined but not currently used. They are available for application programs that may create an application-specific ACL definition.

- **sec_acl_type_unspecified_3**
- **sec_acl_type_unspecified_4**
- **sec_acl_type_unspecified_5**
- **sec_acl_type_unspecified_6**
- **sec_acl_type_unspecified_7**

sec_acl_printstring_t
A **sec_acl_printstring_t** structure contains a printable representation for a permission in a **sec_acl_permset_t** permission set. This allows a generic ACL editing tool to be used for application-specific ACLs. The tool need not know the printable representation for each permission bit in a given permission set. The **sec_acl_get_printstring()** function will query an ACL manager for the print strings of the permissions it supports. The structure consists of three components:

printstring A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.

helpstring A character string of maximum length **sec_acl_printstring_help_len** containing some text that may be used to describe the specified permission.

permissions A **sec_acl_permset_t** permission set describing the permissions that will be represented with the specified print string.

sec_acl_component_name_t
This type is a pointer to a character string, to be used to specify the entity a given ACL is protecting.

Constants

The following constants are used in **sec_acl_** calls:

sec_acl_default_handle
The value of an unbound ACL manager handle.

sec_rgy_acct_key_t
The following 32-bit integer constants are used with the **sec_rgy_acct_key_t** data type:

sec_rgy_acct_key_none
Invalid key.

sec_rgy_acct_key_person
The person name alone is enough.

sec_rgy_acct_key_group
The person and group names are both necessary for the account abbreviation.

sec_rgy_acct_key_org
The person, group, and organization names are all necessary.

sec_rgy_acct_key_last
Key values must be less than this constant.

sec_rgy_pname_t_size
The maximum number of characters in a **sec_rgy_pname_t**.

sec_acl_permset_t
The following constants are defined for use with the **sec_acl_permset_t** data type:

sec_acl_perm_read
The ACL allows read access to the protected object.

sec_acl_perm_write
The ACL allows write access to the protected object.

sec_acl_perm_execute
The ACL allows execute access to the protected object.

sec_acl_perm_owner
The ACL allows owner-level access to the protected object.

sec_acl_perm_insert
The ACL allows insert access to the protected object.

sec_acl_perm_delete
The ACL allows delete access to the protected object.

sec_acl_perm_test
The ACL allows access to the protected object only to the extent of being able to test for existence.

sec_acl_perm_unused_00000080 –
sec_acl_perm_unused_0x80000000
The bits from 0x00000080 to 0x80000000 are not used by the conventional ACL permission set. Constants have been defined so application programs can easily use these bits for extended ACLs.

sec_acl_printstring_len
The maximum length of the printable representation of an ACL permission. (See **sec_acl_printstring_t**.)

sec_acl_printstring_help_len
The maximum length of a help message to be associated with a supported ACL permission. (See **sec_acl_printstring_t**.)

Files

/usr/include/dce/aclbase.idl
The **idl** file from which **aclbase.h** was derived.

Key Management API Data Types

Notes

Key management operations that take a keydata argument expect a pointer to a **sec_passwd_rec_t** structure, and those that take a keytype argument (**void ***) expect a pointer to a **sec_passwd_type_t**. Key management operations that yield a keydata argument as output set the pointer to an array of **sec_passwd_rec_t**. (The array is terminated by an element with a key type of **sec_passwd_none**.)

Operations that take a keydata argument expect a pointer to a **sec_passwd_rec_t** structure. Operations that yield a keydata argument as output set the pointer to an array of **sec_passwd_rec_t**. (The array is terminated by an element with key type **sec_passwd_none**.) Operations that take a keytype argument (**void ***) expect a pointer to a **sec_passwd_type_t**.

Synopsis

#include <dce/keymgmt.h>

Data Types

sec_passwd_type_t

An enumerated set describing the currently supported key types. The possible values are as follows:

sec_passwd_none
Indicates no key types are supported.

sec_passwd_plain
Indicates that the key is a printable string of data.

sec_passwd_des
Indicates that the key is DES encrypted data.

sec_passwd_privkey
Indicates that the key is a private or public key of a public key pair used in public key authentication.

sec_passwd_genprivkey
Indicates the modulus bit size of the private key to be generated for a public key pair used in public key authentication.

sec_passwd_rec_t

A structure containing any of the following: a plaintext password, a preencrypted buffer of password data, a public-key-pair generation request, or a public or private key. The **sec_passwd_rec_t** structure consists of three components:

version_number
The version number of the password.

pepper A character string combined with the password before an encryption key is derived from the password.

key A structure consists of the following components:

key_type The key type can be the following:

sec_passwd_plain
Indicates that a printable string of data is stored in **plain**.

sec_passwd_des
Indicates that an array of data is stored in **des_key.**

sec_passwd_privkey
Indicates that X.509 ASN.1 DER-encoded data is stored in **priv_key**.

sec_passwd_genprivkey
Indicates that unsigned 32-bit data is stored in **modulus_size**.

tagged_union
A structure specifying the password. The value of the structure depends on **key_type**.

If **key_type** is **sec_passwd_plain**, the structure contains **plain**, a character string.

If **key_type** is **sec_passwd_des**, the structure contains **des_key**, a DES key of type **sec_passwd_des_key_t**.

If **key_type** is **sec_passwd_privkey**, the structure contains **priv_key**, a public or private key of type **sec_pk_data_t**.

If **key_type** is **sec_passwd_genprivkey**, the structure contains **modulus_size**, unsigned 32-bit data.

sec_passwd_version_t
An unsigned 32-bit integer that defines the password version number. You can supply a version number or a 0 for no version number. If you supply the constant **sec_passwd_c_version_none**, the security service supplies a system-generated version number.

sec_key_mgmt_authn_service
A 32-bit unsigned integer whose purpose is to indicate the authentication service in use, since a server may have different keys for different levels of security. The possible values of this data type and their meanings are as follows:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_private
DCE private key authentication (an implementation of the Kerberos system).

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

Constants

There are no constants specially defined for use with the key management API.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **keymgmt.h** was derived.

ID Mapping API Data Types

Synopsis

#include <dce/secidmap.h>

Data Types

No special data types are defined for the ID mapping API.

Constants

No special constants are defined for the ID mapping API.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **secidmap.h** was derived.

Password Management API Data Types

Synopsis

#include <dce/sec_pwd_mgmt.h>

Data Types

The following data types are used in **sec_pwd_mgmt_** calls:

sec_passwd_mgmt_handle_t
A pointer to an opaque handle consisting of password management information about a principal. It is returned by **sec_pwd_mgmt_setup()**.

Constants

There are no constants specially defined for use with the password management API.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The **idl** file from which **sec_pwd_mgmt.h** was derived.

Public Key API Data Types

Synopsis

#include <dce/sec_pk.h>

Data Types

The following data types are used in **sec_pk** calls:

sec_pk_domain_t
A UUID of type **uuid_t** associated with the application domain in which a public or private key is used.

sec_pk_usage_flags_t
A 32k-bit set of key-usage flags that describe the use of a key or key pair. The flags are:

sec_pk_usage_digitalSignature

sec_pk_usage_nonRepudiation

sec_pk_usage_keyEncipherment

sec_pk_usage_keyAgreement

sec_pk_usage_keyCertSign

sec_pk_usage_offLineCRLSign

These flags are described in the X.509 (1993E) AM 1 standard.

sec_pk_data_t

A structure that points to an X.509 or X.511 ASN.1 DER-encoded value. The **sec_pk_data_t** data type acts as a base for the following data types, which are aliases for **sec_pk_data_t**:

sec_pk_gen_data_t

sec_pk_pubkey_t

sec_pk_pvtkey_t

sec_pk_signed_t

sec_pk_encrypted_t

sec_pk_algorithm_id_t

The alias data types indicate the specific information pointed to by **sec_pk_data_t**. Instead of using **sec_pk_data_t** directly, use the alias data types.

The **sec_pk_data_t** data type consists of the following elements:

len The size of **data**.

data A pointer to a character string.

sec_pk_gen_data_t

A structure that acts as an alias to a **sec_pk_data_t** that contains plain ASCII data.

sec_pk_pubkey_t

A structure that acts as an alias to a **sec_pk_data_t** that contains an X.509 ASN.1 DER-encoded value of type **SubjectPublicKeyInfo**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_pvtkey_t

A structure that contains an X.509 ASN.1 DER-encoded private key value. The key format depends on the public key infrastructure.

This data type assumes that the public key infrastructure provides functions for generating a private key in this format.

sec_pk_signed_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **SIGNED**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_encrypted_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **ENCRYPTED**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_algorithm_id_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **AlgorithmIdentifier**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

Constants

The following constants are used in **sec_pk** calls:

The following unsigned 32-bit constants, which are used with the **sec_pk_usage_flags_t** data type, correspond to **KeyUsage** types defined in DAM 1 (Dec 1995) to X.509 (1993):

sec_pk_usage_digitalSignature

sec_pk_usage_nonRepudiation

sec_pk_usage_keyEncipherment

sec_pk_usage_dataEncipherment

sec_pk_usage_keyAgreement

sec_pk_usage_keyCertSign

sec_pk_usage_offLineCRLSign

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **sec_pk.h** was derived.

audit_intro

Purpose

Introduction to the DCE audit API runtime

Description

This introduction gives general information about the DCE audit application programming interface (API) and an overview of the following parts of the DCE audit API runtime:

- Runtime services
- Environment variables
- Data types and structures
- Permissions required

Runtime Services

The following is an alphabetical list of the audit API routines. With each routine name is its description. The types of application program that will most likely call the routine are enclosed in parentheses.

dce_aud_close()
Closes an audit trail (client/server applications, audit trail analysis and examination tools).

dce_aud_commit()
Performs the audit action(s) (client/server applications).

dce_aud_discard()
Discards an audit record which releases the memory (client/server applications, audit trail analysis and examination tools).

dce_aud_free_ev_info()
Frees the memory allocated for an event information structure returned from calling the **dce_aud_get_ev_info()** function (audit trail analysis and examination tools).

dce_aud_free_header()
Frees the memory allocated to a designated audit record header structure (audit trail analysis and examination tools).

dce_aud_get_ev_info()
Gets the event-specific information of a specified audit record (audit trail analysis and examination tools).

dce_aud_get_header()
Gets the header of a specified audit record (audit trail analysis and examination tools).

dce_aud_length()
Gets the length of a specified audit record (client/server applications, audit trail analysis and examination tools).

dce_aud_next()
Reads the next audit record from a specified audit trail into a buffer (audit trail analysis and examination tools).

dce_aud_open()
Opens a specified audit trail for read or write (client/server applications, audit trail analysis and examination tools).

dce_aud_print()
Formats an audit record into a human-readable form (audit trail analysis and examination tools).

dce_aud_put_ev_info()
Adds event-specific information to a specified audit record buffer (client/server applications).

dce_aud_set_trail_size_limit()
Sets a limit to the audit trail size (client/server applications).

dce_aud_start()
Determines whether a specified event should be audited given the client's binding information and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_name()
Determines whether a specified event should be audited given the client/server name and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_pac()
Determines whether a specified event should be audited given the client's privilege attribute certificate (PAC) and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_server_binding()
Determines whether a specified event should be audited given the server's binding information and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_uuid()
Determines whether a specified event should be audited given the client/server UUID and the event outcome. If the event must be audited, or if the outcome of the event is not yet known, the memory for the audit record descriptor is allocated and the address of this structure is returned to the caller (client/server applications).

Audit Data Types

The following subsections list the data types and structures used by applications to perform auditing and to analyze audit trails.

Event-Specific Information
The audit APIs allow applications to include event-specific information in audit records. Event-specific information must be represented as information items using the following data type.

```
typedef struct {
        unsigned16 format;
        union {
            idl_small_int small_int;
            idl_short_int short_int;
            idl_long_int long_int;
            idl_hyper_int hyper_int;
            idl_usmall_int usmall_int;
            idl_ushort_int ushort_int;
            idl_ulong_int ulong_int;
            idl_uhyper_int uhyper_int;
            idl_short_float short_float;
            idl_long_float long_float;
            idl_boolean boolean;
            uuid_t uuid;
            utc_t utc;
            sec_acl_t * acl;
            idl_byte * byte_string;
            idl_char * char_string;
        } data;
} dce_aud_ev_info_t;
```

The *format* field of the above data structure defines formatting information that is used to determine the type of the data referenced by the *data* field. The following table shows possible values of the *format* field, their corresponding data types, and their sizes.

Event Data Format Specifiers—intro(3sec)		
Specifier	**Data Type**	**Size**
aud_c_evt_info_small_int	**idl_small_int**	1 byte
aud_c_evt_info_short_int	**idl_short_int**	2 bytes
aud_c_evt_info_long_int	**idl_long_int**	4 bytes
aud_c_evt_info_hyper_int	**idl_hyper_int**	8 bytes
aud_c_evt_info_usmall_int	**idl_usmall_int**	1 bytes
aud_c_evt_info_ushort_int	**idl_ushort_int**	2 bytes
aud_c_evt_info_ulong_int	**idl_ulong_int**	4 bytes

aud_c_evt_info_uhyper_int	**idl_uhyper_int**	8 bytes
aud_c_evt_info_short_float	**idl_short_float**	4 bytes
aud_c_evt_info_long_float	**idl_long_float**	8 bytes
aud_c_evt_info_boolean	**idl_boolean**	1 byte
aud_c_evt_info_uuid	**uuid_t**	16 bytes
aud_c_evt_info_utc	**utc_t**	16 bytes
aud_c_evt_info_acl	**sec_acl_t ***	variable size
aud_c_evt_info_byte_string	**idl_byte ***	variable size
aud_c_evt_info_char_string	**idl_char ***	variable size

Byte strings and character strings are terminated with a 0 (zero) byte. New data types can be added to this list if they are used frequently. Servers could use the pickling service of the IDL compiler to encode complex data types into byte strings that are to be included in an audit record.

Audit Record Header Data Structure

The following data structure is used to store header information obtained from an audit record. This structure is normally only used by audit trail analysis and examination tools. That is, it is hidden from client/server applications.

```
typedef struct {
        unsigned32      format;
        uuid_t          server;
        unsigned32      event;
        unsigned16      outcome;
        unsigned16      authz_st;
        uuid_t client;
        uuid_t cell;
        unsigned16      num_groups;
        utc_t time;
        char *addr;
        uuid_t *groups;
} dce_aud_hdr_t;
```

format Contains the version number of the tail format of the event used for the event-specific information. With this format version number, the audit analysis tools can accommodate changes in the formats of the event-specific information. For example, the event-specific information of an event may initially be defined to be a 32-bit integer, and later changed to a character string. Format version 0 (zero) is assigned to the initial format for each event.

server Contains the UUID of the server that generates the audit record.

event Contains the event number.

outcome Indicates whether the event failed or succeeded. If the event failed, the reason for the failure is given.

authz_st Indicates how the client is authorized: by a name or by a DCE privilege attribute certificate (PAC).

client Contains the UUID of the client.

cell Contains the UUID of the client's cell.

num_groups
Contains the number of local group privileges the client used for access.

groups Contains the UUIDs of the local group privileges that are used by the client for the access. By default, the group information is not be included in the header (num_groups is set to 0 in this case), to minimize the size of the audit records. If the group information is deemed as important, it can be included.

Information about foreign groups (global groups that do not belong to the same cell where the client is registered) is not included in this version of audit header but may be included in later versions when global groups are supported.

time Contains a timestamp of **utc_t** type that records the time when the server committed the audit record (that is, after providing the event information through audit API function calls). Recording this time, rather than

recording the time when the audit record is appended to an audit trail, will better maintain the sequence of events. The implementation of the audit subsystem may involve communication between the server and a remote audit daemon, incurring indefinite delays by network problems or intruders. The inaccuracy in the **utc_t** timestamp may be useful for correlating events. When searching for events in an audit trail that occur within a time interval, if the results of the comparisons between the time of an event and the interval's starting and ending times is **maybe** (because of inaccuracies), then the event should be returned.

addr Records the client's address (port address of the caller). Port addresses are not authenticated. A caller can provide a fraudulent port address to a DCE server. However, if this unauthenticated port address is deemed to be useful information, a DCE server can record this information using this field.

The identity of the server cell is not recorded in the header, because of the assumption that all audit records in an audit trail are for servers within a single cell, and implicitly, the server cell is the local cell.

Audit Record Descriptor

An opaque data type, **dce_aud_rec_t**, is used to represent an audit record descriptor. An audit record descriptor may be created, manipulated, or disposed of by the following functions: The functions **dce_aud_start()**, **dce_aud_start_with_pac()**, **dce_aud_start_with_name()**, **dce_aud_start_with_server_binding()**, and **dce_aud_next()** return a record descriptor. The function **dce_aud_put_ev_info()** adds event information to an audit record through a record descriptor. The functions **dce_aud_get_header()**, **dce_aud_get_ev_info()**, and **dce_aud_length()** get the event and record information through a record descriptor. The function **dce_aud_commit()** commits an audit record through its descriptor. The function **dce_aud_discard()** disposes of a record descriptor. The function **dce_aud_discard()** is necessary only after reading the record (that is, after invoking **dce_aud_next()**.

Audit Trail Descriptor

An opaque data type, **dce_aud_trail_t**, is used to represent an audit trail descriptor. The **dce_aud_open()** function opens an audit trail and returns a trail descriptor; **dce_aud_next()** obtains an audit record from this descriptor; and **dce_aud_commit()** commits an audit record from and to an opened audit trail through this descriptor. The **dce_aud_close()** function disposes of this descriptor.

Environment Variables

The audit API routines use the following environment variables:

DCEAUDITOFF

If this environment variable is defined at the time the application is started, auditing is turned off.

DCEAUDITFILTERON

If this environment variable is defined, filtering is enabled.

DCEAUDITTRAILSIZE

Sets the limit of the audit trail size. This variable overrides the limit set by the **dce_aud_set_trail_size_limit()** function.

Permissions Required

To use an audit daemon's audit record logging service, you need the log (**l**) permission to the audit daemon.

Related Information

Books: *DCE 1.2.2 Command Reference*, *DCE 1.2.2 Application Development Guide*.

pkc_intro

Purpose Introduction to trust list facilities API

Description

This reference page describes the data types used by the trust list facility.

Overview of the Facility

Retrieving keys using this API is a three step process.

The first step involves creating a **pkc** structure called a trust list, which reflects the caller's initial trust. A trust list is a list of {name, key} pairs or certificates that are trusted *a priori*.

An empty trust list is created through a call to the routine **pkc_init_trustlist(3sec)**, and entries are inserted into a trust list by a call to **pkc_append_to_trustlist(3sec)**.

Once the trust list is complete, the application should next call **pkc_init_trustbase(3sec)**. This routine takes the trust list and processes it to produce a stucture called a trust base, which reflects any transitive trust, independent of the name of any desired target.

Creation of the trust base (and the prerequisite trust list) is expected to be performed at application startup, although it can be done any time prior to key retrieval. All processing up to this point is independent of the name(s) of principals whose keys are to be retrieved, and the trust base may be used for multiple key retrieval operations.

Once a trust base has been obtained, it may be used for key retrieval. Keys are retrieved for a given target principal using the **pkc_retrieve_keys(3sec)** routine, which takes a trust base and a name and returns an array of keys.

Data Structures

The following data structures are used by the trust list facilities.

- The **trust_type_t** type consists of an enumeration of the different possible varieties of trust:
  - **UNTRUSTED**

No trust (e.g., unauthenticated).

— **DIRECT_TRUST**

Direct trust via third party (e.g., authenticated registry).

— **CERTIFIED_TRUST**

Trust certified by caller's trust base.

- The **certification_flags_t** structure describes the trust that can be placed in a returned key. It contains the following fields:

  — **trust_type**

  A **trust_type_t** value expressing the style of trust.

  — **missing_crls**

  A **char**; its value is TRUE (not 0) if one or more CRLs are missing.

  — **revoked**

  A **char** whose value is TRUE (not 0) if any certificate has been revoked (even if it was still valid at the retrieval time).

- The **cert_t** structure contains the following fields:

  — **version**

  An **int** whose value must be 0.

  — **cert**

  A pointer to an **unsigned char** representing the ASN.1 encoding of a certificate.

  — **size**

  A **size_t** which represents the size of the encoding.

- The **trusted_key_t** structure contains the following fields:

  — **version**

  An **int** whose value must be 0.

  — **ca**

A pointer to an **unsigned char** (**x500 char**) string which represents the name of the Certification Authority whose key this is. For example, **/.../foo_cell/ca** or **/.../C=US/O=dec/CN=foo_cell/ca**.

— **key**

A pointer to an **unsigned char** representing the Certification Authority's ASN.1 key.

— **size**

A **size_t** representing the size of the CA's ASN.1 key.

— **startDate**

An **utc_t** representing the time at which the key begins to be valid.

— **endDate**

An **utc_t** representing the time at which the key ceases to be valid.

- The **trustitem_t** structure holds either a key, or a certificate. It has the following fields:

  — **type**

  An **int** whose value specifies either that the structure holds a key (**IS_KEY**) or a certificate (**IS_CERT**).

  — Depending on the value of **type**, the structure additionally contains a **trusted_key_t** (if **IS_KEY**) or a **cert_t** (if **IS_CERT**).

- The **selection_t** structure is defined for future enhancements that will enable users to specify usages for the key being retrieved. However, its contents are currently ignored.

Related Information

Functions: **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

crypto_intro

Purpose Introduction to the signature algorithm API registration facility

Description

This reference page describes the data types used by the signature algorithm (or ‘‘cryptographic’’) module registration API.

Accessing and Using Cryptographic Modules

Cryptographic implementations (also known as ‘‘algorithms’’) are identified by OIDs (object identifiers).

Policy implementors are recommended to access cryptographic modules mainly through the following routines, which perform all locking necessary to make the calls thread safe, and also transparently handle any context information that a given cryptographic implementation may need.

- **pkc_crypto_get_registered_algorithms(3sec)**

  Call this routine to get an OID set describing the currently registered algorithm implementations.

- **pkc_crypto_sign(3sec)**

  Call this routine to get data signed.

- **pkc_crypto_verify_signature(3sec)**

  Call this routine to verify signed data.

- **pkc_crypto_generate_keypair(3sec)**

  Call this routine to generate a pair of public/private keys.

Information about a cryptographic module may be obtained by calling **pkc_crypto_lookup_algorithm(3sec)**.

Data can also be signed and verified by looking up the desired algorithm (with **pkc_crypto_lookup_algorithm(3sec)**) and then explicitly calling the module’s

(sign)() or **verify()** routine, although in this case the calling application must take care to avoid multi-threading problems, and is also responsible for opening the crypto module prior to use, and closing it afterwards.

Implementing Cryptographic Modules

Every cryptographic module must export a **pkc_signature_algorithm_t** object.

The **pkc_signature_algorithm_t** data type is used to register a new cryptographic module with the certification API. It fully describes a specific implemented cryptographic algorithm, and provides entry points to its **sign()** and **verify()** functions. It is defined as follows:

```
typedef struct {
 OM_uint32 version;
 gss_OID_desc alg_id>;
 pkc_alg_flags_t flags;
 char reserved[32 - sizeof(pkc_alg_flags_t)];
 char * (* name)(void);
 unsigned32 (*open) (void** context);
 unsigned32 (*close) (void** context);
 unsigned32 (*verify) (void ** context,
    sec_pk_gen_data_t * data,
    sec_pk_data_t * public_key,
    sec_pk_data_t * signature);
 unsigned32 (*sign) (void ** context,
    sec_pk_gen_data_t * data,
    sec_pk_data_t * private_key,
    sec_pk_data_t * signature);
 unsigned32 (*generate_keypair) (void ** context,
    unsigned32 size,
    void * alg_info,
    sec_pk_data_t * private_key,
    sec_pk_data_t * public_key);} pkc_signature_algorithm_t;
```

The **(name)()**, **(open)()**, **(close)()**, **(verify)()**, **(sign)()** and **(generate_keypair)()** routines must be implemented by the application implementing the algorithm and registered by calling the **pkc_crypto_register_signature_alg(3sec)** routine. Note, however, that all the routines except for **(verify)()** and **(name)()** are optional.

Explanations of all the fields in **pkc_signature_algorithm_t** are contained in the following subsections.

Cryptographic Module Data Fields

The structure contains the following data fields:

version Identifies the version of the certification API for which the module is implemented. The value of this field is always **pkc_V1** for DCE 1.2.

alg_id An object identifier that identifies the algorithm; the OID that appears in certificates signed by the algorithm.

flags Describes whether the module's **(sign)()** and **(verify)()** functions are threadsafe, and whether the module supports simultaneous crypto sessions.

The **version** and **alg_id** fields are required for all versions of this data structure. Other fields may be version dependent.

Cryptographic Module Functions

NULL may be supplied as the address of the **(open)()**, **(close)()**, **(sign)()**, or **(generate_keypair)()** routines, if the cryptographic module does not provide or require the corresponding feature; the presence of these functions in a cryptographic module is optional. However, all cryptographic modules must provide **(verify)()** and **(name)()** functions.

Algorithm Flags Data Type

The **pkc_alg_flags_t** data type is used to record various information about a cryptographic module. It is defined as follows:

```
typedef struct {
 char threadsafe;
 char multi_session;}
pkc_alg_flags_t;
```

The structure contains two fields which have the following meanings:

threadsafe Has a non-zero (TRUE) value if the module's **(sign)()** and **(verify)()** routines may be safely called simultaneously (within a single crypto session) by multiple threads.

multi_session
Has a non-zero (TRUE) value if the module implementation supports multiple simultaneous crypto sessions.

Cryptographic Module Data Fields

The structure contains the following data fields:

version Identifies the version of the certification API for which the module is implemented. The value of this field is always **pkc_V1** for DCE 1.2.

alg_id An object identifier that identifies the algorithm; the OID that appears in certificates signed by the algorithm.

flags Describes whether the module's **(sign)()** and **(verify)()** functions are threadsafe, and whether the module supports simultaneous crypto sessions.

The **version** and **alg_id** fields are required for all versions of this data structure. Other fields may be version dependent.

Cryptographic Module Functions

NULL may be supplied as the address of the **(open)()**, **(close)()**, **(sign)()**, or **(generate_keypair)()** routines, if the cryptographic module does not provide or require the corresponding feature; the presence of these functions in a cryptographic module is optional. However, all cryptographic modules must provide **(verify)()** and **(name)()** functions.

Name

(name)() - Returns the algorithm name as a string for use in diagnostic or auditing messages

Synopsis

```
char * (* name)(void);
```

Description

The name should be returned in storage allocated using the **pkc_alloc()** function defined in **pkc_base.h**. Note that this is the only cryptographic module routine that may be called without first calling the **(open)()** routine.

This routine is mandatory.

Name

(open)() - Opens and initializes the cryptographic module

(close)() - Closes the cryptographic module

Both routines are optional.

Synopsis

unsigned32 (*open) (void *context*);

unsigned32 (*close) (void *context*);

Parameters

Output

context An opaque (to the caller) data structure containing any state information required by the module across calls.

Description

Before invoking any of the module's encryption routines (e.g., **(sign)()** or **(verify)()**, the certification API will invoke the module's **(open)()** function. Once the module's **(close)()** routine has been invoked, the certification facility will invoke **(open)()** again before making any further calls to the module.

Both the **(open)()** and the **(close)()** routines require only one argument, *context*. If the cryptographic module requires state information to be maintained between calls, it may use the *context* parameter to do this. The information is initialized by the **(open)()** routine and returned as an opaque object to the caller, who then passes the parameter to subsequent **(sign)()**, **(verify)()**, **(generate_keypair)()**, or **(close)()** calls.

Note that if the **(open)()** routine stores any state in the *context* parameter, the **(close)()** routine should free this storage.

Name

(sign)() - Calculates a signature over the supplied data using the specified key

Synopsis

**unsigned32 (*sign) (void ** *context*,
sec_pk_gen_data_t * *data*,
sec_pk_data_t * *private_key*,
**sec_pk_data_t ** *signature*);

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

data The certificate data that is to be signed.

private_key Key to use to generate the signature, provided as a BER-encoded **PrivateKeyInfo** object, as defined in PKCS#8, as appropriate for the algorithm.

Output

signature The signature generated on the data passed. Storage allocation should be performed by calling the **pkc_alloc()** and **pkc_free()** functions defined in **pkc_base.h**.

Description

The **(sign)()** routine calculates a signature over the supplied data, using the specified key. The *private_key* parameter will be a BER-encoded **PrivateKeyInfo** data object. The *signature* should be returned by the function; storage allocation should be performed by calling the **pkc_alloc()** and **pkc_free()** functions defined in **pkc_base.h**.

This routine is optional.

Name

(verify)() - Checks the supplied signature against the supplied data, thus verifying the certificate in which the data and the signature appear

Synopsis

unsigned32 (*verify) (void ** *context*,
sec_pk_gen_data_t * *data*,**
sec_pk_data_t * *public_key*,**
sec_pk_data_t * *signature*);**

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

data The entire **certificateInfo**.

public_key The public key to use on the signature.

signature The signature to be verified.

Description

The **(verify)()** routine checks the supplied signature against the supplied data. *public_key* is a **SubjectPublicKeyInfo** data structure, encoded in BER, as found within an X.509 certificate.

The routine should return 0 for a correct signature, **pkc_invalid_signature** for an incorrect signature, or another DCE-defined error status to indicate any other errors.

This routine must be implemented in any cryptographic module.

Name

(generate_keypair)() - Generates a pair of public and private keys

Synopsis

```
unsigned32 (*generate_keypair) (void ** context,
  unsigned32 size,
  void *alg_info,
  sec_pk_data_t * private_key,
  sec_pk_data_t * public_key);
```

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

size Specifies the key size.

alg_info Specifies the crypto module.

Output

private_key The generated private key.

public_key The generated public key.

Description

The **(generate_keypair)()** routine generates a pair of private and public keys. The *size* parameter should be used by the routine to determine the key size in some way (for the RSA algorithm, for example, it should be treated as the number of bits in the key modulus). The *private_key* and *public_key* parameters should return BER-encoded **PrivateKeyInfo** and **SubjectPublicKeyInfo** data objects respectively. The *alg_info* parameter can be used for algorithm-specific information to modify the key generation process. However, all crypto modules that offer this function should be prepared to operate when **NULL** is supplied for this parameter.

This routine is optional.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

policy_intro

Purpose Introduction to the policy module registration and service facility

Description

This reference page describes the data types used by the policy module registration and service API.

The routines documented here are intended for the use of policy implementors. Regular users invoke a policy via the high-level API (e.g., **pkc_retrieve_keyinfo(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, etc.) which calls the routines documented below internally.

Accessing Policy Switch Modules

Policy modules are identified by OIDs (object identifiers). A policy module is accessed by passing its identifying OID to **pkc_plcy_lookup_policy(3sec)**.

There are two ways of retrieving a key: either by looking up the desired policy module and then explicitly calling its **(retrieve_keyinfo)()** routine; or by simply calling the **pkc_plcy_retrieve_keyinfo(3sec)** routine, identifying the desired policy by means of an OID passed directly to the call. The latter method, in which the operation is performed in one step, is the recommended one.

Policy Flags Data Type

The **pkc_plcy_flags_t** data type is used to record various information about a policy module. It is defined as follows:

```
typedef struct {char threadsafe; char multi_session;} pkc_plcy_flags_t;
```

The structure contains two fields which have the following meanings:

threadsafe Has a non-zero (TRUE) value if the policy's **retrieve_keyinfo()** function may be safely called simultaneously (within a single policy session) by multiple threads.

multi_session

Has a non-zero (TRUE) value if the policy implementation supports multiple simultaneous policy sessions.

Policy Module Data Type

The **pkc_policy_t** data type is used to register a new policy module with the certification API. It fully describes a policy module's functionality, and provides entry points to its key retrieval functions. It is defined as follows:

```
typedef struct {
 OM_uint32 version;
 gss_OID_desc policy_id;
 pkc_plcy_flags_t flags;
 char reserved[32 - sizeof(pkc_plcy_flags_t)];
 char * (* name) (void);
 unsigned32 (*open) (void** context);
 unsigned32 (*close) (void** context);
 unsigned32 (*establish_trustbase) (void ** context,
    const pkc_trust_list_t & initial_trust,
    const utc_t * date,
    pkc_usage_t desired_usage,
    char initial_explicit_policy_required,
    pkc_trust_list_t & out_trust);


 unsigned32 (*retrieve_keyinfo) (void ** context,
    const pkc_trust_list_t &trust,
    const x500name &subjectName,
    const utc_t * date,
    const uuid_t & domain,
    pkc_key_usage_t desired_usage,
    char initial_explicit_policy_required,
    pkc_key_information_t &key);
 unsigned32 (*delete_trustbase) (void ** context,
    void ** trust_base_handle);
 unsigned32 (*delete_keyinfo) (void ** context,
    void ** keys_handle);
 unsigned32 (*get_key_count) (void ** context,
    void * keys_handle,
    size_t * key_count);
```

```
unsigned32 (*get_key_data) (void ** context,
    void * keys_handle,
    unsigned key_index,
    unsigned char ** key_data,
    size_t * key_length);
unsigned32 (*get_key_trust) (void ** context,
    void * keys_handle,
    unsigned key_index,
    certification_flags_t * flags     uuid_t * domain,
    pkc_generic_key_usage_t * usages);
unsigned32 (*get_key_certifier_count) (void ** context,
    void * keys_handle,
    unsigned key_index,
    size_t * ca_count);
unsigned32 (*get_key_certifier_info) (void ** context,
    void * keys_handle,
    unsigned key_index,
    unsigned ca_index,
    char ** ca_name,
    utc_t * certification_start,
    utc_t * certification_expiration,
    char  * is_crl_valid,
    utc_t * last_crl_seen,
    utc_t * next_crl_expected);
} pkc_policy_t;
```

The **(name)()**, **(open)()**, **(close)()**, **(establish_trustbase()**, **(*get_key_count)()**, **(*get_key_data)()**, **(*get_key_trust)()**, **(*get_key_certifier_count)()**, **(*get_key_certifier_info)()**, and **(*retrieve_keyinfo)()** routines must be implemented by the application implementing the module and registered using the **pkc_register_policy(3sec)** routine. Note, however, that only **(*retrieve_keyinfo)()**, **(*get_key_count)()**, **(*get_key_certifier_count)()** and **(*get_key_data)()** are required. Explanations of all the fields in **pkc_policy_t** are contained in the following subsections.

Policy Module Data Fields

The structure contains the following data fields:

version Identifies the version of the certification API for which the module is implemented. The value of this field is always **pkc_V1** for DCE 1.2.

policy_id An object identifier that identifies the policy.

flags Describes whether the module's key retrieval function is threadsafe, and whether the module supports simultaneous policy sessions.

The **version** and *alg_id* fields are required for all versions of this data structure. Other fields may be version dependent.

Policy Module Functions

NULL may be supplied as the address of the **(name)()**, **(open)()**, **(establish_trustbase)()** or **(close)()** routines, if the policy module does not provide or require the corresponding feature; the presence of these functions in a policy module is optional. However, all policy modules must provide a **(retrieve_keyinfo)()** function.

Name

(name)() — Returns the policy name as a string, suitable for use in diagnostic or auditing messages

This routine is optional.

Synopsis

```
char * (* name) (void);
```

Description

The name should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**. The caller of this routine is expected to invoke **pkc_free(3sec)**to release the storage once the name is no longer required.

Note that this is the only policy module routine that may be called without first calling the **(open)()** routine.

Name

(open)() — Opens and initializes the policy module

(close)() — Closes the policy module

Both these routines are optional.

Synopsis

unsigned32 (*open) (void *context*);**

unsigned32 (*close) (void *context*);**

Parameters

Output

context An opaque (to the caller) data structure containing any state information required by the module across calls.

Description

Before invoking any policy routines (e.g., **(retrieve_keyinfo)()**), the certification API will invoke the module's **(open)()** function. Once the module's **(close)()** routine has been invoked, the certification facility will invoke **(open)()** again before making any further calls to the module.

Both the **(open)()** and the **(close)()** routines require only one argument, *context*. If the policy module requires state information to be maintained between calls, it may use the *context* parameter to do this. The information is initialized by the **(open)()** routine and returned as an opaque object to the caller, who then passes the parameter to subsequent **(retrieve_keyinfo)()**, **(establish_trustbase)()**, or **(close)()** calls.

Note that if the **(open)()** routine stores any state in the *context* parameter, the **(close)()** routine should free this storage.

Name

(establish_trustbase)() — Initializes a trust base

Synopsis

```
unsigned32 (*establish_trustbase) (void ** context,
 const pkc_trust_list_t & initial_trust,
 const utc_t * date,
 char initial_explicit_policy_required,
 pkc_trust_list_t & out_trust);
```

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

initial_trust Specifies the caller's initial trust.

date Specifies time for which information is to be returned.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

out_trust An extended trust list.

Description

This is a one-time call made by an application to initialize a trust base. It returns the *out_trust* parameter, which contains an extended trust list. After this call is made, the application can call **(retrieve_keyinfo)()** to obtain the public keys of any particular principal. If the trust base does not change, **(retrieve_keyinfo)()** can be used to look up another principal's public key without incurring the cost of another call to **(establish_trustbase)()**. A trust base will not change unless the *initial_trust* list changes.

Name

(*delete_trustbase)() — Frees storage allocated for a trust base

This routine is optional.

Synopsis

unsigned32 (*delete_trustbase) (void ** *context*,
void ** *trust_base_handle*);

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

Name

(*delete_keyinfo)() — Frees storage allocated for key information

This routine is optional.

Synopsis

unsigned32 (*delete_keyinfo) (void ** *context*,
void ** *keys_handle*);

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

Description

(*delete_keyinfo)() frees storage that was allocated for key information.

Name

(*get_key_count)() — Returns number of keys

This routine is optional.

Synopsis

```
unsigned32 (*get_key_count) (void ** context,
 void * keys_handle,
 size_t * key_count);
```

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

Output

key_count Number of keys for the principal.

Description

(*get_key_count)() returns the number of keys for the principal. This value is determined by reference to the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

Name

(*get_key_data)() — Returns a public key

This routine is optional.

Synopsis

**unsigned32 (*get_key_data) (void ** *context*,
void * *keys_handle*,
unsigned *key_index*,
**unsigned char ** *key_data*,
size_t * *key_length*);

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index Index (ranging from 0 to *key_count* – 1) of the key desired.

Output

key_data The encoded public key.

key_length Length of the key data returned.

Description

(*get_key_data)() returns the public key specified by *index*. The *key_data* returned is extracted from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

key_data should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**.

Name

(*get_key_trust)() — Returns information about key trust

This routine is optional.

Synopsis

unsigned32 (*get_key_trust) (void ** *context*,
void * *keys_handle*,**
unsigned *key_index*,**
certification_flags_t * *flags* **uuid_t *** *domain*,**
pkc_generic_key_usage_t * *usages*);**

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index Index (ranging from 0 to *key_count* − 1) of the key desired.

Output

flags Information about the trust that can be placed in the key (see below).

domain Indicates domain of retrieved key. A value of **sec_pk_domain_unspecified** or **NULL** means that the policy does not distinguish keys by domain.

usages Indicates usage key is intended for.

Description

(*get_key_trust)() returns information about the trust reposed in the key specified by *index*. This information is determined by reference to the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

The returned **certification_flags_t** structure describes the trust that can be placed in the key. It contains the following fields:

- **trust_type**

  A **trust_type_t** value, which will be one of the following:

  — **UNTRUSTED**

  No trust (e.g., unauthenticated).

  — **DIRECT_TRUST**

  Direct trust via third party (e.g., authenticated registry).

  — **CERTIFIED_TRUST**

  Trust certified by caller's trust base.

- **missing_crls**

  A **char**; its value is TRUE (not 0) if one or more CRLs are missing.

- **revoked**

  A **char** whose value is TRUE (not 0) if any certificate has been revoked (even if it was still valid at the retrieval time).

If **domain** and **usages** are passed as non-**NULL** pointers, upon successful return these parameters will describe the domain and permitted usage(s) of the specified key. Policies that do not distinguish keys according to domain will indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage will indicate all usages are permitted.

The returned **usages** is a bit mask which describes the usage(s), if any, which the key is restricted to. The value is formed by AND-ing together one or more of the following constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A returned **usages** value of **NULL** (or a value with all bits set) means that the key is suitable for any usage.

Name

(*get_key_certifier_count)() — Returns number of key's certifying authorities

This routine is optional.

Synopsis

unsigned32 (*get_key_certifier_count) (void ** *context*,
void * *keys_handle*,
unsigned key_index,**
size_t * *ca_count*);

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index Index (ranging from 0 to *key_count* – 1) of the key desired.

Output

ca_count Number of certifying authorities for the key.

Description

(*get_key_certifier_count)() returns the number of certifying authorities for the key specified by *index*. This information is determined from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

Name

(*get_key_certifier_info)() — Returns information about a certifying authority

This routine is optional.

Synopsis

```
unsigned32 (*get_key_certifier_info) (void ** context,
 void * keys_handle,
 unsigned key_index,
 unsigned ca_index,
 char ** ca_name,
 utc_t * certification_start,
 utc_t * certification_expiration,
 char  * is_crl_valid,
 utc_t * last_crl_seen,
 utc_t * next_crl_expected);
```

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index Index (ranging from 0 to *key_count* − 1) of the key desired.

ca_index Index of the certifier about whom information is desired.

Output

ca_name The name of the certifier.

certification_start
Time at which certification by this certifier starts.

certification_expiration
Time at which certification by this certifier ends.

is_crl_valid If TRUE, there is a certificate revocation list for this certifier.

last_crl_seen
Time at which certificate revocation list was last seen.

next_crl_expected
Time at which next certificate revocation list is expected.

Description

(*get_key_certifier_info)() returns information about the certifying authority specified by *ca_index* for the key specified by *key_index*.

The desired information is extracted by the routine from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

Note that any of the return parameters may be passed as NULL if the corresponding information is not required.

The *certifier_name* parameter should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**.

Name

(retrieve_keyinfo)() — Returns the public key for the specified principal

Synopsis

```
unsigned32 (*retrieve_keyinfo) (void ** context,
 const void * trust_base_handle,
 const x500name & subjectName,
 const utc_t * date,
 const uuid_t & domain,
 pkc_key_usage_t desired_usage,
 char initial_explicit_policy_required,
 void ** keys_handle);
```

Parameters

Input

context An opaque (to the caller) data structure containing any state information required by the module across calls.

trust_base_handle
Specifies

subjectName Specifies the desired subject name.

date Specifies time for which information is to be returned.

domain Specifies the particular domain to which the key-search operation should be restricted. Specify **sec_pk_domain_unspecified** or **NULL** to indicate that keys for any domain should be retrieved.

desired_usage
Specifies the one or more specific usages to which the key-search operation should be restricted.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

keys_handle The handle to the public key for the specified target principal.

Description

The **(retrieve_keyinfo)()** routine reads the certificate for the specified principal name, verifies it, and (if the verification is successful) extracts the public key stored in it and returns it to the caller.

The returned key information handle can be interrogated by various **pkc_cert_** routines to extract the actual key and determine the degree of trust that can be placed in the returned key.

If **domain** and **desired_usage** are passed as non-**NULL** pointers, upon successful return these parameters will describe the domain and permitted usage(s) of the specified key. Policies that do not distinguish keys according to domain will indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage will indicate all usages are permitted.

The **desired_usage** parameter consists of a bit mask, formed by AND-ing together one or more of the constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A **NULL** can be specified for **desired_usage** to indicate that keys for any usage should be retrieved.

Note that some of the routine's parameters relate to X.509 version 3 certificates, support for which is not committed for DCE 1.2. The API has been designed with the intent that it be capable of supporting all currently defined versions of X.509, so that it need not change when version 3 support is added. For version 1 or version 2 policies and certificates, the *desired_usage* parameter will be ignored, and the *initial_explicit_policy_required* parameter must be zero (specifying that the policy need not explicitly appear in the first certificate).

Related Information

Functions: **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_trustlist_intro

Purpose

Introduction to the certificate manipulation facility

Description

This reference page describes the data types used by the certificate manipulation facility.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Trust Lists

The trust list is the fundamental object within the certificate manipulation facility. A trust list is a set of keys which are trusted, plus a list of revoked certificate serial numbers. Keys are inserted into a trust list either directly (via the **pkc_add_trusted_key(3sec)** function) or indirectly (via the **pkc_check_cert_against_trustlist(3sec)** function). The latter routine will only add keys if the certificate signature can be verified by a key already in the trust list, and if the certificate has not been revoked.

Currently, trust lists are relatively static objects: once a key is inserted, its trust properties do not change. If, for example, a key is added that is capable of extending the trust in another key within the list, the second key is not automatically updated.

Using the Certificate Manipulation Facility

The way that a policy module is expected to use the facility is as follows.

1. Create an initial trust list containing the directly trusted keys, that is, the start point(s) of all valid trust chains.

   Typically, this set of keys will be used for multiple certificate chain evaluations. If the policy wishes to impose additional path constraints over the constraints expressed within the certificates, it must maintain a master copy of the original trust list and clone it to create a modifiable version for each chain the policy module wants to verify. After verification of a candidate chain, the cloned trust list must be discarded so that the next trial verification starts from a known state.

2. Using the initial trust list as a starting point, the policy module retrieves a chain of certificates and adds them to the trust list one by one, starting with the certificate(s) closest to the start point(s).

   Multiple chains may be evaluated simultaneously using a single trust list for policies that do not wish to impose additional constraints on the trust chain; however the policy module must ensure that for each trust-chain, certificates are added in the correct order. A future auto-update enhancement may lift this requirement.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

gssapi_intro

Purpose Generic security service application programming interface

Description

This introduction includes general information about the generic security service application programming interface (GSSAPI) defined in Internet RFC 1508, *Generic Security Service Application Programming Interface*, and RFC 1509, *Generic Security Service API : C-bindings*. It also includes an overview of error handling, data types, and calling conventions, including the following:

- Integer types
- String and similar data
- Object identifiers (OIDs)
- Object identifier sets (OID sets)
- Credentials
- Contexts
- Authentication tokens
- Major status values
- Minor status values
- Names
- Channel bindings
- Optional parameters

General Information

The GSSAPI provides security services to applications using peer-to-peer communications (instead of DCE-secure RPC). Using DCE GSSAPI routines, applications can perform the following operations:

- Enabling an application to determine another application's user

- Enabling an application to delegate access rights to another application
- Applying security services, such as confidentiality and integrity, on a per-message basis

GSSAPI represents a secure connection between two communicating applications with a data structure called a *security context*. The application that establishes the secure connection is called the *context indicator* or simply *indicator*. The context initiator is like a DCE RPC client. The application that accepts the secure connection is the *context acceptor* or simply *acceptor*. The context acceptor is like a DCE RPC server.

There are four stages involved in using the GSSAPI, as follows:

1. The context initiator acquires a credential with which it can prove its identity to other processes. Similarly, the context acceptor cquires a credential to enable it to accept a security context. Either application may omit this credential acquistion and use their default credentials in subsequent stages. See the section on credentials for more information.

   The applications use credentials to establish their global identity. The global identity can be, but is not necessarily, related to the local user name under which the application is running. Credentials can contain either of the following:

   - Login context

     The login context includes a principal's network credentials, as well as other account information.

   - Principal name and a key

     The key corresponding to the principal name must be registered with the DCE security registration in a key table. A set of GSSAPI routines enables applications to register and use principal names.

2. The communicating applications establish a joint security context by exchanging authentication tokens.

   The security context is a pair of GSSAPI data structures that contain information that is shared between the communicating applications. The information describes the state of each application. This security context is required for per-message security services.

   To establish a security context, the context initiator calls the **gss_init_sec_context()** routine to get a *token*. The token is cryptographically protected, opaque data. The context initiator transfers the token to the context

acceptor, which in turn passes the token to the **gss_accept_sec_context()** routine to decode and extract the shared information.

As part of the establishing the the security context, the context initiator is authenticated to the context acceptor. The context initiator can require the context acceptor to authenticate itself in return.

The context initiator can *delegate* rights to allow the context acceptor to act as its agent. Delegation means the context initiator gives the context acceptor the ability to initiate additional security contexts as an agent of the context initiator. To delegate, the context initiator sets a flag on the **gss_init_sec_context()** routine indicating that it wants to delegate and sends the returned token in the normal way to the context acceptor. The acceptor passes this token to the **gss_accept_sec_context()** routine, which generates a delegated credential. The context acceptor can use the credential to initiate additional security contexts.

3. The applications exchange protected messages and data.

   The applications can call GSSAPI routines to protect data exchanged in messages. The application sends a protected message by calling the appropriate GSSAPI routine to do the following:

   - Apply protection
   - Bind the message to the appropriate security context

   The application can then send the resulting information to the peer application.

   The application that receives the message passes the received data to a GSSAPI routine, which removes the protection and validates the data.

   GSSAPI treats application data as arbitrary octet strings. The GSSAPI per-message security services can provide either of the following:

   - Integrity and authentication of data origin
   - Confidentiality, integrity, and authentication of data origin

4. When the applications have finished communicating, either one may instruct GSSAPI to delete the security context.

There are two sets of GSSAPI routines, as follows:

- Standard GSSAPI routines, which are defined in the Internet RFC 1508, *Generic Security Service Application Programming Interface*, and RFC 1509, *Generic Security Service API : C-bindings*. These routines have the prefix **gss_**.

- OSF DCE extensions to the GSSAPI routines. These are additional routines that enable an application to use DCE security services. These routines have the prefix **gssdce_**.

The following sections provide an overview of the GSSAPI error handling and data types.

Error Handling

Each GSSAPI routine returns two types of status values:

- Major status values, which are generic API routine errors or calling errors defined in RFC 1509.
- Minor status values, which indicate DCE-specific errors.

If a routine has output parameters that contain pointers for storage allocated by the routine, the output parameters will always contain a valid pointer even if the routine returns an error. If no storage was allocated, the routine sets the pointer to NULL and sets any length fields associated with the pointers (such as in the **gss_buffer_desc** structure) to 0 (zero).

Minor status values usually contain more detailed information about the error. They are not, however, portable between GSSAPI implementations. When designing portable applications, use major status values for handling errors. Use minor status values to debug applications and to display error and error-recovery information to users.

GSSAPI Data Types

This section provides an overview of the GSSAPI data types and their definitions.

Integer Types

The GSSAPI defines the following integer data type:

OM_uint32 32-bit unsigned integer

This integer data type is a portable data type that the GSSAPI routine definitions use for guaranteed minimum bit-counts.

String and Similar Data

Many of the GSSAPI routines take arguments and return values that describe contiguous multiple-byte data, such as opaque data and character strings. Use the

gss_buffer_t data type, which is a pointer to the buffer descriptor **gss_buffer_desc**, to pass the data between the GSSAPI routines and applications.

The **gss_buffer_t** data type has the following structure:

```
typedef struct gss_buffer_desc_struct {
 size_t  length;
 void    *value;
} gss_buffer_desc, *gss_buffer_t;
```

The *length* field contains the total number of bytes in the data and the *value* field contains a pointer to the actual data.

When using the **gss_buffer_t** data type, the GSSAPI routine allocates storage for any data it passes to the application. The calling application must allocate the **gss_buffer_desc** object. It can initialize unused **gss_buffer_desc** objects with the value **GSS_C_EMPTY_BUFFER**. To free the storage, the application calls the **gss_release_buffer()** routine.

Object Identifier

Applications use the **gss_OID** data type to choose a security mechanism, either DCE security or Kerberos, and to specify name types. Select a security mechanism by using the following two OIDs:

- To use DCE security, specify either **GSSDCE_C_OID_DCE_KRBV5_DES** or **GSS_C_NULL_OID**.
- To use Kerberos Version 5, specify **GSSDCE_C_OID_KRBV5_DES**.

Use of the default security mechanisms, specified by the constant **GSS_C_NULL_OID**, helps to ensure the portability of the application.

The **gss_OID** data type contains tree-structured values defined by ISO and has the following structure:

```
typedef struct gss_OID_desc_struct {
 OM_uint32 length;
 void      *elements;
}  gss_OID_desc, *gss_OID;
```

The *elements* field of the structure points to the first byte of an octet string containing the ASN.1 BER encoding of the value of the **gss_OID** data type. The *length* field contains the number of bytes in the value.

The **gss_OID_desc** values returned from the GSSAPI are read-only values. The application should not try to deallocate them.

Object Identifier Sets

The **gss_OID_set** data type represents one or more object identifiers. The values of the **gss_OID_set** data type are used to do the following:

- Report the available mechanisms supported by GSSAPI
- Request specific mechanisms
- Indicate which mechanisms a credential supports

The **gss_OID_set** data type has the following structure:

```
typedef struct gss_OID_set_desc_struct {
 int     count;
 gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

The *count* field contains the number of OIDs in the set. The *elements* field is a pointer to an array of **gss_oid_desc** objects, each describing a single OID. The application calls the **gss_release_oid_set()** routine to deallocate storage associated with the **gss_OID_set** values that the GSSAPI routines return to the application.

Credentials

Credentials establish, or prove, the identity of an application or other principal.

The **gss_cred_id_t** data type is an atomic data type that identifies a GSSAPI credential data structure.

Contexts

The security context is a pair of GSSAPI data structures that contain information shared between the communicating applications. The information describes the cryptographic state of each application. This security context is required for per-message security services and is created by a successful authentication exchange.

The **gss_ctx_id_t** data type contains an atomic value that identifies one end of a GSSAPI security context. The data type is opaque to the caller.

Authentication Tokens

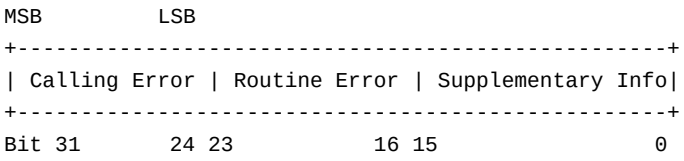
GSSAPI uses tokens to maintain the synchronization between the applications sharing a security context. The token is a cryptographically protected bit string generated by DCE security at one end of the GSSAPI security context for use by the peer application at the other end of the security context. The data type is opaque to the caller.

The applications use the **gss_buffer_t** data type as tokens to GSSAPI routines.

Major Status Values

GSSAPI routines return GSS status codes as their **OM_uint32** function value. These codes indicate either generic API routine errors or calling errors.

A GSS status code can indicate a single, fatal generic API error from the routine and a single calling error. Additional status information can also be contained in the GSS status code. The errors are encoded into a 32-bit GSS status code, as follows:



If a GSSAPI routine returns a GSS status code whose upper 16 bits contain a nonzero value, the call failed. If the calling error field is nonzero, the context initiator's use of the routine was in error. In addition, the routine can indicate additional information by setting bits in the supplementary information field of the status code. The tables that follow describe the routine errors, calling errors, and supplementary information status bits and their meanings.

The following table lists the GSSAPI routine errors and their meanings:

Name	Field Value	Meaning
GSS_S_BAD_MECH	1	The required mechanism is unsupported.
GSS_S_NAME	2	The name passed is invalid.

Name	Field Value	Meaning
GSS_S_NAMETYPE	3	The name passed is unsupported.
GSS_S_BAD_BINDINGS	4	The channel bindings are incorrect.
GSS_S_BAD_STATUS	5	A status value was invalid.
GSS_S_BAD_SIG	6	A token had an invalid signature.
GSS_S_NO_CRED	7	No credentials were supplied.
GSS_S_NO_CONTEXT	8	No context has been established.
GSS_S_DEFECTIVE_TOKEN	9	A token was invalid.
GSS_S_DEFECTIVE _CREDENTIAL	10	A credential was invalid.
GSS_S_CREDENTIALS _EXPIRED	11	The referenced credentials expired.
GSS_S_CONTEXT_EXPIRED	12	The context expired.
GSS_S_FAILURE	13	The routine failed. Check minor status codes.

The following table lists the calling error values and their meanings:

Name	Field Value	Meaning
Name	Value	Meaning
GSS_S_CALL_INACCESSIBLE _READ	1	Could not read a required input parameter.
GSS_S_CALL_INACCESSIBLE _WRITE	2	Could not write a required output parameter.
GSS_S_BAD_STRUCTURE	3	A parameter was incorrectly structured.

The following table lists the supplementary bits and their meanings.

Name	Bit Number	Meaning
GSS_S_CONTINUE_NEEDED	0 (LSB)	Call the routine again to complete its function.
GSS_S_DUPLICATE_TOKEN	1	The token was a duplicate of an earlier token.
GSS_S_OLD_TOKEN	2	The token's validity period expired; the routine cannot verify that the token is not a duplicate of an earlier token.
GSS_S_UNSEQ_TOKEN	3	A later token has been processed.

All **GSS_S_** symbols equate to complete **OM_uint32** status codes, rather than to bitfield values. For example, the actual value of **GSS_S_BAD_NAMETYPE** (value 3 in the routine error field) is 3 << 16.

The major status code **GSS_S_FAILURE** indicates that DCE security detected an error for which no major status code is available. Check the minor status code for details about the error. See the section on minor status values for more information.

The GSSAPI provides the following three macros:

- **GSS_CALLING_ERROR()**
- **GSS_ROUTINE_ERROR()**
- **GSS_SUPPLEMENTARY_INFO()**

Each macro takes a GSS status code and masks all but the relevant field. For example, when you use the **GSS_ROUTINE_ERROR()** macro on a status code, it returns a value. The value of the macro is arrived at by using only the routine errors field and zeroing the values of the calling error and the supplementary information fields.

An additional macro, **GSS_ERROR()**, lets you determine whether the status code indicated a calling or routine error. If the status code indicated a calling or routine error, the macro returns a nonzero value. If no calling or routine error is indicated, the routine returns a 0 (zero).

Note: At times, a GSSAPI routine that is unable to access data can generate a platform-specific signal, instead of returning a **GSS_S_CALL_INACCESSIBLE_READ** or **GSS_S_CALL_INACCESSIBLE_WRITE** status value.

Minor Status Values

The GSSAPI routines return a *minor_status* parameter to indicate errors from either DCE security or Kerberos. The parameter can contain a single error, indicated by an **OM_uint32** value. The **OM_uint32** data type is equivalent to the DCE data type **error_status_t** and can contain any DCE-defined error.

Names

Names identify principals. The GSSAPI authenticates the relationship between a name and the principal claiming the name.

Names are represented in the following two forms:

- A printable form, for presentation to an application
- An internal, canonical form that is used by the API and is opaque to applications

The **gss_import_name()** and **gss_display_name()** routines convert names between their printable form and their **gss_name_t** data type. GSSAPI supports only DCE principal names, which are identified by the constant OID, **GSSCDE_C_OID_DCENAME**.

The **gss_compare_names()** routine compares internal form names.

Channel Bindings

You can define and use channel bindings to associate the security context with the communications channel that carries the context. Channel bindings are communicated to the GSSAPI by using the following structure:

```
typedef struct gss_channel_binding_struct {
 OM_uint32       initiator_addrtype;
 gss_buffer_desc initiator_address;
 OM_uint32       acceptor_addrtype;
 gss_buffer_desc aceptor_address;
 gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

Use the *initiator_addrtype* and *acceptor_addrtype* fields to initiate the type of addresses contained in the *initiator_address* and *acceptor_address* buffers. The address types and their **addrtype** values are as follows:

Unspecified **GSS_C_AF_UNSPEC**

Host-local **GSS_C_AF_LOCAL**

DARPA Internet
GSS_C_AF_INET

ARPAnet IMP
GSS_C_AF_IMPLINK

pup protocols (for example, BSP)
GSS_C_AF_PUP

MIT CHAOS protocol
GSS_C_AF_CHAOS

XEROX NS **GSS_C_AF_NS**

nbs **GSS_C_AF_NBS**

ECMA **GSS_C_AF_ECMA**

datakit protocols
GSS_C_AF_DATAKIT

CCITT protocols (for example, X.25)
GSS_C_AF_CCITT

IBM SNA **GSS_C_AF_SNA**

Digital DECnet
GSS_C_AF_DECnet

Direct data link interface
GSS_C_AF_DLI

LAT **GSS_C_AF_LAT**

NSC Hyperchannel
GSS_C_AF_HYLINK

AppleTalk **GSS_C_AF_APPLETALK**

BISYNC 2780/3780
GSS_C_AF_BSC

Distributed system services
GSS_C_AF_DSS

OSI TP4 **GSS_C_AF_OSI**

X25 **GSS_C_AF_X25**

No address specified
GSS_C_AF_NULLADDR

The tags specify address families rather than addressing formats. For address families that contain several alternative address forms, the *initiator_address* and the *acceptor_address* fields should contain sufficient information to determine which address form is used. Format the bytes that contain the addresses in the order in which the bytes are transmitted across the network.

The GSSAPI creates an octet string by concatenating all the fields (*initiator_addrtype*, *initiator_address*, *acceptor_addrtype*, *acceptor_address*, and *application_data*). The security mechanism signs the octet string and binds the signature to the token generated by the **gss_init_sec_context()** routine. The context acceptor presents the same bindings to the **gss_accept_sec_context()** routine, which evaluates the signature and compares it to the signature in the token. If the signatures differ, the **gss_accept_sec_context()** routine returns a **GSS_S_BAD_BINDINGS** error, and the context is not established.

Some security mechanisms check that the *initiator_address* field of the channel bindings presented to the **gss_init_sec_context()** routine contains the correct network address of the host system. Therefore portable applications should use either the correct address type and value or the **GSS_C_AF_NULLADDR** for the *initiator_addrtype* address field. Some security mechanisms include the channel binding data in the token instead of a signature, so portable applications should not use confidential data as channel-binding components. The GSSAPI does not verify the address or include the plain text bindings information in the token.

Optional Parameters

In routine descriptions, *optional parameters* allow the application to request default behaviors by passing a default value for the parameter. The following conventions are used for optional parameters:

Convention	Value Default	Explanation
gss_buffer_t types	**GSS_C_NO_BUFFER**	For an input parameter, indicates no data is supplied. For an output parameter, indicates that the information returned is not required by the application.
Integer types (input)		Refer to the reference pages for default values.
Integer types (output)	NULL	Indicates that the application does not require the information.
Pointer types (output)	NULL	Indicates that the application does not require the information.
OIDs	**GSS_C_NULL_OID**	Indicates the default choice for name type or security mechanism.
OID sets	**GSS_C_NULL_OID_SET**	Indicates the default set of security mechanisms, DCE security and Kerberos.

Convention	Value Default	Explanation
Credentials	**GSS_C_NO_CREDENTIAL**	Indicates that the application should use the default credential handle.
Channel bindings	**GSS_C_NO_CHANNEL_BINDINGS**	Indicates that no channel bindings are used.

Related Information

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

dce_acl_copy_acl

Purpose

Copies an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_copy_acl(
        sec_acl_t *source,
        sec_acl_t *target,
        error_status_t *status);
```

Parameters

Input

source A pointer to the ACL to be copied.

target A pointer to the new ACL that is to receive the copy.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_copy_acl()** routine makes a copy of a specified ACL. The caller passes the space for the target ACL, but the space for the **sec_acl_entries** array is allocated. To free the allocated space, call **dce_acl_obj_free_entries()**, which frees the entries, but not the ACL itself.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_no_memory
The **rpc_sm_allocate()** routine could not obtain memory.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_free_entries(3sec)**.

dce_acl_inq_acl_from_header

Purpose

Retrieves the UUID of an ACL from an item's header in a backing store

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_acl_from_header(
        dce_db_header_t db_header,
        sec_acl_type_t sec_acl_type,
        uuid_t *acl_uuid,
        error_status_t *status);
```

Parameters

Input

db_header The backing store header containing the ACL object.

sec_acl_type The type of ACL to be identified:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

Output

acl_uuid A pointer to the UUID of the ACL object.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_acl_from_header()** routine gets the UUID for an ACL object of the specified type from the specified backing store header.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

db_s_key_not_found
The specified key was not found in the backing store. (This error is passed through from **dce_db_fetch()**.)

db_s_bad_index_type
The key's type is wrong, or else the backing store is not by name or by UUID. (This error is passed through from **dce_db_fetch()**.)

sec_acl_invalid_type
The *sec_acl_type* parameter does not contain a valid type.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_resolve_by_name(3sec)**, **dce_acl_resolve_by_uuid(3sec)**.

dce_acl_inq_client_creds

Purpose

Returns the client's credentials

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_client_creds(
        handle_t handle,
        sec_cred_pa_handle_t *creds,
        error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

Output

creds A pointer to the returned credentials, or NULL if unauthorized.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_client_creds()** routine returns the client's security credentials found through the RPC binding handle.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

rpc_s_authn_authz_mismatch
Either the client, or the server, or both is not using the **rpc_c_authz_dce** authorization service.

rpc_s_invalid_binding
Invalid RPC binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

rpc_s_binding_has_no_auth
Binding has no authentication information. The client or the server should have called **rpc_binding_set_auth_info()**.

Related Information

Functions: **dce_acl_inq_client_permset(3sec)**, **dce_acl_inq_permset_for_creds(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_client_permset

Purpose

Returns the client's permissions corresponding to an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_client_permset(
        handle_t handle,
        uuid_t *mgr_type,
        uuid_t *acl_uuid,
        uuid_t *owner_id,
        uuid_t *group_id,
        sec_acl_permset_t *permset,
        error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl_uuid A pointer to the UUID of the ACL.

owner_id Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id Identifies the group to which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE

exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

Output

permset The set of permissions allowed to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_client_permset()** routine returns the client's permissions that correspond to the ACL. It finds the ACL in the database as defined for this ACL manager type with **dce_acl_register_object_type()**. The client's credentials are determined from the binding handle. The ACL and credentials determine the permission set.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

acl_s_bad_manager_type
The *mgr_type* parameter does not match the manager type in the ACL itself.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_inq_client_pac(3sec)**, **dce_acl_inq_permset_for_pac(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_permset_for_creds

Purpose

Determines a principal's complete extent of access to an object

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_permset_for_creds(
        sec_cred_pa_handle_t *creds,
        sec_acl_t *ap,
        uuid_t *owner_id,
        uuid_t *group_id,
        sec_acl_posix_semantics_t posix_semantics,
        sec_acl_permset_t *perms,
        error_status_t *status);
```

Parameters

Input

creds The security credentials that represent the principal.

ap The ACL that represents the object.

owner_id Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id Identifies the group in which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

posix_semantics
This parameter is currently unused in OSF's implementation.

Output

perms A bit mask containing a 1 bit for each permission granted by the ACL and 0 (zero) bits elsewhere.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **dce_acl_inq_permset_for_creds()** routine returns a principal's complete extent of access to some object. This routine is useful for implementing operations such as the conventional UNIX access function.

The values allowed for the credentials representing the principal include NULL or unauthenticated.

The routine normally returns TRUE, even when the access permissions are determined to be all 0 (zero) bits (**dce_acl_c_no_permissions**). It returns FALSE only on illogical error conditions (such as unsupported ACL entry types), in which case the status output gets the error status code and the *perms* is set to **dce_acl_c_no_permissions**.

All ACL entry types (of type **sec_acl_entry_type_t**) are supported by this routine

Notes

The meanings of the permission bits have no effect on the action of the **dce_acl_inq_permset_for_creds()** routine. The interpretation of the bits is left entirely to the application.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_inq_client_creds(3sec)**, **dce_acl_inq_client_permset(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_prin_and_group.3sec

Purpose

Inquires the principal and group of an RPC caller

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_prin_and_group(
        handle_t handle,
        uuid_t *principal,
        uuid_t *group,
        error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

Output

principal The UUID of the principal of the caller of the RPC.

group The UUID of the group of the caller of the RPC.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_prin_and_group()** routine finds the principal and group of the caller of a remote procedure call. This information is useful for filling in the *owner_id* and *group_id* fields of standard data or object headers. Setting the owner and group make sense only if your ACL manager will handle owners and groups,

which you specify with the **dce_acl_c_has_owner** and **dce_acl_c_has_groups** flags to **dce_acl_register_object_type()**.

If the caller is unauthenticated, the principal and group are filled with the **NIL** UUID, generated through **uuid_create_nil()**.

Examples

```
dce_db_std_header_init(db, &data, ..., &st);
dce_acl_inq_prin_and_group(h, \
 &data.h.owner_id, &data.h.group_id, &st);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages. The **dce_acl_inq_prin_and_group()** routine can return errors from **dce_acl_inq_client_creds()**, **sec_cred_get_initiator()**, and **sec_cred_get_pa_data()**. It generates no error messages of its own.

Related Information

Functions: **dce_acl_register_object_type(3sec)**.

dce_acl_is_client_authorized

Purpose

Checks whether a client's credentials are authenticated

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_is_client_authorized(
        handle_t handle,
        uuid_t *mgr_type,
        uuid_t *acl_uuid,
        uuid_t *owner_id,
        uuid_t *group_id,
        sec_acl_permset_t desired_perms,
        boolean32 *authorized,
        error_status_t *status);
```

Parameters

Input

handle The client's binding handle.

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl_uuid A pointer to the UUID of the ACL.

owner_id Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id Identifies the group to which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

desired_perms
A permission set containing the desired privileges. This is a 32-bit set of permission flags. The flags may represent the conventional file system permissions (read, write, and execute), the extended AFS permissions (owner, insert, and delete), or some other permissions supported by the specific application ACL manager. For example, a bit that is unused for file system permissions may mean withdrawals are allowed for a bank ACL manager, while it may mean matrix inversions are allowed for a CPU ACL manager. The *mgr_type* identifies the semantics of the bits.

Output

authorized A pointer to the TRUE or FALSE return value of the routine.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_is_client_authorized()** routine returns TRUE in the *authorized* parameter if and only if all of the desired permissions (represented as bits in *desired_perms*) are included in the actual permissions corresponding to the *handle*, the *mgr_type*, and the *acl_uuid* UUID. Otherwise, the returned value is FALSE.

Notes

The routine's return value is **void**. The returned **boolean32** value is in the *authorized* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

acl_s_bad_manager_type
The *mgr_type* does not match the manager type in the ACL itself.

error_status_ok
The call was successful.

dce_acl_obj_add_any_other_entry

Purpose

Adds permissions for **any_other** ACL entry to a given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_any_other_entry(
        sec_acl_t *acl,
        sec_acl_permset_t permset,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset The permissions to be granted to **sec_acl_e_type_any_other**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_any_other_entry()** routine adds an ACL entry for **sec_acl_e_type_any_other** access to the specified ACL. It is equivalent to calling the **dce_acl_obj_add_obj_entry()** routine with the **sec_acl_e_type_any_other** entry type, but is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_obj_entry(3sec)**.

dce_acl_obj_add_foreign_entry

Purpose

Adds permissions for an ACL entry for a foreign user or group to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_foreign_entry(
        sec_acl_t *acl,
        sec_acl_entry_type_t entry_type,
        sec_acl_permset_t permset,
        uuid_t *realm,
        uuid_t *id,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type Must be one of the following types:

- **sec_acl_e_type_foreign_user**
- **sec_acl_e_type_foreign_group**.
- **sec_acl_e_type_for_user_deleg**
- **sec_acl_e_type_for_group_deleg**

permset The permissions to be granted to the foreign group or foreign user.

realm The UUID of the foreign cell.

id The UUID identifying the foreign group or foreign user.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_foreign_entry()** routine adds an ACL entry for **sec_acl_e_type_foreign_xxx** access to the specified ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not one of the four specified types.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**, **sec_id_parse_name(3sec)**.

dce_acl_obj_add_group_entry

Purpose

Adds permissions for a group ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_group_entry(
        sec_acl_t *acl,
        sec_acl_permset_t permset,
        uuid_t *group,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset The permissions to be granted to the group.

group The UUID identifying the group.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_group_entry()** routine adds a group ACL entry to the given ACL. It is equivalent to calling the **dce_acl_obj_add_id_entry()** routine with the **sec_acl_e_type_group** entry type, but is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**.

dce_acl_obj_add_id_entry

Purpose

Adds permissions for an ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_id_entry(
        sec_acl_t *acl,
        sec_acl_entry_type_t entry_type,
        sec_acl_permset_t permset,
        uuid_t *id,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type Must be one of the following types:

- **sec_acl_e_type_user**
- **sec_acl_e_type_group**
- **sec_acl_e_type_foreign_other**
- **sec_acl_e_type_user_deleg**
- **sec_acl_e_type_group_deleg**
- **sec_acl_e_type_for_other_deleg**

permset The permissions to be granted to the **user**, **group**, or **foreign_other**.

id The UUID identifying the **user**, **group**, or **foreign_other** to be added

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_id_entry()** routine adds an ACL entry (user or group, domestic or foreign) to the given ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not one of the six specified types.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_group_entry(3sec)**, **dce_acl_obj_add_user_entry(3sec)**.

dce_acl_obj_add_obj_entry

Purpose

Adds permissions for an object (**obj**) ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_obj_entry(
        sec_acl_t *acl,
        sec_acl_entry_type_t entry_type,
        sec_acl_permset_t permset,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type Must be one of these types:

- **sec_acl_e_type_unauthenticated**
- **sec_acl_e_type_any_other**
- **sec_acl_e_type_mask_obj**
- **sec_acl_e_type_user_obj**
- **sec_acl_e_type_group_obj**
- **sec_acl_e_type_other_obj**
- **sec_acl_e_type_user_obj_deleg**
- **sec_acl_e_type_group_obj_deleg**
- **sec_acl_e_type_other_obj_deleg**
- **sec_acl_e_type_any_other_deleg**

permset The permissions to be granted.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_obj_entry()** routine adds an **obj** ACL entry to the given ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_duplicate_entry
An **obj** ACL entry type already exits for the given ACL.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not a valid ACL entry type.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_any_other_entry(3sec)**, **dce_acl_obj_add_unauth_entry(3sec)**.

dce_acl_obj_add_unauth_entry

Purpose

Adds permissions for **unauthenticated** ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_unauth_entry(
        sec_acl_t *acl,
        sec_acl_permset_t permset,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset The permissions to be granted for **sec_acl_e_type_unauthenticated**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_unauth_entry()** routine adds ACL entry for **sec_acl_e_type_unauthenticated** to the given ACL. It is equivalent to calling the **dce_acl_obj_add_obj_entry()** routine with the **sec_acl_e_type_unauthenticated** entry type, but it is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_obj_entry(3sec)**.

dce_acl_obj_add_user_entry

Purpose

Adds permissions for a user ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_add_user_entry(
        sec_acl_t *acl,
        sec_acl_permset_t permset,
        uuid_t *user,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset The permissions to be granted to the user.

user The UUID identifying the user to be added.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_user_entry()** routine adds a user ACL entry to the given ACL. It is equivalent to calling the **dce_acl_obj_add_id_entry()** routine with the **sec_acl_e_type_user** entry type, but it is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**.

dce_acl_obj_free_entries

Purpose

Frees space used by an ACL's entries

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_free_entries(
        sec_acl_t *acl,
        error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be freed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_free_entries()** routine frees space used by an ACL's entries, then sets the pointer to the ACL entry array to NULL and the entry count to 0 (zero).

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_init(3sec)**.

dce_acl_obj_init

Purpose

Initializes an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_obj_init(
        uuid_t *mgr_type,
        sec_acl_t *acl,
        error_status_t *status);
```

Parameters

Input

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl A pointer to the ACL that is to be created.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_init()** routine initializes an ACL. The caller passes in the pointer to the already-existing ACL structure (of type **sec_acl_t**), for which the caller provides the space.

Examples

This example shows the use of **dce_acl_obj_init()** and the corresponding routine to free the entries, **dce_acl_obj_free_entries()**.

```
sec_acl_t acl;
extern uuid_t my_mgr_type;
error_status_t status;
dce_acl_obj_init(&my_mgr_type, &acl, &status);
/* ... use the ACL ... */
dce_acl_obj_free_entries(&acl, &status);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_free_entries(3sec)**.

dce_acl_register_object_type

Purpose

Registers an ACL manager's object type

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_register_object_type(
        dce_db_handle_t db,
        uuid_t *mgr_type,
        unsigned32 printstring_size,
        sec_acl_printstring_t *printstring,
        sec_acl_printstring_t *mgr_info,
        sec_acl_permset_t control_perm,
        sec_acl_permset_t test_perm,
        dce_acl_resolve_func_t resolver,
        void *resolver_arg,
        unsigned32 flags,
        error_status_t *status);
```

Parameters

Input

db The *db* parameter specifies the handle to the backing store database in which the ACL objects are stored. It must be indexed by UUID and not use backing store headers. The database is obtained through **dce_db_open()**, which is called prior to this routine.

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

printstring_size
The number of items in the *printstring* array.

printstring An array of **sec_acl_printstring_t** structures containing the printable representation of each specified permission. These are the printstrings used by **dcecp** or other ACL editors.

mgr_info A single **sec_acl_printstring_t** containing the name and short description for the given ACL manager.

control_perm
The permission set needed to change an ACL, typically **sec_acl_perm_control**. If the value is 0, then anyone is allowed to change the ACL. The permission must be listed in the **printstring**.

test_perm The permission set needed to test an ACL, typically **sec_acl_perm_test**. If the value is 0, then anyone is allowed to test the ACL. The permissions must be listed in the **printstring**.

resolver The function for finding an ACL's UUID.

resolver_arg The argument to pass to the *resolver* function. If using **dce_acl_resolve_by_name()** or **dce_acl_resolve_by_uuid()**, then pass the database handle to the name or UUID backing store database. The backing store must use the standard backing store header. See **dce_db_open(3dce)**.

flags A bit mask with the following possible bit values:

dce_acl_c_orphans_ok
If this bit is specified, it is possible to replace an ACL with one in which no control bits are turned on in any of the ACL entries. (Use the **rdacl_replace** operation to replace an ACL.) This is a write-once operation, and once it has been done, no one can change the ACL.

dce_acl_c_has_owner
If this bit is set, then the ACL manager supports the concept of user owners of objects. This is required to use ACL entries of type **user_obj** and **user_obj_deleg**. entries such as **sec_acl_e_type_user_obj**.

dce_acl_c_has_groups
A similar bit for group owners of objects.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_register_object_type()** routine registers an ACL manager's object types with the ACL library.

The *resolver* function may be the **dce_acl_resolve_by_name()** or the **dce_acl_resolve_by_uuid()** routine, if the application uses the standard header in the backing store database, or it may be some other user-supplied routine, as appropriate. A user-supplied routine must be of type **dce_acl_resolve_func_t**. The *resolver* function finds the UUID of the ACL of the given object. The *resolver*'s parameters must match the type **dce_db_convert_func_t** defined in the file **<dce/aclif.h>**. Observe the use of the resolver function **dce_acl_convert_func()** in **EXAMPLES**.

Unless the **dce_acl_c_orphans_ok** bit is set in the *flags* parameter, all ACLs must always have *someone* able to modify the ACL.

Another way to express this is that if **dce_acl_c_orphans_ok** is cleared in a call to **dce_acl_register_object_type()** where a *control_perm* value is specified, then a subsequent ACL replacement using an ACL that has no control bits set in any nondelegation entry will fail, resulting in the **acl_s_no_control_entries** error. If **dce_acl_c_orphans_ok** is set, but no *control_perm* bits are specified, then **dce_acl_c_orphans_ok** is ignored, and the replacement works in all cases.

Files

/usr/include/dce/aclif.h
Definition of **dce_acl_resolve_func_t**.

Examples

The **dce_acl_register_object_type()** routine should be called once for each type of object that the server manages. A typical call is shown below. The sample code defines three variables: the manager printstring, the ACL printstrings, and the ACL database. Note that the manager printstring does not define any permission bits; they will be

set by the library to be the union of all permissions in the ACL printstring. The code also uses the global **my_uuid** as the ACL manager type UUID. The ACL printstring uses the standard **sec_acl_perm_XXX** bits.

```
include <dce/aclif.h>

/* Manager help. */
sec_acl_printstring_t my_acl_help = {
    "me", "My manager"
};

/*
 * ACL permission descriptions;
 * these are from /usr/include/dce/aclbase.idl
 * This example refrains from redefining any of the
 * conventionally established bits.
 */
sec_acl_printstring_t my_printstring[] = {
    { "r", "read", sec_acl_perm_read },
    { "f", "foobar", sec_acl_perm_unused_00000080 },
    { "w", "write", sec_acl_perm_write },
    { "d", "delete, sec_acl_perm_delete },
    { "c", "control", sec_acl_perm_control }
};

dce_db_open("my_acldb", NULL,
    dce_db_c_std_header | dce_db_c_index_by_uuid,
    (dce_db_convert_func_t)dce_acl_convert_func,
    &dbh, &st);

dce_acl_register_object_type(dbh, &my_manager_uuid,
    sizeof my_printstring / sizeof my_printstring[0],
    my_printstring, &my_acl_help, sec_acl_perm_control,
    0, xxx_resolve_func, NULL, 0, &st);
```

If the ACL manager can use the standard collection of ACL bits (that is, has not defined any special ones), then it can use the global variable **dce_acl_g_printstring** that predefines a printstring. Here is an example of its use:

```
dce_acl_register_object_type(acl_db, &your_mgr_type,
    sizeof dce_acl_g_printstring / sizeof dce_acl_g_printstring[0],
    dce_acl_g_printstring, &your_acl_help,
    dced_perm_control, dced_perm_test, your_resolver, NULL, 0, st);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_group_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_group_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission. CL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission.

Related Information

Functions: **dce_acl_resolve_by_name(3sec)**, **dce_acl_resolve_by_uuid(3sec)**, **dce_db_open(3dce)**.

dce_acl_resolve_by_name

Purpose

Finds an ACL's UUID, given an object's name

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_resolve_by_name(
        handle_t handle,
        sec_acl_component_name_t component_name,
        sec_acl_type_t sec_acl_type,
        uuid_t *mgr_type,
        boolean32 writing,
        void *resolver_arg,
        uuid_t *acl_uuid,
        error_status_t *status);
```

Parameters

Input

handle A client binding handle passed into the server stub. Use **sec_acl_bind()** to create this handle.

component_name
A character string containing the name of the target object.

sec_acl_type The type of ACL to be resolved:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting

the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

writing This parameter is ignored in OSF's implementation.

resolver_arg This argument is passed into **dce_acl_register_object_type()**. It should be a handle for a backing store indexed by name.

Output

acl_uuid The ACL UUID, as resolved by **dce_acl_resolve_by_name()**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_resolve_by_name()** routine finds an ACL's UUID, given an object's name, as provided in the *component_name* parameter. The user does not call this function directly. It is an instance of the kind of function provided to the *resolver* argument of **dce_acl_register_object_type()**.

If **dce_acl_resolve_by_name()** and **dce_acl_resolve_by_uuid()** are inappropriate, the user of **dce_acl_register_object_type()** must provide some other *resolver* function.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_register_object_type(3sec)**, **dce_acl_resolve_by_uuid(3sec)**, **dce_db_open(3dce)**, **dce_db_header_fetch(3dce)**.

dce_acl_resolve_by_uuid

Purpose

Finds an ACL's UUID, given an object's UUID

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

dce_acl_resolve_func_t dce_acl_resolve_by_uuid(
        handle_t handle,
        sec_acl_component_name_t component_name,
        sec_acl_type_t sec_acl_type,
        uuid_t *mgr_type,
        boolean32 writing,
        void *resolver_arg,
        uuid_t *acl_uuid,
        error_status_t *status);
```

Parameters

Input

handle A client binding handle passed into the server stub. Use **sec_acl_bind()** to create this handle.

component_name
A character string containing the name of the target object. (The **dce_acl_resolve_by_uuid()** routine ignores this parameter.)

sec_acl_type The type of ACL to be resolved:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

mgr_type A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

writing This parameter is ignored in OSF's implementation.

resolver_arg This argument is passed into **dce_acl_register_object_type()**. It should be a handle for a backing store indexed by UUID.

Output

acl_uuid The ACL UUID, as resolved by **dce_acl_resolve_by_uuid()**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_resolve_by_uuid()** routine finds an ACL's UUID, given an object's UUID, as provided through the *handle* parameter. The user does not call this function directly. It is an instance of the kind of function provided to the *resolver* argument of **dce_acl_register_object_type()**.

If **dce_acl_resolve_by_uuid()** and **dce_acl_resolve_by_name()** are inappropriate, the user of **dce_acl_register_object_type()** must provide some other *resolver* function.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_register_object_type(3sec)**, **dce_acl_resolve_by_name(3sec)**, **dce_db_open(3dce)**, **dce_db_header_fetch(3dce)**.

dce_aud_close

Purpose

Closes an audit trail file. Used by client/server applications and audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_close(
        dce_aud_trail_t at,
        unsigned32 *status);
```

Parameters

Input

at A pointer to an audit trail descriptor returned by a previous call to **dce_aud_open()**.

Output

status The status code returned by this routine.

Description

The **dce_aud_close()** function releases data structures of file openings, RPC bindings, and other memory associated with the audit trail that is specified by the audit trail descriptor.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_open(3sec)**.

dce_aud_commit

Purpose

Writes the audit record in the audit trail file. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void dce_aud_commit(
        dce_aud_trail_t at,
        dce_aud_rec_t ard,
        unsigned32 options,
        unsigned16 format,
        unsigned32 outcome,
        unsigned32* status);
```

Parameters

Input

at Designates an audit trail file to which the completed audit record will be written. The audit trail file must have been previously opened by a successful call to the **dce_aud_open()** function.

ard Designates an audit record descriptor that was returned by a previously successful call to one of the **dce_aud_start_*()** functions. The content of this record buffer will be appended to the audit trail specified by *at*.

options Bitwise **OR** of option values described below. A value of 0 (zero) for *options* results in the default operation (normal writing to the file without flushing to stable storage). The possible option value is

aud_c_evt_commit_sync
Flushes the audit record to stable storage before the function returns.

aud_c_evt_always_log
Unconditionally logs the audit record to the audit trail.

aud_c_evt_always_alarm
Unconditionally displays the audit record on the console.

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accommodate changes in the formats of the event specific information, or use different formats dynamically.

outcome The event outcome to be stored in the header. The possible event-outcome values are as follows:

aud_c_esl_cond_success
The event completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is not known. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_commit()** function determines whether the event should be audited given the event outcome. If it should be audited, the function completes the audit

record identified by **ard** and writes it to the audit trail designated by **at**. If any of the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** options is selected, the event is always audited (logged or an alarm message is sent to the standard output).

If the **aud_c_evt_commit_sync** option is selected, the function attempts to flush the audit record to stable storage. If the stable storage write cannot be performed, the function either continues to try until the stable-storage write is completed or returns an error status.

Upon successful completion, **dce_aud_commit()** calls **dce_aud_discard()** internally to release the memory of the audit record that is being committed.

The caller should not change the outcome between the **dce_aud_start()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_wrong_protection_level
Client used the wrong protection level.

aud_s_dmn_disabled
The daemon is disabled for logging.

aud_s_log_access_denied
The client’s access to the Audit log was denied.

aud_s_cannot_gettime
The audit library cannot backup a trail file due to failure of the **utc_gettime()** call.

aud_s_cannot_getgmtime
The audit library cannot backup a trail file due to failure of the **utc_gmtime()** call.

aud_s_rename_trail_file_rc
Cannot rename the audit trail file.

aud_s_cannot_reopen_trail_file_rc
Internally, the audit trail file was being reopened and the reopening of the file failed.

aud_s_rename_trail_index_file_rc
Internally, the audit trail index file was being renamed and the renaming of the file failed.

aud_s_cannot_reopen_trail_index_file_rc
Internally, the audit trail index file was being reopened and the reopening of the file failed.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_invalid_outcome
The event outcome parameter that was provided is invalid.

aud_s_outcomes_inconsistent
The event outcome parameter is inconsistent with the outcome parameter provided in the **dce_aud_start()** call.

aud_s_trl_write_failure
The audit record cannot be written to stable storage.

aud_s_ok The call was successful.

Status codes passed from **dce_aud_discard()**

Status codes passed from **rpc_binding_inq_auth_caller()**

Status codes passed from **dce_acl_is_client_authorized()**

Status codes passed from **audit_pickle_dencode_ev_info()** (RPC idl compiler)

Related Information

Functions: **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_discard

Purpose

Discards an audit record (releases the memory). Used by client/server applications and trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_discard(
        dce_aud_rec_t ard,
        unsigned32* status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to one of the **dce_aud_start_*()** functions or the **dce_aud_next()** function.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_discard()** function releases the memory used by the audit record descriptor and the associated audit record that is to be discarded.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **dce_aud_free_header()**

Related Information

Functions: **dce_aud_open(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_free_ev_info

Purpose

Frees the memory allocated for an event information stucture returned from calling **dce_aud_get_ev_info()**. Used by the audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_free_ev_info(
        dce_aud_ev_info_t *event_info,
        unsigned32 *status);
```

Parameters

Input

event_info Designates an event-specific information item returned from a previous successful call to the **dce_aud_get_ev_info()** function.

Output

status The status code returned by this routine.

Description

The **dce_aud_free_ev_info()** function frees the memory allocated for an event information stucture returned by a previous successful call to the **dce_aud_get_ev_info()** function.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_get_ev_info(3sec)**, **dce_aud_next(3sec)**.

dce_aud_free_header

Purpose

Frees the memory allocated to a designated audit record header structure. Used by the audit trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

void dce_aud_free_header(
        dce_aud_hdr_t *header,
        unsigned32 *status);
```

Parameters

Input

ard Designates a pointer to an audit record header structure that was returned by a previous successful call to the **dce_aud_get_header()** function.

Output

status The status code returned by this routine.

Description

The **dce_aud_free_header()** frees the memory allocated to a designated audit record header structure. The designated audit record header is usually obtained from an audit record by calling **dce_aud_get_header()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_get_header(3sec)**, **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_get_ev_info

Purpose

Returns a pointer to an event information stucture (**dce_aud_ev_info_t**). Used by the audit trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

void dce_aud_get_ev_info(
        dce_aud_rec_t ard,
        dce_aud_ev_info_t **event_info,
        unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to the **dce_aud_next()** function.

Output

event_info Returns an event-specific information item of the designated audit record. Returns NULL if there are no more information items.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_get_ev_info()** function returns a pointer to an event information structure. The designated record is usually obtained from an audit trail by calling **dce_aud_open()** and **dce_aud_next()**. If there is more than one item of event-specific information in the audit record, then one item is returned through one call to **dce_aud_get_ev_info()**. The order in which the items are returned is the same as the

order in which they were included in the audit record through **dce_aud_put_ev_info()** calls. This function allocates the memory to hold the human-readable representation of the audit record and returns the address of this memory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_get_header

Purpose

Gets the header of a specified audit record. Used by the audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_get_header(
        dce_aud_rec_t ard,
        dce_aud_hdr_t **header,
        unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to the **dce_aud_next()** function.

Output

header Returns the header information of the designated audit record.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_get_header()** function gets the header information of a designated audit record. The designated record is usually obtained from an audit trail by calling **dce_aud_open()** and **dce_aud_next()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_length

Purpose

Gets the length of a specified audit record. Used by client/server applications and trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

unsigned32 dce_aud_length(
        dce_aud_rec_t ard,
        unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to **dce_aud_next()**, or one of the **dce_aud_start_*()** functions.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_length()** function gets the length of a designated audit record. The designated record (in binary format) may be obtained from an audit trail by calling the **dce_aud_open()** and **dce_aud_next()** functions.

Applications can use this function to know how much space an audit record will use before it is committed. This function can also be used by audit trail analysis and examination tools to determine the space that a previously committed audit record uses before it is read.

Return Values

The size of the specified audit record in number of bytes.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok The call was successful.

Status codes passed from **idl_es_encode_dyn_buffer()**

Status codes passed from **audit_pickle_dencode_ev_info()**
(RPC IDL compiler)

Status codes passed from **idl_es_handle_free()**

Status codes passed from **rpc_sm_client_free()**

Related Information

Functions: **dce_aud_next(3aud)**, **dce_aud_open(3aud)**, **dce_aud_put_ev_info(3aud)**, **dce_aud_start(3aud)**, **dce_aud_start_with_name(3aud)**, **dce_aud_start_with_pac(3aud)**, **dce_aud_start_with_server_binding(3aud)**.

dce_aud_next

Purpose

Reads the next audit record from a specified audit trail file into a buffer. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_next(
        dce_aud_trail_t *at,
        char *predicate,
        unsigned16 format,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

predicate Criteria for selecting the audit records that are to be read from the audit trail file. A predicate statement consists of an attribute and its value, separated by any of the following operators: = (equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to):

- *attribute=value*
- *attribute>value*
- *attribute>=value*
- *attribute<value*
- *attribute<=value*

Attribute names are case sensitive, and no space is allowed within a predicate expression. Multiple predicates are delimited by a comma, in the following form:

attribute1=*value1*,*attribute2*>*value2*, . . .

No space is allowed between predicates. Note that when multiple predicates are defined, the values are logically ANDed together.

The possible attribute names, their values, and allowable operators are as follows:

SERVER The UUID of the server principal that generated the record. The attribute value must be a UUID string. Operator allowed: = (equal to).

EVENT The audit event number. The attribute value must be a hexadecimal number. Operator allowed: = (equal to).

OUTCOME
The event outcome of the record. The possible attribute values are **SUCCESS**, **FAILURE**, **PENDING**, or **DENIAL**. Operator allowed: = (equal to).

STATUS The authorization status of the client. The possible attribute values are **DCE** for DCE authorization (PAC based), and **NAME** for name-based authorization. Operator allowed: = (equal to).

CLIENT The UUID of the client principal. The attribute value must be a UUID string. Operator allowed: = (equal to).

TIME The time the record was generated. The attribute value must be a null-terminated string that expresses an absolute time. Operators allowed: <= (less than or equal to), < (less than), >= (greater than or equal to), and > (greater than).

CELL The UUID of the client's cell. The attribute value must be a UUID string. Operator allowed: = (equal to).

GROUP The UUID of one of the client's group(s). The attribute value must be a UUID string. Operator allowed: = (equal to).

ADDR The address of the client. The attribute is typically the string representation of an RPC binding handle. Operator allowed: = (equal to).

FORMAT The format version number of the audit event record. The attribute value must be an integer. Operators allowed: = (equal to), < (less than), and > (greater than).

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accomodate changes in the formats of the event specification information, or use different formats dynamically.

Output

ard A pointer to the audit record descriptor containing the returned record.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. See ''Errors'' for a list of the possible status codes and their meanings.

Description

The **dce_aud_next()** function attempts to read the next record from the audit trail file specified by the audit trail descriptor, *at*. This function also defines the predicate to be used to search for the next record and returns a matching record if one exists. The **dce_aud_next()** function can be used to search for successive records in the trail that match the defined predicate. By default, if no predicate is explicitly defined, the function returns the next record from the audit trail.

If no record satisfies the predicate specified for the call, a value of zero (**NULL**) is returned through *ard*.

The value returned through **ard** can be supplied as an input parameter to the functions **dce_aud_get_header()**, **dce_aud_length()**, **dce_aud_discard()**, **dce_aud_print()**, **dce_aud_get_event()**, and **dce_aud_get_ev_info()**.

Storage allocated by this function must be explicitly freed by a call to **dce_aud_discard()** with *ard* as the input parameter.

If the function successfully reads an audit trail record, the cursor associated with the audit trail descriptor *at* will be advanced to the next record in the audit trail. The calling routine does not need to set or move the cursor explicitly.

If no appropriate record can be found in the audit trail, an *ard* value of **NULL** is returned and the cursor is advanced to the end of the audit trail. If a call is unsuccessful, the position of the cursor does not change.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successfully completed.

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid.

aud_s_trail_file_corrupted
The trail file is corrupted.

aud_s_index_file_corrupted
The index trail file is corrupted.

aud_s_cannot_allocate_memory
The **malloc()** call failed.

Status codes passed from **idl_es_decode_buffer()**

Status codes passed from **idl_es_handle_free()**

Status codes passed from **audit_pickle_dencode_ev_info()**
(RPC IDL compiler)

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_get_header(3sec)**, **dce_aud_length(3sec)**, **dce_aud_get_ev_info(3sec)**, **dce_aud_open(3sec)**, **dce_aud_discard(3sec)**, **dce_aud_print(3sec)**, **dce_aud_get_event(3sec)**.

dce_aud_open

Purpose

Opens a specified audit trail file for read or write. Used by client/server applications and trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_open(
        unsigned32 flags,
        char *description,
        unsigned32 first_evt_number,
        unsigned32 num_of_evts,
        dce_aud_trail_t *at,
        unsigned32 *status);
```

Parameters

Input

flags Specifies the mode of opening. The flags parameter is set to the bitwise OR of the following values:

- **aud_c_trl_open_read**
- **aud_c_trl_open_write**
- **aud_c_trl_ss_wrap**

description A character string specifying an audit trail file to be opened. If **description** is NULL, the default audit trail file is opened. When the audit trail file is opened for write, the default audit trail is an RPC interface to a local audit daemon.

first_evt_num
The lowest assigned audit event number used by the calling server.

num_of_evts The number of audit events defined for the calling server.

Output

at A pointer to an audit trail descriptor. When the audit trail descriptor is no longer needed, it must be released by calling the **dce_aud_close()** function.

status Returns the status code from this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_open()** function opens the audit trail file specified by the **description** parameter. If **description** is NULL, the function uses the default audit trail which is an RPC interface to the local audit daemon.

This function must be invoked after the server has finished registering with RPC and before calling **rpc_server_listen()**.

If the **flags** parameter is set to **aud_c_trl_open_read**, the specified file (**description** cannot be null in this case) is opened for reading audit records, using the **dce_aud_next()** function. If **flags** is set to **aud_c_trl_open_write**, the specified file or the default audit trail device is opened and initialized for appending audit records using the **dce_aud_commit()** function. Only one of the **aud_c_trl_open_read** and **aud_c_trl_open_write** flags may be specified in any call to **dce_aud_open()**. If the **flags** parameter is set to **aud_c_trl_ss_wrap**, the audit trail operation is set to **wrap** mode. The **aud_c_trl_ss_wrap** flag has meaning only if you specify the **aud_c_trl_open_write** flag.

If the audit trail specified is a file and the calling server does not have the read and write permissions to the file, a NULL pointer is returned in **at**, and **status** is set to **aud_s_cannot_open_trail_file_rc**. The same values will be returned if the default audit trail file is used (that is, through an audit daemon) and if the calling server is not authorized to use the audit daemon to log records.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

aud_s_trl_invalid_open_flags
The flags argument must include either **aud_c_trl_open_read** or **aud_c_trl_open_write** flag, but not both.

aud_s_cannot_open_dmn_binding_file
The local audit daemon trail file is designated, but the daemon's binding file cannot be opened.

Status codes passed from **sec_login_get_current_context()**
When the local audit daemon trail file is designated, a login context is needed for making secure audit logging RPC to the audit daemon.

aud_s_cannot_open_dmn_identity_file
The local audit daemon trail file is designated, but the daemon's identity file cannot be opened.

Status codes passed from **rpc_binding_set_auth_info()**
When the local audit daemon trail file is designated, **dce_aud_open()** sets authentication information in the RPC binding handle for making secure audit logging RPC to the audit daemon. This is done by calling **rpc_binding_set_auth_info()**.

aud_s_cannot_open_trail_file_rc
Cannot open a local trail file.

aud_s_cannot_allocate_memory
Memory allocation failed.

aud_s_cannot_init_trail_mutex
Audit trail mutex initialization failed.

Status codes passed from **rpc_server_inq_bindings()**
When filtering is turned on, **dce_aud_open()** gets the caller's RPC bindings to be used for registering an RPC interface in receiving filter update notification from the local audit daemon. This is done by calling **rpc_server_inq_bindings()**.

Status codes passed from **rpc_binding_to_string_binding()**
: When filtering is turned on, the caller's RPC bindings are converted to string bindings before they are stored in a file. This is done by calling **rpc_binding_to_string_binding()**.

aud_s_cannot_mkdir
: Cannot create a directory for storing the bindings file for the filter update notification interface.

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_next(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_prev

Purpose

Reads the previous audit record from a specified audit trail file into a buffer. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_prev(
        dce_aud_trail_t* at,
        char *predicate,
        unsigned16 format,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

predicate Criteria for selecting the audit records that are to be read from the audit trail file. A predicate statement consists of an attribute and its value, separated by any of the following operators: = (equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to).

- *attribute=value*
- *attribute>value*
- *attribute>=value*
- *attribute<value*
- *attribute<=value*

Attribute names are case sensitive, and no space is allowed within a predicate expression. Multiple predicates are delimited by a comma, in the following form:

attribute=value1,attribute>value2, . . .

No space is allowed between predicates. Note that when multiple predicates are defined, the values are logically ANDed together.

The possible attribute names, their values, and allowable operators are as follows:

SERVER The UUID of the server principal that generated the record. The attribute value must be a UUID string. Operator allowed: = (equal to).

EVENT The audit event number. The attribute value must be a hexadecimal number. Operator allowed: = (equal to).

OUTCOME The event outcome of the record. The possible attribute values are: **SUCCESS**, **FAILURE**, **PENDING**, or **DENIAL**. Operator allowed: = (equal to).

STATUS The authorization status of the client. The possible attribute values are **DCE** for DCE authorization (PAC based) and **NAME** for name-based authorization. Operator allowed: = (equal to).

TIME The time the record was generated. The attribute value must be a null terminated string that expresses an absolute time. Operators allowed: <= (less than or equal to), < (less than), >= (greater than or equal to), and > (greater than).

CELL The UUID of the client's cell. The attribute value must be a UUID string. Operator allowed: = (equal to).

GROUP The UUID of one of the client's group(s). The attribute value must be a UUID string. Operator allowed: = (equal to).

ADDR The address of the client. The attribute is typically the string representation of an RPC binding handle. Operator allowed: = (equal to).

FORMAT The format version number of the audit event record. The attribute value must be an integer. Operators allowed: = (equal to), < (less than), and > (greater than).

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accommodate changes in the formats of the event specification information, or use different formats dynamically.

Output

ard A pointer to the audit record descriptor containing the returned record.

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. See ''Errors'' for a list of the possible status codes and their meanings.

Description

The **dce_aud_prev()** function attempts to read the previous record from the audit trail file specified by the audit trail descriptor, *at*. This function also defines the predicate to be used to search for the previous record and returns a matching record if one exists. **dce_aud_prev()** can be used to search for previous records in the trail file that match the defined predicate. By default, if no predicate is explicitly defined, the function returns the previous record read from the audit trail.

If no record satisfies the predicate specified for the call, a value of zero (**NULL**) is returned in *ard*.

The value returned in *ard* can be supplied as an input parameter to the functions: **dce_aud_get_header()**, **dce_aud_length()**, **dce_aud_discard()**, **dce_aud_print()**, **dce_aud_get_event()**, and **dce_aud_get_ev_info()**.

Storage allocated by this function must be explicitly freed by a call to **dce_aud_discard()** with *ard* as the input parameter.

If the function successfully reads an audit trail record, the cursor associated with the audit trail descriptor *at* will be moved to the previous record in the audit trail file. The calling routine does not need to set or move the file cursor explicitly.

If no appropriate record can be found in the audit trail, an *ard* value of **NULL** is returned and the cursor is set back to the beginning of the audit trail. If a call is unsuccessful, the position of the cursor does not change.

Return Value

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successfully completed

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid

aud_s_trail_file_corrupted
The audit trail is corrupted

aud_s_index_file_corrupted
The index trail file is corrupted

aud_s_cannot_allocate_memory
The **malloc()** call failed

Status codes passed from **idl_es_decode_buffer()**

Status codes passed from **idl_es_handle_free()**

Status codes passed from **audit_pickle_dencode_ev_info()**
(RPC IDL compiler)

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_get_header(3sec)**, **dce_aud_length(3sec)**, **dce_aud_get_ev_info(3sec)**, **dce_aud_open(3sec)**, **dce_aud_discard(3sec)**, **dce_aud_print(3sec)**, **dce_aud_get_event(3sec)**.

dce_aud_print

Purpose

Formats an audit record into human-readable form. Used by audit trail examination and analysis tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_print(
        dce_aud_rec_t ard,
        unsigned32 options,
        char **buffer,
        unsigned32 *status);
```

Parameters

Input

ard
An audit record descriptor. This descriptor can be obtained from an opened audit trail by calling **dce_aud_next()** or it can be a new record established by calling one of the **dce_aud_start_*()** functions.

options
The options governing the transformation of the binary audit record information into a character string. The value of the *options* parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all the optional information (that is, groups, address, and event specific information).

aud_c_evt_groups_info
Includes the groups' information.

aud_c_evt_address_info
Includes the address information.

aud_c_evt_specific_info
Includes the event specific information.

Output

buffer Returns the pointer to a character string converted from the audit record specified by *ard*.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_print()** function transforms the audit record specified by *ard* into a character string and places it in a buffer. The buffer is allocated using **malloc()**, and must later be freed by the caller. (This function allocates the memory to hold the human-readable text of the audit record and returns the address of this memory in the *buffer* parameter.)

The *options* parameter is set to the bitwise OR of flag values defined in the **dce/audit.h** header file. A value of 0 (zero) for options will result in default operation, that is, no group, address, and event-specific information is included in the output string.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_cannot_allocate_memory
The **malloc()** call failed.

aud_s_ok The call was successful.

Status codes passed from **sec_login_get_current_context()**

Status codes passed from **sec_login_inquire_net_info()**

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_put_ev_info

Purpose

Adds event-specific information to a specified audit record buffer. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void dce_aud_put_ev_info(
        dce_aud_rec_t ard,
        dce_aud_ev_info_t info,
        unsigned32 *status);
```

Parameters

Input

ard A pointer to an audit record descriptor initialized by one of the **dce_aud_start_*()** functions.

info A data structure containing an event-specific information item that is to be appended to the tail of the audit record identified by **ard**. The possible formats of the event-specific information are listed in the **sec_intro(3sec)** reference page of this book.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_put_ev_info()** function adds event-specific information to an audit record. The event-specific information is included in an audit record by calling **dce_aud_put_ev_info()** one or more times. The order of the information items

included by multiple calls is preserved in the audit record, so that they may be read in the same order by the **dce_aud_get_ev_info()** function. This order is also observed by the **dce_aud_print()** function. The **info** parameter is a pointer to an instance of the self-descriptive **dce_aud_ev_info_t** structure.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The input audit record descriptor is invalid.

aud_s_evt_tail_info_exceeds_limit
The tail portion of the audit trail record has exceeded its limit of 4K.

aud_s_ok The call was successful.

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_reset

Purpose

Resets the cursors and the file pointers of the specified audit trail file. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_reset(
        dce_aud_trail_t *at,
        unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened by the function **dce_aud_open()**.

Output

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. For a list of the possible status codes and their meanings, see ‘‘Errors’’.

Description

The **dce_aud_reset()** function resets the cursors and the file pointers of the specified audit trail file. The function is used to explicitly reset the current cursors and file pointers to the beginning of the audit trail file.

dce_aud_open() must be called to specify the desired audit trail file. Otherwise, **dce_aud_reset()** will reset the audit trail which is currently set in the value of *at*.

If the call is successful, the file cursors are set to the beginning of the file.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages. The possible status codes and their meanings are:

aud_s_ok The call was successful

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid

Related Information

Functions: **dce_aud_rewind(3sec)**, **dce_aud_clean(3sec)**, **dce_aud_open(3sec)**.

dce_aud_rewind

Purpose

Rewinds the specified audit trail file. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void dce_aud_rewind(
        dce_aud_trail_t *at,
        unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for writing by the function **dce_aud_open()**.

Output

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. For a list of the possible status codes and their meanings, see ‘‘Errors’’.

Description

The **dce_aud_rewind()** function rewinds the specified audit trail file. This function can be used to instantly clean up the audit trail file if it is no longer needed.

dce_aud_open() must be called to specify the desired audit trail file, and the specified audit trail file must be opened with the **aud_c_trl_open_write** flag. Otherwise, the routine will rewind the audit trail which is currently set in the value of *at*.

If the call is successful, the file cursors are set to the beginning of the file.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

aud_s_invalid_trail_descriptor
The Audit Trail descriptor is invalid

aud_s_trl_invalid_open_flag
The Audit Trail is opened with open flag

aud_s_rewind_trail_file
The ftruncate() call failed on trail file

aud_s_rewind_index_file
The ftruncate() call failed on index file

Related Information

Functions: **dce_aud_clean(3sec)**, **dce_aud_open(3sec)**.

dce_aud_set_trail_size_limit

Purpose

Sets a limit to the audit trail size. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void dce_aud_set_trail_size_limit(
        dce_aud_trail_t at,
        unsigned32 file_size_limit_value,
        unsigned32 * status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

file_size_limit_value
The desired maximum size of the audit trail file, in bytes.

Output

status Returns the status code of this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_set_trail_size_limit()** function can be used by an application that links with **libaudit** to set the maximum size of the audit trail. This function must be called immediately after calling **dce_aud_open()**.

For added flexibility, the environment variable **DCEAUDITTRAILSIZE** can also be used to set the maximum trail size limit.

If none of these methods are used for setting the trail size, then a hardcoded limit of 2 megabytes will be assumed.

If set, the value of the environment variable **DCEAUDITTRAILSIZE** overrides the value set by this function. Any of the values set by **DCEAUDITTRAILSIZE** or this function overrides the hardcoded default.

When the size limit is reached, the current trail file is copied to another file. The name of this new file is the original filename appended by a timestamp. For example, if the name of the original trail file is **central_trail**, its companion trail file is named **central_trail.md_index**. These two files will be copied to the following locations:

central_trail.1994-09-26-16-38-15
central_trail.1994-09-26-16-38-15.md_index

When a trail file is copied to a new file by the audit library because it has reached the size limit, a serviceability message is issued to the console notifying the user that an audit trail file (and its companion index file) is available to be backed up. Once the backup is performed, it is advisable to remove the old trail file, so as to prevent running out of disk space.

Auditing will then continue, using the original name of the file, (in our example, **central_trail**).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_trail_descriptor
The audit trail descriptor *at* is null.

aud_s_ok The call is successful.

Related Information

Functions: **dce_aud_open(3sec)**.

dce_aud_start

Purpose

Determines whether a specified event should be audited given the client binding information and the event outcome. Used by client/server applications

Synopsis

```
#include <dce/audit.h>

void dce_aud_start(
        unsigned32 event,
        rpc_binding_handle_t binding,
        unsigned32 options,
        unsigned32 outcome,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

binding Specifies the client's RPC binding handle from which the client identification information is retrieved to set the *client*, *cell*, *num_groups*, *groups*, and *addr* fields in the audit record header.

options Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, or **aud_c_evt_address_info**).

It can also be used to specify whether the audit records are always logged (**aud_c_evt_always_log**) or that an alarm message is always sent to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable **DCEAUDITOFF** has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* other than **aud_c_esl_cond_unknown** must be provided when calling the **dce_aud_commit()** function.

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start()** function determines if an audit record should be generated for the specified event. The decision is based on the event filters, an environment variable (**DCEAUDITOFF**), the client's identity provided in the **binding** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, *ard*). If the event does not need to be audited, a NULL *ard* is returned. If an internal error(s) has occurred, a NULL pointer is returned in *ard*. If the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start()** function is designed to be used by RPC applications. Non-RPC applications that use the DCE authorization model (that is, DCE ACL and PAC) must use **dce_aud_start_with_pac()**. Non-RPC applications that do not use the DCE authorization model must use **dce_aud_start_with_name()**.

This function obtains the client identity information from the RPC binding handle and records it in the newly-created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. This function can be called multiple times after calling **dce_aud_start()** and before calling **dce_aud_commit()**. A completed audit record will be appended to an audit trail file or sent to the audit daemon (depending on the value of the **description** parameter used in the previous call to **dce_aud_open**) by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start()** call, **dce_aud_start()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **rpc_binding_to_string_binding()**

Status codes passed from **rpc_string_free()**

Status codes passed from **dce_aud_start_with_name()**

Status codes passed from **sec_cred_get_initiator()**

Status codes passed from **sec_cred_get_v1_pac()**

Status codes passed from **dce_aud_start_with_pac()**

Status codes passed from **sec_cred_get_delegate()**

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_name

Purpose

Determines whether a specified event should be audited given the client/server name and the event outcome. Used by non-RPC based client/server applications that do not use the DCE authorization model

Synopsis

```
#include <dce/audit.h>

void dce_aud_start_with_name(
        unsigned32event ,
        unsigned_char_t *client,
        unsigned_char_t *address,
        unsigned32 options,
        unsigned32 outcome,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

client Specifies the principal name of the remote client/server.

address Specifies the address of the remote client/server. The address could be in any format of the underlying transport protocol.

options Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**).

It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed. The value of the **options**

parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and

the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters or if the environment variable **DCEAUDITOFF** has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, the function may not be able to determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_name()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (**DCEAUDITOFF**), the client's identity provided in the input parameters, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_name()** function is designed to be used by non-RPC applications that do not use the DCE authorization model (that is, DCE PAC and ACL). RPC applications must use **dce_aud_start()**. Non-RPC applications that use the DCE authorization model must use **dce_aud_start_with_pac()**.

This function records the input identity parameters in the newly created audit record descriptor.

Event-specific information can be added to the record by using the **dce_aud_put_ev_info()** function, which can be called multiple times after calling any of the **dce_aud_start_*** and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_name()** call, **dce_aud_start_with_name()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_name()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_name()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_name()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **sec_rgy_site_open()**

Status codes passed from **sec_id_parse_name()**

Status codes passed from **dce_aud_start_with_pac()**

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_pac

Purpose

Determines whether a specified event must be audited given the client's privilege attribute certificate (PAC) and the event outcome. Used by non-RPC based client/server applications that use the DCE authorization model

Synopsis

```
#include <dce/audit.h>

void dce_aud_start_with_pac(
        unsigned32 event,
        sec_id_pac_t *pac,
        unsigned_char_t *address,
        unsigned32 options,
        unsigned32 outcome,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

pac Specifies the client's PAC from which the client's identification information is retrieved to set the *client*, *cell*, *num_groups*, and *groups* fields in the audit record header.

address Specifies the client's address. The address can be in any format that is native to the underlying transport protocol.

options Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**). It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the

standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
: Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
: Includes the groups' information in the audit record header.

aud_c_evt_address_info
: Includes the client address information in the audit record header.

aud_c_evt_always_log
: Bypasses the filter and indicates that the event must be logged.

aud_c_evt_always_alarm
: Bypasses the filter and indicates that an alarm message must be sent to the system console for the event.

outcome The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
: The event was completed successfully.

aud_c_esl_cond_denial
: The event failed because of access denial.

aud_c_esl_cond_failure
: The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
: The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable **DCEAUDITOFF** has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_pac()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (**DCEAUDITOFF**), the client's identity provided in the **pac** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, then an audit record descriptor will always be created and returned.

The **dce_aud_start_with_pac()** function is designed to be used by non-RPC applications that use the DCE authorization model (that is, DCE PAC and ACL). RPC applications must use **dce_aud_start()**. Non-RPC applications that do not use the DCE authorization model must use **dce_aud_start_with_name()**.

This function obtains the client's identity information from the client's privilege attribute certificate (PAC) and records it in the newly created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. This function can be called multiple times after calling any of the **dce_aud_start_*** functions and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling the **dce_aud_commit()** function.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_pac()** call, **dce_aud_start_with_pac()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_pac()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_pac()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_pac()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **sec_rgy_site_open()**

Status codes passed from **sec_rgy_properties_get_info()**

Status codes passed from **uuid_create_nil()**

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_server_binding

Purpose

Determines whether a specified event must be audited given the server binding information and the event outcome. Used by client/server applications

Synopsis

```
#include <dce/audit.h>

void dce_aud_start_with_server_binding(
        unsigned32 event,
        rpc_binding_handle_t binding,
        unsigned32 options,
        unsigned32 outcome,
        dce_aud_rec_t *ard,
        unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

binding Specifies the server's RPC binding handle from which the server identification information is retrieved to set the client, cell, and addr fields in the audit record header. Note that when an application client issues an audit record, the server identity is represented in the *client* field of the record.

options This parameter can be used to specify the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**). It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_address_info
: Includes the server address information in the audit record header.

aud_c_evt_always_log
: Bypasses the filter and indicates that the event must be logged.

aud_c_evt_always_alarm
: Bypasses the filter and indicates that an alarm message must be sent to the system console for the event.

outcome The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
: The event was completed successfully.

aud_c_esl_cond_denial
: The event failed because of access denial.

aud_c_esl_cond_failure
: The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
: The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
: The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable **DCEAUDITOFF** has been set, a NULL pointer is returned. If

the function is called with **outcome** set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_server_binding()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (**DCEAUDITOFF**), the server's identity provided in the **binding** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_server_binding()** function is designed to be used by RPC applications. Non-RPC applications that use the DCE authorization model must use the **dce_aud_start_with_pac()** function. Non-RPC applications that do not use the DCE authorization model must use the **dce_aud_start_with_name()** function.

This function obtains the server identity information from the RPC binding handle and records it in the newly created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. The **dce_aud_put_ev_info()** function can be called multiple times after calling any of the **dce_aud_start_*** functions and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_server_binding()** call, **dce_aud_start_with_server_binding()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_server_binding()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of 0 (zero) or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_server_binding()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_server_binding()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **rpc_binding_inq_auth_info()**

Status codes passed from **rpc_binding_to_string_binding()**

Status codes passed from **dce_aud_start_with_name()**

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**.

dce_aud_start_with_uuid

Purpose

Determines whether a specified event should be audited given the client/server UUID and the event outcome. Used by client/server applications which already know the UUIDs of their clients and wish to avoid the overhead of the audit library acquiring them

Synopsis

```
#include <dce/audit.h>

void dce_aud_start_with_uuid(
        unsigned32 event,
        uuid_t server_uuid,
        uuid_t client_uuid,
        uuid_ trealm_uuid,
        unsigned_char_t *address,
        unsigned3 2options,
        unsigned32 outcome,
        dce_aud_rec_t * ard,
        unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

server_uuid Specifies the calling application's principal uuid.

client_uuid Specifies the remote client/server's principal uuid.

realm_uuid Specifies the remote client/server's cell uuid.

address Specifies the remote client/server's address. The address could be in any format of the underlying transport protocol.

options Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**).

It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed. The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_unknown
The event outcome (denial, failure, or success) is still unknown.

aud_c_esl_cond_success
The event completed successfully.

aud_c_esl_cond_denial
The event failed due to access denial.

aud_c_esl_cond_failure
The event failed due to reasons other than access denial.

aud_c_esl_cond_pending
The event outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable **DCEAUDITOFF** has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome*, different from **unknown**, must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_uuid()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (**DCEAUDITOFF**), the client's identity provided in the input parameters, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_uuid()** function is designed to be used by RPC applications that know their client's identity in UUID form. Otherwise, RPC applications should use **dce_aud_start()**. Non-RPC applications that use the DCE authorization model should use **dce_aud_start_with_pac()**. The **dce_aud_start_with_name()** function should be used by non-RPC applications that do not use the DCE authorization model.

This function records the input identity parameters in the newly-created audit record descriptor.

Event-specific information can be added to the record by using the **dce_aud_put_ev_info()** function, which can be called multiple times after calling any of the **dce_aud_start_*** and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_uuid()** call, **dce_aud_start_with_uuid()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_uuid()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_uuid()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_uuid()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *DCE 1.2.2 Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok The call was successful.

Status codes passed from **dce_aud_start_with_pac()**

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S