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.

Chapter 1

DCE Routines

dce_intro

Purpose

Introduction to the DCE routines

Description

The DCE routines provide several facilities that are applicable across more than one DCE component. They can be divided into the following major areas:

DCE Attribute Interface Routines

These routines allow applications to define and access attribute types (schema entries) in a schema of your choice. They are based on the extended registry attribute (ERA) interface, which defines and accesses attribute types in the register database schema.

For more information about the individual attribute interface routines, see the **dce_attr_intro(3dce)** reference page.

DCE Configuration Routines

These routines return information based on the contents of the local DCE configuration file, which is created during the DCE cell-configuration or machine-configuration process.

For more information about the various individual configuration routines, see the **dce_config_intro(3dce)** reference page.

DCE Backing Store Routines

These routines allow you to maintain typed data between program invocations. The backing store routines can be used in servers, in clients or in standalone programs that do not involve remote procedure calls (RPCs).

For more information about the individual backing store routines, see the **dce_db_intro(3dce)** reference page.

DCE Messaging Interface Routines

These routines give you access to message catalogs, to specific message texts and message IDs, and to in-memory message tables.

For more information about the individual messaging interface routines, see the **dce_msg_intro(3dce)** reference page.

DCE Server Routines

These routines are used by servers to register themselves with DCE. This includes RPC runtime, the local endpoint mapper, and the namespace. Routines are also available to set up DCE security so that servers can receive and invoke authenticated RPCs.

For more information about the individual server routines, see the **dce_server_intro(3dce)** reference page.

DCE Serviceability Routines

These routines are intended for use by servers that export the serviceability interface defined in **service.idl**. There are also a set of DCE serviceability macros can be used for diagnostic purposes, and to create a serviceability handle.

For more information about the individual serviceability routines, see the **dce_svc_intro(3dce)** reference page. For more information about the individual DCE serviceability macros, see the **DCE_SVC_INTRO(3dce)** reference page.

DCE Host Daemon Application Programming Interface

These routines give management applications remote access to various data, servers, and services on DCE hosts.

For more information about the individual host daemon application programming interface routines, see the **dced_intro(3dce)** reference page.

dce_attr_intro

Purpose

Introduction to the DCE attribute interface routines

Description

The DCE attribute interface API allows applications to define and access attributes types (schema entries) in a schema of your choice. It is based on the extended registry attribute (ERA) interface, which defines and accesses attribute types in the registry database schema. Except for the binding methods, the two APIs are similar.

Note however, that the extended registry attribute API provides routines to create attribute types in the registry schema, to create and manipulate attribute instances, and to attach those instances to objects. The DCE attribute interface in its current state provides calls only to create attribute types.

The DCE Attribute Interface Routines

The DCE attribute interface consists of the following routines:

dce_attr_sch_bind()
Returns an opaque handle of type **dce_attr_sch_handle_t** to a schema object specified by name and sets authentication and authorization parameters for the handle.

dce_attr_sch_bind_free()
Releases an opaque handle of type **dce_attr_sch_handle_t**.

dce_attr_sch_create_entry()
Creates a schema entry in a schema bound to with **dce_attr_sch_bind()**.

dce_attr_sch_update_entry()
Updates a schema entry in a schema bound to with **dce_attr_sch_bind()**.

dce_attr_sch_delete_entry()
Deletes a schema entry in a schema bound to with **dce_attr_sch_bind()**.

dce_attr_sch_scan()
Reads a specified number of schema entries.

dce_attr_sch_cursor_init()
Allocates resources to and initializes a cursor used with **dce_attr_sch_scan()**. The **dce_attr_sch_cursor_init()** routine makes a remote call that also returns the current number of schema entries in the schema.

dce_attr_sch_cursor_alloc()
Allocates resources to a cursor used with **dce_attr_sch_scan()**. The **dce_attr_sch_cursor_alloc()** routine is a local operation.

dce_attr_sch_cursor_release()
Releases states associated with a cursor created by **dce_attr_sch_cursor_alloc()** or **dce_attr_sch_cursor_init()**.

dce_attr_sch_cursor_reset()
Reinitializes a cursor used with **dce_attr_sch_scan()**. The reset cursor can then be reused without releasing and reallocating.

dce_attr_sch_lookup_by_id()
Reads a schema entry identified by attribute type UUID.

dce_attr_sch_lookup_by_name()
Reads a schema entry identified by attribute name.

dce_attr_sch_get_acl_mgrs()
Retrieves the manager types of the ACLs protecting objects dominated by a named schema.

dce_attr_sch_aclmgr_strings()
Returns printable ACL strings associated with an ACL manager protecting a schema object.

Data Types and Structures

dce_attr_sch_handle_t
An opaque handle to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

dce_attr_component_name_t
A pointer to a character string used to further specify a schema object.

dce_bind_auth_info_t
An enumeration that defines whether or not the binding is authenticated. This data type is defined exactly as the **sec_attr_bind_auth_info_t** data type in the ERA interface. See the **sec_intro(3sec)** reference page for more information on **sec_attr_bind_auth_info_t**.

dce_attr_schema_entry_t
: A structure that defines a complete attribute entry for the schema catalog. This data type is defined exactly as the **sec_attr_schema_entry_t** data type in the ERA interface. See the **sec_intro(3sec)** reference page for more information on **sec_attr_schema_entry_t**.

dce_attr_cursor_t
: A structure that provides a pointer into a database and is used for multiple database operations. This cursor must minimally represent the object indicated by **dce_attr_sch_handle_t**. The cursor may additionally represent an entry within that schema.

dce_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 is defined exactly as the **sec_attr_schema_entry_parts_t** data type in the ERA interface. See the **sec_intro(3sec)** reference page for more information on **sec_attr_schema_entry_parts_t**.

dce_cf_intro

Purpose

Introduction to the DCE configuration routines

Description

The DCE configuration routines return information based on the contents of the local DCE configuration file, which is created during the DCE cell-configuration or machine-configuration process. A configuration file is located on each DCE machine; it contains the host's name, the primary name of the cell in which the host is located, and any aliases for that cell name.

The configuration routines can also be used to get the following additional information corollary to the host name:

- The host's principal name
- Binding information to the host

The configuration file on machines that belong to internationalized DCE cells also contains the pathname to the code set registry object file on the host.

The security service component on each DCE machine must be able to find out, by strictly local means, its machine's host name, the host machine's principal name, and its cell's name. The DCE configuration routines exist primarily to enable security components to do these things. But because this information can be useful to DCE applications as well, these routines are made available as part of the general application programming interface.

Note that *host name* as used throughout this section refers to the *DCE host name* (that is, the machine's **/.../***cellname*/ *host_directory*/ *hostname* entry in the CDS namespace), and not, for example, its Domain Name Service (DNS) host name, which could be quite different from the DCE name.

The DCE configuration routines are as follows:

dce_cf_binding_entry_from_host()
: Returns the host binding entry name.

dce_cf_dced_entry_from_host()
Returns the **dced** entry name on a host.

dce_cf_find_name_by_key()
Returns a string tagged by key (this is a lower-level utility routine that is used by the others).

dce_cf_free_cell_aliases()
Frees a list of cell aliases for a cell.

dce_cf_get_cell_aliases()
Returns a list of cell aliases for a cell.

dce_cf_get_cell_name()
Returns the primary cell name for the local cell.

dce_cf_get_csrgy_filename()
Returns the pathname of the local code set registry object file.

dce_cf_get_host_name()
Returns the host name relative to a local cell.

dce_cf_prin_name_from_host()
Returns the host's principal name.

dce_cf_profile_entry_from_host
Returns the host's profile entry.

dce_cf_same_cell_name()
Indicates whether or not two cell names refer to the same cell.

Files

*dcelocal/***dce_cf.db**
The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

The format of the configuration file is as follows:

Each of the entries is tagged with its own identifier, which must be the first nonblank token on a line that does not begin with a **#** (number sign) comment character. The second token on a line is assumed to be the name associated with the tag that was detected in front of it.

For example, **cellname** and **hostname** are tags, identifying the cell name and host name, respectively, for the machine on which the configuration file is located. A sample configuration file could have the following contents, which would identify the host **brazil** in the **osf.org** cell:

```
cellname /.../osf.org
hostname hosts/brazil
```

Text characterized by the following is ignored:

- Garbage lines (lines that do not conform to the previously described format)
- Leading and trailing spaces in lines
- Additional tokens appearing on a line after the second token

The configuration file should be writable only by privileged users, and readable by all.

Output

The DCE configuration routines return names without global or cell-relative prefixes, such as the following:

host_directory/hostname

or

principalname

where *host_directory* is usually **hosts**.

However, the DCE Name Service Interface (NSI) routines require names passed to them to be expressed either in a cell-relative form or as global names. Cell-relative names have the following form:

/.:/*host_directory/hostname*

Global names, with the global root prefix **/.../** and the cell name, have the following form:

/.../*cellname/host_directory/hostname*

Therefore, an application must add either the cell-relative prefix (**/.:/**) or correct global prefix (**/.../***cellname*) to any name it receives from a DCE configuration routine before it passes the name to an NSI routine. (NSI routines all have names beginning with **rpc_ns_**.) For example, the name *host_directory/hostname* would become the following, if expressed in cell-relative form:

/.:/*hosts/hostname*

The cell-relative form of the name *principalname* would be

/.:/sec/principals/*principalname*

where *hostname* and *principalname* are the host's name and principal name, respectively.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_dced_entry_from_host(3dce)**, **dce_cf_find_name_by_key(3dce)**, **dce_cf_free_cell_aliases(3dce)**, **dce_cf_get_cell_aliases(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_csrgy_filename(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**, **dce_cf_profile_entry_from_host(3dce)**, **dce_cf_same_cell_name(3dce)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components, DCE 1.2.2 Command Reference*.

dce_db_intro

Purpose Introduction to the DCE backing store interface

Description

The DCE backing store interface allows you to maintain typed data between program invocations. For example, you might store application-specific configuration data in a backing store, and then retrieve it from the backing store when the application restarts. The backing store routines can be used in servers, in clients or in standalone programs that do not involve remote procedure calls (RPCs). A program can have more than one backing store open at the same time.

Sometimes the backing store is called a database. For instance, the associated IDL file is **dce/database.idl**, and the name of the backing store routines begin with **dce_db_**. The backing store is, however, not a full-fledged database in the conventional sense, and it has no support for SQL or for any other query system.

Backing Store Data

The backing store interface provides for the tagged storage and retrieval of typed data. The tag (or retrieval key) can be either a UUID or a standard C string. For a specific backing store, the data type must be specified at compile time, and is established through the IDL encoding services. Each backing store can contain only a single data type.

Each data item (also called a data object or data record) consists of the data stored by a single call to a storage routine (**dce_db_store()**, **dce_db_store_by_name()**, or **dce_db_store_by_uuid()**). Optionally, data items can have headers. If a backing store has been created to use headers, then every data item must have a header. For a description of the data item header, see the section in this reference page entitled **Data Types and Structures**.

Encoding and Decoding in the Backing Store

When an RPC sends data between a client and a server, it serializes the user's data structures by using the IDL encoding services (ES), described in the *DCE 1.2.2 Application Development Guide*.

The backing store uses this same serialization scheme for encoding and decoding, informally called *pickling*, when storing data structures to disk. The IDL compiler, **idl**, writes the routine that encodes and decodes the data.

This routine is passed to **dce_db_open()**, remembered in the handle, and used by the store and fetch routines:

- **dce_db_fetch()**
- **dce_db_fetch_by_name()**
- **dce_db_fetch_by_uuid()**
- **dce_db_header_fetch()**
- **dce_db_store()**
- **dce_db_store_by_name()**
- **dce_db_store_by_uuid()**

Memory Allocation

When fetching data, the encoding services allocate memory for the data structures that are returned. These services accept a structure, and use **rpc_sm_allocate()** to provide additional memory needed to hold the data.

The backing store library does not know what memory has been allocated, and therefore cannot free it. For fetch calls that are made from a server stub, this is not a problem, since the memory is freed automatically when the server call terminates. For fetch calls that are made from a nonserver, the programmer is responsible for freeing the memory.

Programs that call the fetch or store routines, such as **dce_db_fetch()**, outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first.

The Backing Store Routines

Many of the backing store routines appear in three versions: plain, by name, and by UUID. The plain version will work with backing stores that were created to be indexed either by name, or by UUID, while the restricted versions accept only the matching type. It is advantageous to use the restricted versions when they are appropriate, because they provide type checking by the compiler, as well as visual clarity of purpose.

The backing store routines are as follows, listed in alphabetical order:

dce_db_close()
Frees the handle returned by **dce_db_open()**. It closes any open files and releases all other resources associated with the backing store.

dce_db_delete()
Deletes an item from a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**.

dce_db_delete_by_name()
Deletes an item only from a backing store that is indexed by name.

dce_db_delete_by_uuid()
Deletes an item only from a backing store that is indexed by UUID.

dce_db_fetch()
Retrieves data from a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**.

dce_db_fetch_by_name()
Retrieves data only from a backing store that is indexed by name.

dce_db_fetch_by_uuid()
Retrieves data only from a backing store that is indexed by UUID.

dce_db_free()
Releases the data supplied from a backing store.

dce_db_header_fetch()
Retrieves a header from a backing store.

dce_db_inq_count()
Returns the number of items in a backing store.

dce_db_iter_done()
Terminates and iteration operation initiated by **dce_db_iter_start()**. It should be called when iteration is done.

dce_db_iter_next()
Returns the key for the next item from a backing store that is indexed by name or by UUID. The **db_s_no_more** return value indicates that there are no more items.

dce_db_iter_next_by_name()
Returns the key for the next item only from a backing store that is indexed by name. The **db_s_no_more** return value indicates that there are no more items.

dce_db_iter_next_by_uuid()
Returns the key for the next item only from a backing store that is indexed by UUID. The **db_s_no_more** return value indicates that there are no more items.

dce_db_iter_start()
Prepares for the start of iteration.

dce_db_lock()
Locks a backing store. A lock is associated with an open backing store's handle. The storage routines, **dce_db_store()**, **dce_db_store_by_name()**, and **dce_db_store_by_uuid()**, all acquire the lock before updating.

dce_db_open()
Creates a new backing store or opens an existing one. The backing store is identified by a filename. Flags allow you to

- Create a new backing store, or open an existing one.
- Create a new backing store indexed by name, or indexed by UUID.
- Open an existing backing store read/write, or read-only.
- Use the standard data item header, or not.

The routine returns a handle by which subsequent routines can reference the opened backing store.

dce_db_std_header_init()
Initializes a standard backing store header retrieved by **dce_db_header_fetch()**. It only places the values into the header, and does not write into the backing store.

dce_db_store()
Stores a data item into a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**.

dce_db_store_by_name()
Stores a data item only into a backing store that is indexed by name.

dce_db_store_by_uuid()
Stores a data item only into a backing store that is indexed by UUID.

dce_db_unlock()
Unlocks a backing store.

Data Types and Structures

dce_db_handle_t
An opaque handle to a backing store. Use **dce_db_open()** to acquire the handle.

dce_db_header_t
The data structure that defines a standard backing store header for data items. Use **dce_db_header_fetch()** to retrieve it from a backing store and **dce_db_std_header_init()** to initialize it.

dce_db_convert_func_t
An opaque pointer to the data conversion function to be used when storing or retrieving data. This function is specified as an argument to **dce_db_open()** at open time. It converts between native format and on-disk (serialized) format. It is generated from the IDL file by the IDL compiler.

Cautions

You can not use conformant arrays in objects stored to a backing store. This is because the idl-generated code that encodes (pickles) the structure has no way to predict or detect the size of the array. When the object is fetched, there will likely be insufficient space provided for the structure, and the array's data will destroy whatever is in memory after the structure.

Files

database.idl

database.h

db.h

dbif.h

Related Information

Books: *DCE 1.2.2 Application Development Guide*

dce_msg_intro

Purpose Introduction to the DCE messaging interface

Description

All DCE message texts are assigned a unique message ID. This is a 32-bit number, with the special value of all-bits-zero reserved to indicate success. All other numbers are divided into a technology/component that identifies the message catalog, and an index into the catalog.

All messages for a given component are stored in a single message catalog generated by the **sams** utility when the component is built. (The messages may also be compiled into the application code, rendering the successful retrieval of message text independent of whether or not the message catalogs were correctly installed.)

In typical use, a message is first retrieved from a message catalog, allowing localization to occur. If this fails, the default message is retrieved from an in-memory table. If this fails, a fallback text identifying the message number is generated. The two most useful routines, **dce_error_inq_text()** and **dce_msg_get()**, and the DCE **printf** routines follow these rules. The rest of this API gives direct access for special needs.

The **dce_msg_cat_*()** routines provide a DCE abstraction to standard message catalog routines, mapping DCE message IDs to message catalog names. They offer a convenient way of opening and accessing a message catalog simply by supplying the ID of a message contained in it, rather than the name of the catalog itself. Once opened, the catalog is accessed by means of an opaque handle (the **dce_msg_cat_handle_t** typedef).

The DCE Messaging Routines

The messaging routines are as follows, listed in alphabetical order:

dce_error_inq_text()
: Retrieves from the installed DCE component message catalogs the message text associated with an error status code returned by a DCE library routine.

dce_fprintf()
Functions much like **dce_printf()**, except that it prints the message and its arguments on the specified stream.

dce_msg_cat_close()
Closes the message catalog (which was opened with **dce_msg_cat_open())**.

dce_msg_cat_get_msg()
Retrieves the text for a specified message.

dce_msg_cat_open()
Opens the message catalog that contains the specified message, and returns a handle that can be used in subsequent calls to **dce_msg_cat_get_msg()**.

dce_msg_define_msg_table()
Registers an in-memory table containing the messages.

dce_msg_get()
Retrieves the text for a specified message. A convenience form of the **dce_msg_get_msg()** routine.

dce_msg_get_cat_msg()
A convenience form of the **dce_msg_cat_get_msg()** routine. Unlike **dce_msg_cat_get_msg()**, **dce_msg_get_cat_msg()** does not require the message catalog to be explicitly opened.

dce_msg_get_default_msg()
Retrieves a message from the application's in-memory tables.

dce_msg_get_msg()
Retrieves the text for a specified message.

dce_msg_translate_table()
The **dce_msg_translate_table()** routine overwrites the specified in-memory message table with the values from the equivalent message catalogs.

dce_pgm_fprintf()
Equivalent to **dce_fprintf()**, except that it prepends the program name and appends a newline.

dce_pgm_printf()
Equivalent to **dce_printf()**, except that it prepends the program name and appends a newline.

dce_pgm_sprintf()
Equivalent to **dce_sprintf()**, except that it prepends the program name and appends a newline.

dce_printf() Retrieves the message text associated with the specified message ID, and prints the message and its arguments on the standard output.

dce_sprintf()
Retrieves the message text associated with the specified message ID, and prints the message and its arguments into an allocated string that is returned.

Data Types and Structures

dce_error_string_t
An array of characters big enough to hold any error text returned by **dce_error_inq_text()**.

dce_msg_cat_handle_t
An opaque handle to a DCE message catalog. (Use **dce_msg_cat_open()** to get a handle.)

Files

dce/dce_msg.h

Related Information

Books: *DCE 1.2.2 Application Development Guide*

dce_server_intro

Purpose

Introduction to the DCE server routines

Description

The routines described on this reference page are used by servers to register themselves with DCE. This includes registering with the RPC runtime, the local endpoint mapper, and the namespace. Routines are also available to set up DCE security so that servers can receive and invoke authenticated RPCs.

The DCE Server Routines

The server routines are as follows, listed in alphabetical order:

dce_server_disable_service()
: Unregisters an individual interface of a DCE server from the RPC runtime, and marks the server's endpoints as disabled in the **dced**'s endpoint mapper service.

dce_server_enable_service()
: Registers an individual interface (application service) of a DCE server with the RPC runtime, and marks the server's endpoints as enabled in the **dced**'s endpoint mapper service.

dce_server_inq_attr()
: Obtains application-specific attribute data from the **dced** server configuration data.

dce_server_inq_server()
: Obtains the server configuration data **dced** used to start the server.

dce_server_inq_uuids()
: Obtains the UUIDs that **dced** used in its **srvrconf** and **srvrexec** facilities to identify the server's configuration and execution data.

dce_server_register()
: Registers a DCE server by establishing a server's binding information, registering its services (represented by interface IDs) with the RPC

runtime, and entering its endpoints in the **dced**'s endpoint mapper service.

dce_server_sec_begin()
Prepares a server to receive and generate authenticated RPCs.

dce_server_sec_done()
Releases the resources previously set up by a call to **dce_server_sec_begin()**.

dce_server_unregister()
Unregisters a DCE server by unregistering a servers services (interfaces) from the RPC runtime, and removing the server's endpoints from the **dced**'s endpoint mapper service.

dce_server_use_protseq()
Registers a protocol sequence to use for the server.

Data Types and Structures

dce_server_handle_t
An opaque data structure containing information the runtime uses to establish the server with DCE.

dce_server_register_data_t
A structure that contains an interface handle (generated by IDL), a default EPV, and a count and array of **dce_server_type_t**s for services that use RPC object types.

dce_server_type_t
A structure containing a manager type UUID and an RPC entry-point vector (EPV) that specified which routines implement the IDL interface for the specific type.

server_t See **dced_intro(3dce)** for a complete description of **server_t**.

Files

dce/dced.h

dce/dced_base.idl

Related Information

Books: *DCE 1.2.2 Application Development Guide*

dce_svc_intro

Purpose

Introduction to the DCE serviceability interface

Description

The routines listed below are intended to be used by servers that export the serviceability interface defined in **service.idl**. The complete list of these remote serviceability implementation calls is as follows (the remote operation name is given in the left column, and the corresponding implementation routine is given in the right column).

Remote Operation	Implementation Routine
dce_svc_set_route	**dce_svc_routing**
dce_svc_set_dbg_route	**dce_svc_debug_routing**
dce_svc_set_dbg_levels	**dce_svc_debug_set_levels**
dce_svc_inq_components	**dce_svc_components**
dce_svc_inq_table	**dce_svc_table**
dce_svc_filter_control	**dce_svc_filter**
dce_svc_inq_stats	**dce_svc_inq_stats**

These routines perform all the necessary processing (except for checking clients' authorization) that must be done by the application manager to implement the remote serviceability operations.

Note that most of these routines have little meaning except as implementations of remote operations. However, the **dce_svc_routing()**, **dce_svc_filter()**, **dce_svc_debug_routing()** and **dce_svc_debug_set_levels()** routines can conceivably be used by servers as purely local operations (for example, in order to allow routing and debug levels to be set via command line flags when the server is invoked).

The **dce_svc_log_** routines provide read access to **BINFILE** format logs which are created and written by the DCE serviceability routines; see **svcroute(5)** for further

information. The **dce_svc_log_handle_t** typedef is an opaque pointer to a handle for an opened log file.

Applications that use the serviceability interface can install a routine that will be effectively hooked into the operation of the interface. If a filter is installed, it will be called whenever one of the serviceability output routines (**dce_svc_printf()**) is about to output a message; whenever this happens, the filter will receive a group of parameters that describe the message that is about to be output and the circumstances that provoked the action. The filter can then allow the message output to proceed, or suppress the message.

Along with the filter routine itself, the application also installs a filter control routine, whose purpose is to permit the behavior of the filter to be altered dynamically while the application is running. The **dce_svc_filter()** routine is the interface's call-in to such an installed filter control.

The DCE Serviceability Routines

The serviceability routines are as follows, listed in alphabetical order:

dce_assert() Adds runtime ''can't happen'' assertions to programs (such as, programming errors).

dce_svc_components()
Returns an array containing the names of all components in the program that have been registered with the **dce_svc_register()** routine.

dce_svc_debug_routing()
Specifies both the level of an applications's serviceability debug messaging, and where the messages are routed.

dce_svc_debug_set_levels()
Sets serviceability debugging message level(s) for a component.

dce_svc_define_filter()
Lets applications define serviceability filtering routines.

dce_svc_filter()
Controls the behavior of the serviceability message filtering routine, if one exists.

dce_svc_log_close()
Closes an open binary format serviceability log and releases all internal state associated with the handle.

dce_svc_log_get()
Reads the next entry from a binary format serviceability log.

dce_svc_log_open()
Opens the specified file for reading.

dce_svc_log_rewind()
Rewinds the current reading position of the specified (by *handle*) log file to the first record.

dce_svc_printf()
Provides the normal call for writing or displaying serviceability messages.

dce_svc_register()
Registers a serviceability handle and subcomponent table.

dce_svc_routing()
Specifies how normal (non-debug) serviceability messages are routed.

dce_svc_set_progname()
If not called, the application's generated serviceability messages will be identified by its process ID.

dce_svc_table()
Returns the serviceability subcomponent table registered with the specified component.

dce_svc_unregister()
Destroys a serviceability handle, releasing all allocated resources associated with the handle.

Data Types and Structures

dce_svc_filter_proc_t
The prototype of a serviceability filtering routine.

dce_svc_filterctl_proc_t
The prototype of a serviceability filter-control routine.

dce_svc_handle_t
An opaque handle to generate serviceability messages. (Use **dce_svc_register()** or **DCE_DEFINE_SVC_HANDLE** to obtain one.)

dce_svc_log_handle_t
An opaque handle to an open serviceability binary format log file. (Use **dce_svc_log_open()** to obtain one.)

dce_svc_log_prolog_t
A structure containing data about a serviceability binary format log entry.

dce_svc_prolog_t
A structure containing the initial message parameters passed to the filtering routine.

Files

dce/service.idl

dce/dce_svc.h

Related Information

Books: *DCE 1.2.2 Application Development Guide*

dced_intro

Purpose

Introduction to the DCE host daemon routines

Description

This introduces the DCE host daemon application programming interface: the **dced** API. This API gives management applications remote access to various data, servers, and services on DCE hosts. Servers manage their own configuration in the local **dced** by using the routines starting with **dce_server**, introduced in the **dce_server_intro(3dce)** reference page.

The dced API Naming Conventions

All of the **dced** API routine names begin with the **dced_** prefix. This API contains some specialized routines that operate on services represented by the following keywords in the routine names:

hostdata The host data management service stores host-specific data such as the host name, the host's cell name, and other data, and it provides access to these data items.

server The server control service configures, starts, and stops servers, among other things. Applications must distinguish two general states of server control: server configuration (**srvrconf**) and server execution (**srvrexec**).

secval The security validation service maintains a host's principal identity and ensures applications that the DCE security daemon is genuine.

keytab The key table management service remotely manages key tables.

The **dced** also provides the endpoint mapper service which has its own API, described with the RPC API. These routines begin with **rpc_ep** and **rpc_mgmt_ep**.

Since some of the **dced** daemon's services require the same operations (but on different data types), the **dced** API also contains generic routines that may operate on more than one of the preceding services. For example, you use the routine **dced_object_read()** to read a data item (object) from the **hostdata**, **srvrconf**, **srvrexec**, or **keytab** services.

dced Binding Routines

A binding must be established to a **dced** service on a particular host before you can use any other **dced** routines. The resources of the **dced** binding should also be released when an application is finished with the service.

dced_binding_create()
Establishes a **dced** binding to a host service.

dced_binding_from_rpc_binding()
Establishes a **dced** binding to a **dced** service on the host specified in an already-established RPC binding handle to any server.

dced_binding_set_auth_info()
Sets authentication, authorization, and protection level information for a **dced** binding handle.

dced_binding_free()
Releases the resources of a **dced** binding handle.

Generic Entry Routines

All data maintained by **dced** is managed as entries. Most of the services of **dced** have lists of entries traversed with a cursor that describe where the actual data is maintained.

dced_entry_add()
Adds a **keytab** or **hostdata** entry.

dced_entry_remove()
Removes a **hostdata** or **keytab** data entry from **dced**.

dced_initialize_cursor()
Obtains a list of data entries from **dced** and sets a cursor at the beginning of the list.

dced_entry_get_next()
Obtains the next data entry from a list of entries.

dced_release_cursor()
Releases the resources associated with a cursor which traverses a service's list of entries.

dced_list_get()
Returns the list of data entries maintained by a DCE host service.

dced_list_release()
Releases the resources of a list of entries.

dced_inq_id()
Obtains the UUID associated with an entry name.

dced_inq_name()
Obtains the name associated with an entry UUID.

Generic Routines to Read Data Objects

These routines obtain the actual data for items to which entries refer (objects).

dced_object_read()
Reads one data item of a **dced** service, based on the entry UUID.

dced_object_read_all()
Reads all the data of a **dced** service's entry list.

dced_objects_release()
Releases the resources allocated for data obtained.

Host Data Management Routines

dced_hostdata_create()
Creates a **hostdata** item and the associated entry.

dced_hostdata_read()
Reads a **hostdata** item.

dced_hostdata_write()
Replaces an existing **hostdata** item.

dced_hostdata_delete()
Deletes a **hostdata** item from a specific host and removes the associated entry.

Server Configuration Control Routines

dced_server_create()
Creates a DCE server's configuration data.

dced_server_modify_attributes()
Modifies a DCE server's configuration data.

dced_server_delete()
Deletes a DCE server's configuration data.

dced_server_start()
Starts a DCE-configured server.

Server Execution Control Routines

dced_server_disable_if()
Disables a service provided by a server.

dced_server_enable_if()
Re-enables a service provided by a server.

dced_server_stop()
Stops a DCE-configured server.

Security Validation Routines

dced_secval_start()
Starts a host's security validation service.

dced_secval_validate()
Validates that the DCE security daemon (**secd**) used by a specific host is legitimate.

dced_secval_status()
Returns a status parameter of TRUE if the security validation service is activated and FALSE if not.

dced_secval_stop()
Stops a host's security validation service.

Key Table Management Routines

dced_keytab_create()
Creates a key table with a list of keys in a new file.

dced_keytab_delete()
Deletes a key table file and removes the associated entry.

dced_keytab_initialize_cursor()
Obtains a list of keys from a key table and sets a cursor at the beginning of the list.

dced_keytab_get_next_key()
Returns a key from a cached list, and advances the cursor.

dced_keytab_release_cursor()
Releases the resources associated with a cursor that traverses a key table.

dced_keytab_add_key()
Adds a key to a key table.

dced_keytab_change_key()
Changes a key in both a key table and in the security registry.

dced_keytab_remove_key()
Removes a key from a key table.

Data Types and Structures

The following data types used with the **dced** API are defined in **dce/dced_base.idl** and are shown here in alphabetical order.

dced_attr_list_t
This data structure specifies the configuration attributes to use when you start a server via **dced**. The structure consists of the following:

count An **unsigned32** number representing the number of attributes in the list.

list An array of configuration attributes where each element is of type **sec_attr_t** . This data type is described in the **sec_intro(3sec)** reference page. For **dced**, the **list[i].attr_id** field can have values of either **dced_g_uuid_fileattr** specifying plain text or **dced_g_uuid_binfileattr** specifying binary data.

dced_binding_handle_t
A **dced** binding handle is an opaque pointer that refers to information that includes a **dced** service (**hostdata**, **srvrconf**, **srvrexec** , **secval**, or **keytab**) and RPC binding information for a specific DCE host daemon.

dced_cursor_t
The entry list cursor is an opaque pointer used to keep track of a location in an entry list between calls that traverse the list.

dced_entry_t
An *entry* is the structure that contains information about a data item (or object) maintained by a **dced** service. The actual data is maintained elsewhere. Each entry consists of the following structure members:

id A unique identifer of type **uuid_t** that **dced** maintains for every data item it maintains

name The name for the data item. The data type is **dced_string_t** .

description A brief description the data item (of type **dced_string_t**) for the convenience of human users.

storage_tag A string of type **dced_string_t** describing the location of the actual data. This is implementation-specific and may be a file (with a pathname) on the host system or a storage identifier for the **dced** process.

dced_entry_list_t

An entry list is a uniform way to list the data items a **dced** service maintains. The entry list structure contains a list of all the entries for a given service. For example, the complete list of all entries of hostdata, server configuration data, server execution data, and keytab data are each maintained in separate entry lists. The structure consists of the following:

count An **unsigned32** number representing the number of entries in the list.

list An array of entries where each element is of type **dced_entry_t**.

dced_key_t A key consists of the following structure members:

principal A **dced_string_t** type string representing the principal for the key.

version An **unsigned32** number representing the version number of the key.

authn_service
An **unsigned32** number representing the authentication service used.

passwd A pointer to a password. This is of type **sec_passwd_rec_t** .

See also the security introduction reference page, **sec_intro(3sec)**.

dced_key_list_t

A key list contains all the keys for a given key table and consists of the following structure elements:

count An **unsigned32** number representing the number of keys in the list.

list An array of keys where each element is of type **dced_key_t**.

dced_keytab_cursor_t

The keytab cursor is an opaque pointer used to keep track of a location in a key list between calls that traverse the list.

dced_opnum_list_t

A list of operation numbers is used in the **service_t** structure. This structure consists of the following fields:

count An **unsigned32** number representing the number of operations in the list.

list An array of UUIDs where each element is of type **uuid_t**.

dced_service_type_t

The **dced** service type distinguishes the services provided by **dced**. It is an enumerated type used mainly in a parameter of the **dced_binding_from_rpc_binding()** routine. It can have one of the following values:

dced_e_service_type_hostdata
The host data management service.

dced_e_service_type_srvrconf
The server configuration management service.

dced_e_service_type_srvrexec
The server execution management service.

dced_e_service_type_secval
The security validation service.

dced_e_service_type_keytab
The key table management service.

dced_e_service_type_null
A NULL service type used internally.

dced_string_t

This data type is a character string from the Portable Character Set (PCS).

dced_string_list_t

A list of strings with the following format:

count An **unsigned32** number representing the number of strings in the list.

list An array of strings where each element is of type **dced_string_t**.

dced_tower_list_t

A list of protocol towers used in the **service_t** structure. This structure consists of the following fields:

count An **unsigned32** number representing the number of protocol towers in the list.

list An array of pointers where each element is a pointer to a protocol tower of the type **sec_attr_twr_set_p_t**. This data type is described in the **sec_intro(3sec)** reference page.

server_fixedattr_t

This structure is a field in the **server_t** structure. It contains the following fields:

startupflags

This field is of type **unsigned32** and can be any combination of the following bits:

server_c_startup_at_boot

This means that **dced** should start the server when **dced** is started.

server_c_startup_auto

This means that the server can be started automatically if **dced** determines there is a need.

server_c_startup_explicit

This means **dced** can start the server if it receives an explicit command to do so via **dced_server_start()** or the **dcecp** operation **server start**.

server_c_startup_on_failure

This means that the server should be restarted by **dced** if it exits with an unsuccesful exit status.

Several bits are also reserved for vendor-specific startup and include the following:

server_c_startup_vendor1

server_c_startup_vendor2

server_c_startup_vendor3

server_c_startup_vendor4

flags This represents the execution state of the server and is the **unsigned32** type. This field is maintained only by **dced** and should not be modified. Valid values to check for are self-explanatory and include the following:

server_c_exec_notrunning

server_c_exec_running

Several bits are also reserved for vendor-specific execution states and include:

server_c_exec_vendor1

server_c_exec_vendor2

server_c_exec_vendor3

server_c_exec_vendor4

program This is the full path name of the server and is of type **dced_string_t** .

arguments This is a list of arguments for the server and is of type **dced_string_list_t** .

prerequisites
This is an advisory field that means this server is a client of other prerequisite servers whose IDs are in a list of type **uuid_list_t**. The UUIDs should be the **id** fields from the **server_t** structures of the relevent servers.

keytables This is a list of keytab entry UUIDs representing the key tables for this server and is of type **uuid_list_t**.

posix_uid This is a POSIX execution attribute for the user ID. It is of type **unsigned32**.

posix_gid This is a POSIX execution attribute for the group ID. It is of type **unsigned32**.

posix_dir This is a POSIX execution attribute for the directory in which the server started when it is invoked. It is of type **dced_string_t**.

server_t The DCE host daemon describes a server as follows:

id Each server has a unique ID of type **uuid_t**.

name Each server's name is of type **dced_string_t**.

entryname The server's entry name is a hint as to where the server appears in the namespace. This is of type **dced_string_t**.

services Each server offers a list of services specified in a list of type **service_list_t**. This structure has the following members:

count An **unsigned32** number representing the number of services in the list.

list A pointer to an array of services where each element is of type **service_t** .

fixed This is a set of attributes common to all DCE implementations. The data type is **server_fixedattr_t**.

attributes This field is of type **dced_attr_list_t** and contains a list of attributes representing the behavior specific to a particular server or host.

prin_names This field is a list of principal names for the server and is of type **dced_string_list_t**.

exec_data Data about an executing server is maintained in a tagged union (named **tagged_union**) with a discriminator of type **unsigned32** named **execstate** representing the server's execution state.

The union has the following two execution states:

server_c_exec_notrunning
For the case where the server is not running, the union member has no value. For example:

```
if(server->exec_data.execstate ==
server_c_exec_notrunning)
server->exec_data.tagged_union
= NULL;
```

server_c_exec_running
For the case where the server is running, and the value of the union member is a **srvrexec_data_t** data type named **running_data** . A **srvrexec_data_t** structure contains the following members:

instance Each instance of a server on a host is identified with a UUID (type **uuid_t**).

posix_pid Each server has a POSIX process ID of type **unsigned32**.

service_t This structure describes each service offered by a server. The **server_t** structure, described earlier, contains an array of these structures. The **service_t** structure contains the following fields:

ifspec An interface specification of type *rpc_if_id_t*, generated by an **idl** compilation of the interface definition representing the service. This data type is described in the **rpc_intro(3rpc)** reference page.

ifname An interface name of type **dced_string_t**.

annotation An annotation about the purpose of the interface (type **dced_string_t**). This field is for user display purposes only.

flags The flag field is of type **unsigned32** and currently has only one bit field defined, **service_c_disabled**. If this flag is set, it indicates that the service is not currently available for the server. Also, the **dced** endpoint mapper will not map an endpoint to a disabled service. Several values are also reserved for vendor-specific use:

service_c_vendor1

service_c_vendor2

service_c_vendor3

service_c_vendor4

entryname The entry name (type **dced_string_t**) is a hint as to where this service appears in the namespace. If the value is NULL, the value in the **entryname** field of the **server_t** structure is used.

objects This is a list of objects supported by the service. The list is of type **uuid_list_t**.

operations This is a list of operation numbers of type **dced_opnum_list_t**. This field is not currently used.

towers This is a list of protocol towers of type **dced_tower_list_t**, specifying the endpoints where this server can be reached.

srvrexec_stop_method_t

The server execution stop method is an enumerated type with one of the following values:

srvrexec_stop_rpc

Stops the running server gracefully by letting the server complete all outstanding remote procedure calls. This causes **dced** to invoke the **rpc_mgmt_server_stop_listening()** routine in that server.

srvrexec_stop_soft

This uses a system-specific mechanism such as the **SIGTERM** signal. It stops the running server with a mechanism that the server can ignore or intercept in order to do application-specific cleanup.

srvrexec_stop_hard

This uses a system-specific mechanism such as the **SIGKILL** signal. It stops the running server immediately with a mechanism that the server cannot intercept.

srvrexec_stop_error
: This uses a system-specific mechanism such as the **SIGABRT** signal. The local operating system captures the server's state before stopping it, and the server can also intercept it.

uuid_list_t
: A list of UUIDs in the following format:

  count An **unsigned32** number representing the number of UUIDs in the list.

  list A pointer to an array of UUIDs where each element is of type **uuid_t**.

Files

dce/dced_base.h

dce/dced.h

dce/dced_data.h

dce/rpctypes.idl

dce/passwd.idl

dce/sec_attr_base.idl

Related Information

Functions: **dced_*** API.

Books: *DCE 1.2.2 Application Development Guide*

DCE_SVC_INTRO

Purpose Introduction to the DCE serviceability macros

Description

The **DCE_SVC_DEFINE_HANDLE** macro is used to create a serviceability handle. This can be useful in a library that has no explicit initialization routine in which a call to **dce_svc_register()** could be added. The remaining macros can be compiled out of production code, or left in to aid diagnostics, depending on whether or not **DCE_DEBUG** (normally found in **dce/dce.h**) is defined.

The DCE Serviceability Macros

The serviceability macros are as follows, listed in alphabetical order:

DCE_SVC_DEBUG()
Used to generate debugging output.

DCE_SVC_DEBUG_ATLEAST()
Can be used to test the debug level of a subcomponent for a specified handle. Tests whether the debug level is at least at the specified level.

DCE_SVC_DEBUG_IS()
Can be used to test the debug level of a subcomponent for a specified handle. Tests for an exact match with the specified level.

DCE_SVC_DEFINE_HANDLE()
Registers a serviceability message table.

DCE_SVC_LOG()
Generates debugging output based on a message defined in an application's **sams** file.

Files

dce/service.idl

dce/dce_svc.h

Related Information

Books: *DCE 1.2.2 Application Development Guide*

dce_assert

Purpose

Inserts program diagnostics

Synopsis

#define
DCE_ASSERT #include <dce/assert.h>

voiddce_assert(
dce_svc_handle_t *handle***,**
int *expression***);**

Parameters

Input

handle A registered serviceability handle.

expression An expression the truth of which is to be tested.

Description

The **dce_assert** macro is used to add runtime ‘‘can’t happen’’ assertions to programs (that is, programming errors). On execution, when *expression* evaluates to 0 (that is, to FALSE), then **dce_svc_printf()** is called with parameters to generate a message identifying the expression, source file and line number. The message is generated with a severity level of **svc_c_sev_fatal**, with the **svc_c_action_abort** flag specified (which will cause the program to abort when the assertion fails and the message is generated). See the **dce_svc_register(3dce)** reference page for more information.

The *handle* parameter should be a registered serviceability handle; it can also be NULL, in which case an internal serviceability handle will be used.

Assertion-checking can be enabled or disabled at compile time. The header file **dce/assert.h** can be included multiple times. If **DCE_ASSERT** is defined before the header

is included, assertion checking is performed. If it is not so defined, then the assertion-checking code is not compiled in. The system default is set in **dce/dce.h**.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_register(3dce)**.

dce_attr_sch_bind

Purpose

Returns an opaque handle to a schema object

Synopsis

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

voiddce_attr_sch_bind(
        dce_attr_component_name_t schema_name,
        dce_bind_auth_info_t * auth_info,
        dce_attr_sch_handle_t * h,
        error_status_t * status);
```

Parameters

Input

schema_name
: A pointer to a value of type **dce_attr_component_name_t** that specifies the name of the schema object to bind to.

auth_info
: A value of type **dce_bind_auth_info_t** that defines the authentication and authorization parameters to use with the binding handle. If set to NULL, the default authentication and authorization parameters are used.

Output

h
: An opaque handle of type **dce_attr_sch_handle_t** to the named schema object for use with **dce_attr_sch** operations.

status
: A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_bind()** routine returns an opaque handle of type **dce_attr_sch_handle_t** to a named schema object. The returned handle can then be used for subsequent **dce_attr_sch** operations performed on the object.

Permissions Required

The **dce_attr_sch_update_entry()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_name

sec_login_s_no_current_context

rpc_s_entry_not_found

rpc_s_no_more_bindings

dce_attr_s_unknown_auth_info_type

dce_attr_s_no_memory

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_bind_free(3dce)**.

dce_attr_sch_bind_free

Purpose

Releases an opaque handle of type **dce_attr_sch_handle_t** to a schema object

Synopsis

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

voiddce_attr_sch_bind_free(
        dce_attr_sch_handle_t * h,
        error_status_t * status);
```

Parameters

Input

h An opaque handle of type **dce_attr_sch_handle_t**.

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_attr_sch_bind_free()** routine releases an opaque handle of type **dce_attr_sch_handle_t** . The handle was returned with the **dce_attr_sch_bind()** routine and used to perform **dce_attr_sch** operations.

Permissions Required

The **dce_attr_sch_bind_free()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_sch.idl
The **idl** file from which **dce/dce_attr_sch.h** was derived.

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

Related Information

Functions: **dce_attr_intro(3dce)**, **dce_attr_sch_bind(3dce)**.

dce_attr_sch_create_entry

Purpose

Creates a schema entry in a schema bound to by a previous **dce_attr_sch_bind()**

Synopsis

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

voiddce_attr_sch_create_entry(
        dce_attr_sch_handle_t h,
        dce_attr_schema_entry_t * schema_entry,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

schema_entry
A pointer to a **dce_attr_schema_entry_t** that contains the schema entry values for the schema in which the entry 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_attr_sch_create_entry()** routine creates schema entries that define attribute types in the schema object bound to by *h*.

Permissions Required

The **dce_attr_sch_create_entry()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)**, **dce_attr_sch_delete_entry(3dce)**, **dce_attr_sch_update(3dce)**.

dce_attr_sch_cursor_alloc

Purpose

Allocates resources to a cursor used with **dce_attr_sch_scan()**

Synopsis

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

voiddce_rgy_attr_cursor_alloc(
        dce_attr_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Output

cursor A pointer to a **dce_attr_cursor_t**.

status A pointer to the completion status. On successful completion, the call returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_cursor_alloc()** routine allocates resources to a cursor used with the **dce_attr_sch_scan()** routine. This routine, which is a local operation, does not initialize *cursor*.

The **dce_attr_sch_cursor_init()** routine, which makes a remote call, allocates and initializes the cursor. In addition, **dce_attr_sch_cursor_init()** returns the total number of entries found in the schema as an output parameter; **dce_attr_sch_cursor_alloc()** does not.

Permissions Required

The **dce_attr_sch_cursor_alloc()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_no_memory

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_cursor_init(3dce)** , **dce_attr_sch_cursor_release(3dce)** , **dce_attr_sch_scan(3dce).**

dce_attr_sch_cursor_init

Purpose

Initializes and allocates a cursor used with **dce_attr_sch_scan()**

Synopsis

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

voiddce_rgy_attr_cursor_init(
        dce_attr_sch_handle_t h,
        unsigned32 * cur_num_entries,
        dce_attr_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

Output

cur_num_entries
A pointer to an unsigned 32-bit integer that specifies the total number of entries contained in the schema at the time of this call.

cursor A pointer to a **dce_attr_cursor_t** that is initialized to the first entry in the the schema.

status A pointer to the completion status. On successful completion, the call returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_cursor_init()** routine initializes and allocates a cursor used with the **dce_attr_sch_scan()** routine. This call makes remote calls to initialize the cursor. To limit the number of remote calls, use the **dce_attr_sch_cursor_alloc()** routine to allocate *cursor*, but not initialize it. If the cursor input to **dce_attr_sch_scan()** has not been initialized, **dce_attr_sch_scan()** routine will initialize it; if it has been initialized, **dce_attr_sch_scan()** advances it.

Unlike the **dce_attr_sch_cursor_alloc()** routine, the **dce_attr_sch_cursor_init()** routine supplies the total number of entries found in the schema as an output parameter.

Permissions Required

None.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

dce_attr_s_no_memory

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_cursor_allocate(3dce), dce_attr_sch_cursor_release(3dce)** , **dce_attr_sch_scan(3dce).**

dce_attr_sch_cursor_release

Purpose

Releases states associated with a cursor that has been allocated with either **dce_attr_sch_cursor_init()** or **dce_attr_sch_cursor_alloc()**

Synopsis

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

voiddce_attr_sch_cursor_init(
        dce_attr_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input/Output

cursor A pointer to a **dce_attr_cursor_t**. As an input parameter, *cursor* must have been initialized to the first entry in a schema. As an output parameter, *cursor* is uninitialized with all resources released.

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_attr_sch_cursor_init()** routine releases the resources allocated to a cursor that has been allocated by either **dce_attr_sch_cursor_init()** or **dce_attr_sch_cursor_alloc()** . This call is a local operation and makes no remote calls.

Permissions Required

None.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_cursor_alloc(3dce)** , **dce_attr_sch_cursor_init(3dce)** , **dce_attr_sch_cursor_reset(3dce)** , **dce_attr_sch_scan(3dce)**.

dce_attr_sch_cursor_reset

Purpose

Resets a cursor that has been allocated with either **dce_attr_sch_cursor_init()** or **dce_attr_sch_cursor_alloc()**

Synopsis

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

voiddce_attr_cursor_reset(
        dce_attr_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input/Output

cursor A pointer to a **dce_attr_cursor_t**. As an input parameter, an initialized *cursor*. As an output parameter, *cursor* is reset to the first attribute in the schema.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_cursor_reset()** routine resets a **dce_attr_cursor_t** that has been allocated by either the **dce_attr_sch_cursor_init()** routine or the **dce_attr_sch_cursor_alloc()** routine. The reset cursor can then be used to process a new **dce_attr_sch_scan** query by reusing the cursor instead of releasing and reallocating it. This is a local operation and makes no remote calls.

Permissions Required

None.

Files

/usr/include/dce/dce_attr_sch.idl
The **idl** file from which **dce/dce_attr_sch.h** was derived.

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

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_cursor_alloc(3dce)** , **dce_attr_sch_cursor_init(3dce)** , **dce_attr_sch_scan(3dce)**.

dce_attr_sch_delete_entry

Purpose

Deletes a schema entry

Synopsis

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

voiddce_attr_sch_delete_entry(
        dce_attr_sch_handle_t h,
        uuid_t * attr_id,
        error_status_t * status);
```

Parameters

Input

h — An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

attr_id — A pointer to a **uuid_t** that identifies the schema entry to be deleted in the schema bound to by **h**.

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_attr_sch_delete_entry()** routine deletes a schema entry. Because this is a radical operation that invalidates any existing attributes of this type on objects dominated by the schema, access to this operation should be severely limited.

Permissions Required

The **dce_attr_sch_delete_entry()** routine requires requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)**, **dce_attr_sch_create_entry(3dce)**, **dce_attr_sch_update_entry(3dce)**.

dce_attr_sch_get_acl_mgrs

Purpose

Retrieves the manager types of the ACLs protecting the objects dominated by a named schema

Synopsis

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

voiddce_attr_sch_get_acl_mgrs(
        dce_attr_sch_handle_t h,
        unsigned32 size_avail,
        unsigned32 * size_used,
        unsigned32 * num_acl_mgr_types,
        uuid_t acl_mgr_types[ ],
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

size_avail An unsigned 32-bit integer containing the allocated length of the *acl_manager_types*[] array.

Output

size_used An unsigned 32-bit integer containing the number of output entries returned in the *acl_mgr_types*[] array.

num_acl_mgr_types
An unsigned 32-bit integer containing the number of types returned in the *acl_mgr_types*[] array. This may be greater than *size_used* if there was not enough space allocated by *size_avail* for all the manager types in the *acl_manager_types*[] array.

acl_mgr_types[]
: An array of the length specified in *size_avail* to contain UUIDs (of type **uuid_t**) identifying the types of ACL managers protecting the target 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_attr_sch_get_acl_mgrs()** routine returns a list of the manager types protecting the schema object identified by *h*.

ACL editors and browsers can use this operation to determine the ACL manager types protecting a selected schema object.

Permissions Required

The **dce_attr_sch_get_acl_mgrs()** routine requires appropriate permissions on the schema object for which the ACL manager types are to be returned. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
: The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_not_implemented

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)**.

dce_attr_sch_lookup_by_id

Purpose

Reads a schema entry identified by UUID

Synopsis

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

voiddce_attr_sch_lookup_by_id(
        dce_attr_sch_handle_t h,
        uuid_t * attr_id,
        dce_attr_schema_entry_t * schema_entry,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

attr_id A pointer to a **uuid_t** that identifies a schema entry.

Output

schema_entry
A **dce_attr_schema_entry_t** that contains an entry identified by *attr_id*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_lookup_by_id()** routine reads a schema entry identified by *attr_id*. This routine is useful for programmatic access.

After a successful call, free the resources allocated by this routine for the *schema_entry* parameter by using the **sec_attr_util_sch_ent_free_ptrs()** routine.

Permissions Required

The **dce_attr_sch_lookup_by_id()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)**, **dce_attr_sch_lookup_by_name(3dce)**, **dce_attr_sch_scan(3dce)**.

dce_attr_sch_lookup_by_name

Purpose

Reads a schema entry identified by name

Synopsis

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

voiddce_attr_sch_lookup_by_name(
        dce_attr_sch_handle_t h,
        char * attr_name,
        dce_attr_schema_entry_t * schema_entry,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

attr_name A pointer to a character string that identifies the schema entry.

Output

schema_entry
A **dce_attr_schema_entry_t** that contains the schema entry identified by *attr_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_attr_sch_lookup_by_name()** routine reads a schema entry identified by name. This routine is useful for use with an interactive editor.

After a successful call, free the resources allocated by this routine for the **attr** parameter by using the **sec_attr_util_inst_free_ptrs()** routine.

Permissions Required

The **dce_attr_sch_lookup_by_name()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)**, **dce_attr_sch_lookup_by_id(3dce)**, **dce_attr_sch_scan(3dce)**.

dce_attr_sch_scan

Purpose

Reads a specified number of schema entries

Synopsis

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

voiddce_attr_sch_scan(
        dce_attr_sch_handle_t h,
        dce_attr_cursor_t * cursor,
        unsigned32 num_to_read,
        unsigned32 * num_read,
        dce_attr_schema_entry_t schema_entries[ ] ,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

num_to_read
An unsigned 32-bit integer specifying the size of the *schema_entries*[] array and the maximum number of entries to be returned.

Input/Output

cursor A pointer to a **dce_attr_cursor_t**. As input *cursor* must be allocated and can be initialized. If *cursor* is not initialized, **dce_attr_sch_scan** will initialize it. As output, *cursor* is positioned at the first schema entry after the returned entries.

Output

num_read A pointer to an unsigned 32-bit integer specifying the number of entries returned in *schema_entries*[].

schema_entries[]
A **dce_attr_schema_entry_t** that contains an array of the returned schema entries.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_attr_sch_scan()** routine reads schema entries. The read begins at the entry at which the input *cursor* is positioned and ends after the number of entries specified in *num_to_read*.

The input *cursor* must have been allocated by either the **dce_attr_sch_cursor_init()** or the **dce_attr_sch_cursor_alloc()** routine. If the input *cursor* is not initialized, **dce_attr_sch_scan()** initializes it; if cursor is initialized, **dce_attr_sch_scan()** simply advances it.

To read all entries in a schema, make successive **dce_attr_sch_scan()** calls. When all entries have been read, the routine returns the message **no_more_entries**.

This routine is useful as a browser.

Permissions Required

The **dce_attr_sch_scan()** routine requires requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

dce_attr_s_bad_cursor

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_cursor_alloc(3dce)** , **dce_attr_sch_cursor_init(3dce)** , **dce_attr_sch_cursor_release(3dce)** .

dce_attr_sch_update_entry

Purpose

Updates a schema entry

Synopsis

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

voiddce_attr_sch_update_entry(
        dce_attr_sch_handle_t h,
        dce_attr_schema_entry_parts_t modify_parts,
        dce_attr_schema_entry_t * schema_entry,
        error_status_t * status);
```

Parameters

Input

h An opaque handle bound to a schema object. Use **dce_attr_sch_bind()** to acquire the handle.

modify_parts
A value of type **dce_attr_schema_entry_parts_t** that identifies the fields in the schema bound to by *h* that can be modified.

schema_entry
A pointer to a **dce_attr_schema_entry_t** that contains the schema entry values for the schema entry to be updated.

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_attr_sch_update_entry()** routine modifies schema entries. Only those schema entry fields set to be modified in the **dce_attr_schema_entry_parts_t** data type can be modified.

Some schema entry components can never be modified. Instead, in order to make any changes to these components, the schema entry must be deleted (which deletes all attribute instances of that type) and recreated. The schema entry components that can never be modified are as follows:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger types
- Comment

Fields that are arrays of structures (such as **acl_mgr_set** and **trig_binding**) are completely replaced by the new input array. This operation cannot be used to add a new element to the existing array.

Permissions Required

The **dce_attr_sch_update_entry()** routine requires appropriate permissions on the schema object. These permissions are managed by the target server.

Files

/usr/include/dce/dce_attr_base.idl
The **idl** file from which **dce/dce_attr_base.h** was derived.

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_attr_s_bad_binding

error_status_ok

Related Information

Functions: **dce_attr_intro(3dce)** , **dce_attr_sch_create_entry(3dce)** , **dce_attr_sch_delete_entry(3dce)** .

dce_cf_binding_entry_from_host

Purpose

Returns the host binding entry name

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_binding_entry_from_host(
        char * hostname,
        char ** entry_name,
        error_status_t * status);
```

Parameters

Input

hostname Specifies the name of the host. Note that host names are case sensitive. If NULL, the configuration file is searched for the host name, and that name, if found, is used.

Output

entry_name The binding entry name associated with the specified host.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_binding_entry_from_host()** routine returns the binding entry name string associated with the *hostname* passed to it. If *hostname* is NULL, the binding entry name associated with the name returned by **dce_cf_get_host_name()** is returned.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation completed successfully.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
: No host name entry in the DCE configuration file.

Related Information

Functions: **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_dced_entry_from_host

Purpose

Returns the **dced** entry name on a host

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_dced_entry_from_host(
        char * hostname,
        char ** entry_name,
        error_status_t * status);
```

Parameters

Input

hostname Specifies the name of the host. Note that host names are case sensitive. If this value is NULL, the value returned by **dce_cf_get_host_name()** is used.

Output

entry_name The **dced** entry name associated with the specified host. Storage for this name is dynamically allocated; release it with **free()** when you no longer need it.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_dced_entry_from_host()** routine returns the name entered into the DCE namespace for a DCE host daemon (**dced**) on the host specified by the *hostname* parameter. If the *hostname* parameter is NULL, the **dced** name associated with the

name returned by **dce_cf_get_host_name()** is returned. The string name is of the form **/.:/hosts/** *hostname*/**config**, and specifies the entry point into the **dced** namespace on the host. This is the location in the DCE namespace at which **dced** stores the objects associated with the host services it provides (the **hostdata**, **srvrconf**, **srvrexec**, **secval**, and **keytab** services, as well as ACL editing). It is also an actual name in the DCE namespace that you can import if you want to create your own RPC binding to **dced**.

You can use the **dced** entry name returned by this routine as input to the **dced_binding_create()** routine, input to **sec_acl_*** routines, or to **rpc_ns_binding_import_** * routines to establish a binding to a **dced** host service.

If using **dced_binding_create()**, you append a service name to the entry returned by this routine. If using **sec_acl_** * routines, you append the service and the object name. If using **rpc_ns_binding_import_***, you use only the entry returned by the routine.

You can also use the returned string to name objects that **dced** maintains, for example, when editing these objects' ACLs with **dcecp**. For example, the string name **/.:/hosts/vineyard/config/srvrconf/dtsd** names the server configuration data for the DTS server on the host **vineyard**.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation completed successfully.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
No host name entry in the DCE configuration file.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**, **dced_binding_create(3dce)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*, *DCE 1.2.2 Command Reference*.

dce_cf_find_name_by_key

Purpose

Returns a string tagged by a character string key

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_find_name_by_key(
        FILE * fp,
        char * key,
        char ** name,
        error_status_t * status);
```

Parameters

Input

fp A file pointer to a correctly formatted text file opened for reading.

key A character string key that will be used to find *name*.

Input/Output

name A pointer to a string (**char ****) in which a string containing the name found will be placed. The name string will be allocated by **malloc()**.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_find_name_by_key()** routine searches a text file for the first occurrence of a string tag identical to the string passed in *key*. The tag string, in order to be found,

must be the first nonwhitespace string on an uncommented line. If the tag string is found, **dce_cf_find_name_by_key()** allocates (by a call to **malloc()**) a buffer for the next string found on the same line as the tag string, copies this second string into the buffer, and returns its address in the *name* input parameter.

The name of the DCE configuration file is in the constant **dce_cf_c_db_name**; in turn, this constant is defined in the header file **<dce_cf.h>**.

Cautions

The memory for a returned name string is allocated by **malloc()**, and must be freed by the original caller of the configuration routine that called **dce_cf_find_name_by_key()**.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation completed succesfully.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
: No match for *key* in the file.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_free_cell_aliases

Purpose

Frees a list of cell name aliases for the local cell

Synopsis

```
#include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_free_cell_aliases(
        char ** cell_alias_list,
        error_status_t * status);
```

Parameters

Input

cell_alias_list
: The address of a cell alias list, which is a null-terminated array of pointers to the cell alias names for the local cell.

Output

status
: Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_free_cell_aliases()** routine frees the list of aliases for the local cell that the **dce_cf_free_cell_aliases()** routine allocated. The routine frees the memory allocated to hold the array of pointers to cell alias string buffers, and also frees the string buffers.

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_cf_st_ok

dce_cf_e_file_open

dce_cf_e_no_mem

dce_cf_e_no_match

Related Information

Functions: **dce_cf_get_cell_aliases(3dce)** , **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**, **dce_cf_same_cell_name(3dce)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*, *DCE 1.2.2 Command Reference*.

dce_cf_get_cell_aliases

Purpose

Returns a list of aliases for the local cell

Synopsis

```
#include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_get_cell_aliases(
        char *** cell_alias_list,
        error_status_t * status);
```

Parameters

Input

None.

Output

cell_alias_list
: The address of a string pointer array. This routine sets this address to point to the address of an allocated null-terminated array of pointers to the cell alias names for the local cell. If no aliases exist, the routine returns NULL in this parameter.

status
: Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_get_cell_aliases()** routine retrieves the local cell's cell name aliases. If cell aliases are found, the routine returns the address of an allocated list of cell alias names in the *cell_alias_list* parameter. If no aliases exist for the cell, the routine returns NULL.

Use the **dce_cf_free_cell_aliases()** routine to free the memory allocated by the **dce_cf_get_cell_aliases()** routine.

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_cf_st_ok

dce_cf_e_file_open

dce_cf_e_no_mem

dce_cf_e_no_match

Related Information

Functions: **dce_cf_free_cell_aliases(3dce)** , **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_same_cell_name(3dce)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*, *DCE 1.2.2 Command Reference*.

dce_cf_get_cell_name

Purpose

Returns the primary name for the local cell

Synopsis

```
#include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_get_cell_name(
        char ** cellname,
        error_status_t * status);
```

Parameters

Input

None.

Output

cellname The address of a string pointer. This pointer will be set by the function to point to an allocated buffer that contains the cell name.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_get_cell_name()** routine retrieves the primary name for the local cell. If the name is found, **dce_cf_get_cell_name()** returns an allocated (by a call to **malloc()**) copy of it in the *cellname* input parameter. Use **free()** to free the allocated copy when you no longer need it.

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_cf_st_ok

dce_cf_e_file_open

dce_cf_e_no_mem

dce_cf_e_no_match

Related Information

Functions: **dce_cf_free_cell_aliases(3dce)** , **dce_cf_get_cell_aliases(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_get_csrgy_filename

Purpose

Returns the pathname of the code set registry file on a host

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_get_csrgy_filename(
        char ** csrgy_filename,
        error_status_t * status);
```

Parameters

Input

None.

Input/Output

csrgy_filename
: The address of a string pointer. This pointer will be set by the function to point to a buffer that contains the pathname to the code set registry file.

Output

status
: Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_get_csrgy_filename()** routine is a DCE function that returns the pathname of a code set registry file that has been created on a given host with the **csrc** utility. DCE RPC routines for code set interoperability use this routine when they need to

locate a host's code set registry file in order to map between unique code set identifiers and their operating system-specific local code set names, or to obtain supported code sets for a client or server. User-written code set interoperability routines can also use the routine.

The **dce_cf_get_csrgy_filename()** routine searches the DCE configuration file for the name of the local host's code set registry file, allocates a buffer for it (by a call to **malloc()**), copies the name into the buffer, and returns its address in the *csrgy_filename* input parameter.

Cautions

The memory for a returned name string is allocated by **malloc()**, and must be freed by the caller of **dce_cf_get_csrgy_filename()**.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation successfully completed.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

Related Information

Functions: **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**, **rpc_rgy_get_codesets(3rpc)**.

Commands: **csrc(8dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_get_host_name

Purpose

Returns the host name relative to the local cell root

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_get_host_name(
        char ** hostname,
        error_status_t * status);
```

Parameters

Input

None.

Input/Output

hostname The address of a string pointer. This pointer will be set by the function to point to a buffer that contains the host name.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_get_host_name()** routine searches the DCE configuration file for the local host's name relative to the local cell's root. If the name is found, **dce_cf_get_host_name()** allocates (by a call to **malloc()**) a buffer for it, copies the name into the buffer, and returns its address in the *hostname* input parameter.

Cautions

The memory for a returned name string is allocated by **malloc()**, and must be freed by the caller of **dce_cf_get_host_name()**.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation successfully completed.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
: No host name entry in the DCE configuration file.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_prin_name_from_host

Purpose

Returns the host's principal name

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_prin_name_from_host(
        char * hostname,
        char ** prin_name,
        error_status_t * status);
```

Parameters

Input

hostname The name of the host. Note that host names are case sensitive. If NULL, the configuration file is searched for the host name, and that name, if found, is used.

Output

prin_name The principal name associated with the specified host.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_prin_name_from_host()** routine returns the principal name associated with the *hostname* passed to it. If *hostname* is NULL, **dce_cf_prin_name_from_host()** returns the principal name associated with the name returned by **dce_cf_get_host_name()**.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation completed successfully.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
: No host name entry in the DCE configuration file.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_profile_entry_from_host

Purpose

Returns the host profile entry

Synopsis

```
 #include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_profile_entry_from_host(
        char * hostname,
        char ** prof_name,
        error_status_t * status);
```

Parameters

Input

hostname Specifies the name of the host. Note that host names are case sensitive. If NULL, the configuration file is searched for the host name, and that name, if found, is used.

Output

prof_name The profile entry associated with the specified host.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_cf_profile_entry_from_host()** routine returns the profile entry string associated with the *hostname* passed to it. If *hostname* is NULL, the profile entry associated with the name returned by **dce_cf_get_host_name()** is returned.

Files

dcelocal/**dce_cf.db**
: The machine's local DCE configuration file (where *dcelocal* is usually something like **/opt/dcelocal**).

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_cf_st_ok
: Operation completed successfully.

dce_cf_e_file_open
: File open error.

dce_cf_e_no_mem
: No memory available.

dce_cf_e_no_match
: No host name entry in the DCE configuration file.

Related Information

Functions: **dce_cf_binding_entry_from_host(3dce)**, **dce_cf_find_name_by_key(3dce)**, **dce_cf_get_cell_name(3dce)**, **dce_cf_get_host_name(3dce)**, **dce_cf_prin_name_from_host(3dce)**.

Books: *DCE 1.2.2 Administration Guide*.

dce_cf_same_cell_name

Purpose

Indicates whether or not two cell names refer to the same cell

Synopsis

```
#include <stdio.h>
#include <dce/dce_cf.h>

voiddce_cf_same_cell_name(
        char * cell_name1,
        char * cell_name2,
        boolean result,
        error_status_t * status);
```

Parameters

Input

cell_name1 A character string that specifies the name of a cell.

cell_name2 A character string that specifies the name of a cell to compare with *cell_name1*. If this value is NULL, the routine determines whether or not the cell name specified in *cell_name1* is the name of the local cell.

Output

result A boolean value that indicates whether or not the specified cell names match, when two cell names are given, and indicates whether or not the specified cell name is the name of the local cell, when only one cell name is given. A value of TRUE indicates that the cell names refer to the same cell.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

dce_cf_same_cell_name(3dce)

Description

The **dce_cf_same_cell_name ()** routine, when given the names of two cells as input parameters, compares the cell names to determine whether or not they refer to the same call. The *result* parameter is set to TRUE if they do, and to FALSE if they do not.

If only one cell name is specified as an input parameter, the **dce_cf_same_cell_name()** routine determines whether or not the specified cell name is the same as the local cell's primary name (which it retrieves by calling **dce_cf_get_cell_name()**). You can use the routine in this way to determine whether a given cell name is the primary name of your local cell.

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_cf_st_ok

dce_cf_e_no_match

Related Information

Functions: **dce_cf_free_cell_aliases(3dce)** , **dce_cf_get_cell_aliases(3dce)**, **dce_cf_get_cell_name(3dce)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components, DCE 1.2.2 Command Reference*.

dce_db_close

Purpose

Closes an open backing store

Synopsis

#include <dce/dce.h> #include <dce/dbif.h>

voiddce_db_close(
dce_db_handle_t * *handle***,**
error_status_t * *status***);**

Parameters

Input

handle A handle identifying the backing store to be closed.

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_db_close()** routine closes a backing store that was opened by **dce_db_open()**. It also frees the storage used by the handle, and sets the handle's value to NULL.

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_db_open(3dce)**.

dce_db_delete

Purpose

Deletes an item from a backing store

Synopsis

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

voiddce_db_delete(
        dce_db_handle_t handle,
        void * key,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A pointer to a string or UUID that is the key to the item in the backing store. The datatype of *key* must match the key method that was selected in the *flags* parameter to **dce_db_open()** when the backing store was created.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error code.

Description

The **dce_db_delete()** routine deletes an item from the backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It is a general deletion routine, interpreting the *key* parameter according to the type of index with which the backing store was created.

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_del_failed
The deletion did not occur. The global variable **errno** may indicate further information about the error.

db_s_bad_index_type
The *key*'s type is wrong, or the backing store is not by name or by UUID.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Deletion is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_delete_by_name(3dce)**, **dce_db_delete_by_uuid(3dce)**, **dce_db_open(3dce)**.

dce_db_delete_by_name

Purpose

Deletes an item from a string-indexed backing store

Synopsis

#include <dce/dce.h> #include <dce/dbif.h>

void dce_db_delete_by_name(
dce_db_handle_t *handle***,**
char * *key***,**
error_status_t * *status***);**

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A NULL-terminated string that is the key to the item in the backing store.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error code.

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_del_failed
The deletion did not occur. The global variable **errno** may indicate further information about the error.

db_s_bad_index_type
The backing store is not indexed by name.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Deletion is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_delete(3dce)**, **dce_db_delete_by_uuid(3dce)**, **dce_db_open(3dce)**.

dce_db_delete_by_uuid

Purpose

Deletes an item from a UUID-indexed backing store

Synopsis

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

voiddce_db_delete_by_uuid(
        dce_db_handle_t handle,
        uuid_t * key,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A pointer to a UUID that is the key to the item in the backing store.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error code.

Description

The **dce_db_delete_by_uuid()** routine deletes an item from the backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It is a specialized deletion routine for backing stores that are indexed by UUID, as selected by the **db_c_index_by_uuid** bit in the *flags* parameter to **dce_db_open()** when the backing store was created.

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_del_failed
The deletion did not occur. The global variable **errno** may indicate further information about the error.

db_s_bad_index_type
The backing store is not indexed by UUID.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Deletion is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_delete(3dce)**, **dce_db_delete_by_name(3dce)**, **dce_db_open(3dce)**.

dce_db_fetch

Purpose

Retrieves data from a backing store

Synopsis

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

voiddce_db_fetch(
        dce_db_handle_t handle,
        void * key,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A string or UUID that is the key to the item in the backing store. The datatype of *key* must match the key method that was selected in the *flags* parameter to **dce_db_open()** when the backing store was created.

Output

data A pointer to the returned data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_fetch()** routine retrieves data from the backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It is a general

retrieval routine, interpreting the *key* parameter according to the type of index with which the backing store was created.

The *data* parameter is shown as a pointer to an arbitrary data type. In actual use it will be the address of the backing-store-specific data type.

Notes

After calling **dce_db_fetch()**, it may be necessary to free some memory, if the call was made outside of an RPC, on the server side. This is done by calling **rpc_sm_client_free()**. (Inside an RPC the memory is allocated through **rpc_sm_allocate()**, and is automatically freed.)

Programs that call **dce_db_fetch()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_fetch()** must do **rpc_sm_allocate()**, but in the server side of an RPC, this is already done.

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 circumstance is not necessarily an error.)

db_s_bad_index_type
: The *key*'s type is wrong, or else the backing store is not by name or by UUID.

error_status_ok
: The call was successful.

Related Information

Functions: **dce_db_fetch_by_name(3dce)**, **dce_db_fetch_by_uuid(3dce)**, **dce_db_free(3dce)**, **dce_db_open(3dce)**.

dce_db_fetch_by_name

Purpose

Retrieves data from a string-indexed backing store

Synopsis

#include <dce/dce.h> #include <dce/dbif.h>

void dce_db_fetch_by_name(
dce_db_handle_t *handle***,**
char * *key***,**
void * *data***,**
error_status_t * *status***);**

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A null-terminated string that is the key to the item in the backing store.

Output

data A pointer to the returned data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_fetch_by_name()** routine retrieves data from the string-indexed backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It is a specialized retrieval routine for backing stores that are indexed by string, as selected by the **db_c_index_by_name** bit in the *flags* parameter to **dce_db_open()** when the backing store was created.

The *data* parameter is shown as a pointer to an arbitrary data type. In actual use it will be the address of the backing-store-specific data type.

Notes

After calling **dce_db_fetch_by_name()**, it may be necessary to free some memory, if the call was made outside of an RPC, on the server side. This is done by calling **rpc_sm_client_free()**. (Inside an RPC the memory is allocated through **rpc_sm_allocate()**, and is automatically freed.)

Programs that call **dce_db_fetch_by_name()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_fetch_by_name()** must do **rpc_sm_allocate()**, but in the server side of an RPC, this is already done.

Examples

This example shows the use of the user-defined data type as the *data* parameter.

```
    extern dce_db_handle_t  db_h;
uuid_t                  key_uuid;
my_data_type_t          my_data;
error_status_t          status;
/* set key_uuid = xxx; */
dce_db_fetch_by_name(db_h, &key_uuid, &my_data, &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.

db_s_key_not_found
: The specified key was not found in the backing store. (This circumstance is not necessarily an error.)

db_s_bad_index_type
The backing store is not indexed by name.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_fetch(3dce)**, **dce_db_fetch_by_uuid(3dce)**, **dce_db_free(3dce)**, **dce_db_open(3dce)**.

dce_db_fetch_by_uuid

Purpose

Retrieves data from a UUID-indexed backing store

Synopsis

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

void dce_db_fetch_by_uuid(
        dce_db_handle_t handle,
        uuid_t * key,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A UUID that is the key to the item in the backing store.

Output

data A pointer to the returned data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_fetch_by_uuid()** routine retrieves data from the UUID-indexed backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It is a specialized retrieval routine for backing stores that are indexed by UUID, as selected by the **db_c_index_by_uuid** bit in the *flags* parameter to **dce_db_open()** when the backing store was created.

The *data* parameter is shown as a pointer to an arbitrary data type. In actual use it will be the address of the backing-store-specific data type.

Notes

After calling **dce_db_fetch_by_uuid()**, it may be necessary to free some memory, if the call was made outside of an RPC, on the server side. This is done by calling **rpc_sm_client_free()**. (Inside an RPC the memory is allocated through **rpc_sm_allocate()**, and is automatically freed.)

Programs that call **dce_db_fetch_by_uuid()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_fetch_by_uuid()** must do **rpc_sm_allocate()**, but in the server side of an RPC, this is already done.

Examples

This example shows the use of the user-defined data type as the *data* parameter.

```
    extern dce_db_handle_t  db_h;
uuid_t                  key_uuid;
my_data_type_t          my_data;
error_status_t          status;
/* set key_uuid = xxx; */
dce_db_fetch_by_uuid(db_h, &key_uuid, &my_data, &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.

db_s_key_not_found
: The specified key was not found in the backing store. (This circumstance is not necessarily an error.)

db_s_bad_index_type
The backing store is not indexed by UUID.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_fetch(3dce)**, **dce_db_fetch_by_name(3dce)**, **dce_db_free(3dce)**, **dce_db_open(3dce)**.

dce_db_free

Purpose

Releases the data supplied from a backing store

Synopsis

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

voiddce_db_free(
        dce_db_handle_t handle,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

data The data area to be released.

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_db_free()** routine is designed to free the data area previously returned via a call to any of the routines **dce_db_fetch()**, **dce_db_fetch_by_name()**, or **dce_db_fetch_by_uuid()**.

Notes

In the current implementation, the **dce_db_free()** routine does not perform any action. For servers that execute properly, this is of little consequence because their allocated memory is automatically cleaned up when a remote procedure call finishes. For completeness, and for compatibility with future releases, the use of **dce_db_free()** is recommended.

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_db_fetch(3dce)**, **dce_db_fetch_by_name(3dce)**, **dce_db_fetch_by_uuid(3dce)**.

dce_db_header_fetch

Purpose

Retrieves the header from a backing store

Synopsis

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

voiddce_db_header_fetch(
        dce_db_handle_t handle,
        void * key,
        dce_db_header_t * hdr,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A string or UUID that is the backing store key.

Output

hdr A pointer to a caller-supplied header structure to be filled in by the library.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_header_fetch()** routine returns a pointer to a copy of the header of the object in the backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. The caller must free the copy's storage. It was allocated

(as with other fetch routines) through **rpc_ss_alloc()**. The *key* parameter is interpreted according to the type of index with which the backing store was created.

The *hdr* parameter is shown as a pointer to an arbitrary data type. In actual use it will be the address of the backing-store-specific data 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.

db_s_key_not_found
The key was not found in the backing store.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_fetch(3dce)**, **dce_db_std_header_init(3dce)**.

dce_db_inq_count

Purpose

Returns the number of items in a backing store

Synopsis

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

voiddce_db_inq_count(
        dce_db_handle_t handle,
        unsigned32 * count,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

count A pointer to the number of items in the backing store.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_inq_count()** routine returns the number of items in the backing store that is identified by the *handle* parameter, which was obtained from **dce_db_open()**. It performs identically on backing stores that are indexed by UUID and those that are indexed by string. The count of items can be helpful when iterating through a backing store.

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_iter_not_allowed
: The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Determining the count is not allowed during iteration.

error_status_ok
: The call was successful.

Related Information

Functions: **dce_db_iter_next(3dce)**.

dce_db_iter_done

Purpose

Frees the state associated with iteration

Synopsis

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

voiddce_db_iter_done(
        dce_db_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **dce_db_iter_done()** routine frees the state that permits iteration. It should be called after an iteration through a backing store is finished.

The iteration state is established by **dce_db_iter_start()**. The routines for performing iteration over the items are **dce_db_iter_next()**, **dce_db_iter_next_by_name()**, and **dce_db_iter_next_by_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.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_iter_next(3dce)**, **dce_db_iter_next_by_name(3dce)**, **dce_db_iter_next_by_uuid(3dce)**, **dce_db_iter_start(3dce)**.

dce_db_iter_next

Purpose

During iteration, returns the next key from a backing store

Synopsis

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

voiddce_db_iter_next(
        dce_db_handle_t handle,
        void ** key,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

key A pointer to the string or UUID that is the key to the item in the backing store.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_iter_next()** routine retrieves the next key from the backing store that is identified by the *handle* parameter. An iterator established by the **dce_db_iter_start()** routine maintains the identity of the current key. Use one of the **dce_db_fetch()** routines to retrieve the actual data.

The iteration functions scan sequentially through a backing store, in no particular order. The **dce_db_iter_start()** routine initialized the process, a **dce_db_iter_next()** routine retrieves successive keys, for which the data can be retrieved with **dce_db_fetch()**, and the **dce_db_iter_done()** routine finishes the process. The iteration can also use the **dce_db_iter_next_by_name()** and **dce_db_iter_next_by_uuid()** routines; the fetching can use the **dce_db_fetch_by_name()** and **dce_db_fetch_by_uuid()** routines.

The iteration routine returns a pointer to a private space associated with the handle. Each call to the iteration routine reuses the space, instead of using allocated space.

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_no_more
: All the keys in the backing store have been accessed; there are no more iterations remaining to be done.

error_status_ok
: The call was successful.

Related Information

Functions: **dce_db_fetch(3dce)**, **dce_db_fetch_by_name(3dce)**, **dce_db_fetch_by_uuid(3dce)**, **dce_db_iter_done(3dce)**, **dce_db_iter_next_by_name(3dce)**, **dce_db_iter_next_by_uuid(3dce)**, **dce_db_iter_start(3dce)**.

dce_db_iter_next_by_name

Purpose

During iteration, returns the next key from a backing store indexed by string

Synopsis

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

voiddce_db_iter_next_by_name(
        dce_db_handle_t handle,
        char ** key,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

key The string that is the key to the item in the backing store.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_iter_next_by_name()** routine retrieves the next key from the backing store that is identified by the *handle* parameter. An iterator established by the **dce_db_iter_start()** routine maintains the identity of the current key. Use the **dce_db_fetch_by_name()** routine to retrieve the actual data.

This iteration routine is the same as **dce_db_iter_next()**, except that it only works with backing stores indexed by name, and returns an error if the backing store index is the wrong type.

The iteration routine returns a pointer to a private space associated with the handle. Each call to the iteration routine reuses the space, instead of using allocated space.

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_no_more
: All the keys in the backing store have been accessed; there are no more iterations remaining to be done.

error_status_ok
: The call was successful.

Related Information

Functions: **dce_db_fetch_by_uuid(3dce)**, **dce_db_iter_done(3dce)**, **dce_db_iter_next(3dce)**, **dce_db_iter_next_by_uuid(3dce)**, **dce_db_iter_start(3dce)**.

dce_db_iter_next_by_uuid

Purpose

During iteration, returns the next key from a backing store indexed by UUID

Synopsis

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

voiddce_db_iter_next_by_uuid(
        dce_db_handle_t handle,
        uuid_t ** key,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

key The UUID that is the key to the item in the backing store.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_iter_next_by_uuid()** routine retrieves the next key from the backing store that is identified by the *handle* parameter. An iterator established by the **dce_db_iter_start()** routine maintains the identity of the current key. Use the **dce_db_fetch_by_uuid()** routine to retrieve the actual data.

This iteration routine is the same as **dce_db_iter_next()**, except that it only works with backing stores indexed by UUID, and returns an error if the backing store index is the wrong type.

The iteration routine returns a pointer to a private space associated with the handle. Each call to the iteration routine reuses the space, instead of using allocated space.

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_db_iter_done(3dce)**, **dce_db_iter_next(3dce)**, **dce_db_iter_next_by_name(3dce)**, **dce_db_iter_start(3dce)**.

dce_db_iter_start

Purpose

Prepares a backing store for iteration

Synopsis

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

voiddce_db_iter_start(
        dce_db_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **dce_db_iter_start()** routine prepares the backing store that is identified by the *handle* parameter for iterative retrieval of all its keys in succession.

A given handle can support only a single instance of iteration at one time.

To avoid the possibility that another thread will write to the backing store during an iteration, always use the **dce_db_lock()** routine before calling **dce_db_iter_start()**.

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_iter_not_allowed
: The function was called while an iteration was already in progress. The concept of nested iterations is not supported.

error_status_ok
: The call was successful.

Related Information

Functions: **dce_db_iter_done(3dce)**, **dce_db_iter_next(3dce)**, **dce_db_iter_next_by_name(3dce)**, **dce_db_iter_next_by_uuid(3dce)**, **dce_db_lock(3dce)**, **dce_db_open(3dce)**, **dce_db_unlock(3dce)**.

dce_db_lock

Purpose

Applies an advisory lock on a backing store

Synopsis

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

voiddce_db_lock(
        dce_db_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

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_db_lock()** routine acquires the lock associated with the handle.

There is an advisory lock associated with each handle. The routines for storing and deleting backing stores apply the lock before updating a backing store. This routine provides a means to apply the lock for other purposes, such as iteration.

Advisory locks allow cooperating threads to perform consistent operations on backing stores, but do not guarantee consistency; that is, threads may still access backing stores without using advisory locks, possibly resulting in inconsistencies.

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_already_locked
An attempt was made to lock a backing store, but it was already locked.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_delete(3dce)**, **dce_db_delete_by_name(3dce)**, **dce_db_delete_by_uuid(3dce)**, **dce_db_store(3dce)**, **dce_db_store_by_name(3dce)**, **dce_db_store_by_uuid(3dce)**, **dce_db_unlock(3dce)**.

dce_db_open

Purpose

Opens an existing backing store or creates a new one

Synopsis

#include <dce/dce.h> #include <dce/dbif.h>

voiddce_db_open(
const char * *name***,**
const char * *backend_type***,**
unsigned32 *flags***,**
dce_db_convert_func_t *convert***,**
dce_db_handle_t * *handle***,**
error_status_t * *status***);**

Parameters

Input

name The filename of the backing store to be opened or created.

backend_type
Either of the strings, **bsd4.4-hash** or **bsd4.4-btree**, or a null pointer, which defaults to hash. This parameter specifies the backing store backend type for licensees adding multiple backends.

flags The manner of opening, as specified by any of the following bits:

db_c_index_by_name
The backing store is to be indexed by name. Either this or **db_c_index_by_uuid**, but not both, must be selected.

db_c_index_by_uuid
The backing store is to be indexed by UUID. Either this or **db_c_index_by_name**, but not both, must be selected.

db_c_std_header
The first field of each item (which is defined as a union in **dce_db_header_t**) is the standard backing store header, with the case **dce_db_header_std** selected. The selection for header cannot have both **db_c_std_header** and **db_c_acl_uuid_header**. If neither header flag is specified, no header is used.

db_c_acl_uuid_header
The first field of each item (the union) is an ACL UUID, with the case **dce_db_header_acl_uuid** selected. The selection for header cannot have both **db_c_std_header** and **db_c_acl_uuid_header**. If neither header flag is specified, no header is used.

db_c_readonly
An existing backing store is to be opened in read-only mode. Read/write is the default.

db_c_create
Creates an empty backing store if one of the given name does not already exist. It is an error to try to create an existing backing store.

convert The function, generated by the IDL compiler, that is called to perform serialization.

Output

handle A pointer to a handle that identifies the backing store being used.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_db_open()** routine opens the specified backing store. The *flags* parameter must specify whether the backing store is to be indexed by name or by UUID. If all of a server's objects have entries in the CDS namespace, then it is probably best to use a UUID index. If the server provides a junction or another name-based lookup operation, then it is probably best to use a name index.

The IDL code in **/usr/include/dce/database.idl** defines the backing store header (selected by the *flags* parameter) that is placed on each item, the possible header types, and the form of the function for serializing headers.

Notes

Backing stores are also called databases. For instance, the associated IDL header is **dce/database.idl**, and the name of the backing store routines begin with **dce_db_**. Nevertheless, backing stores are not databases in the conventional sense, and have no support for SQL or for any other query system.

Examples

Standardized use of the backing store library is encouraged. The following is the skeleton IDL interface for a server's backing store:

```
interface XXX_db
{
import "dce/database.idl";
    typedef XXX_data_s_t {
dce_db_header_t header;
/* server-specific data */
} XXX_data_t;
    void XXX_data_convert(
[in] handle_t h,
[in, out] XXX_data_t *data,
[out] error_status_t *st
);
}
```

This interface should be compiled with the following ACF:

```
interface XXX_db
{
[encode, decode] XXX_data_convert();
```

```
}
```

A typical call to **dce_db_open()**, using the preceding IDL example, follows:

```
dce_db_open("XXX_db", NULL,
db_c_std_header | db_c_index_by_uuid,
(dce_db_convert_func_t)XXX_data_convert,
&handle, &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.

db_s_bad_index_type
The index type in *flags* is specified neither by name nor by UUID, or else it is specified as both.

db_s_bad_header_type
The header type in *flags* is specified as both standard header and ACL header.

db_s_index_type_mismatch
An existing backing store was opened with the wrong index type.

db_s_open_already_exists
The backing store file specified for creation already exists.

db_s_no_name_specified
No filename is specified.

db_s_open_failed_eacces
The server does not have permission to open the backing store file.

db_s_open_failed_enoent
The specified directory or backing store file was not found.

db_s_open_failed
The underlying database-open procedure failed. The global variable **errno** may provide more specific information.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_close(3dce)**.

dce_db_std_header_init

Purpose

Initializes a standard backing store header

Synopsis

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

voiddce_db_std_header_init(
        dce_db_handle_t handle,
        dce_db_header_t * hdr,
        uuid_t * uuid,
        uuid_t * acl_uuid,
        uuid_t * def_object_acl,
        uuid_t * def_container_acl,
        unsigned32 ref_count,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

hdr Pointer to the object header part of the users' structure.

uuid The UUID to be placed into the header. Can be NULL.

acl_uuid The UUID of the ACL protecting this object, to be placed into the header. Can be NULL.

def_object_acl
The UUID of the default object ACL, to be placed into the header. Can be NULL.

def_container_acl
The UUID of the default container ACL, to be placed into the header. Can be NULL.

ref_count The reference count to be placed into the header.

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_db_std_header_init()** routine initializes the fields of the standard header for a data object whose backing store is identified by the handle parameter. The fields are only set in memory and should be stored to the backing store by one of the store routines. The handle was obtained from **dce_db_open()**, which must have been called with the **db_c_std_header** flag.

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_bad_header_type
The header type is not **dce_db_header_std**.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_header_fetch(3dce)**.

dce_db_store

Purpose

Stores data into a backing store

Synopsis

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

voiddce_db_store(
        dce_db_handle_t handle,
        void * key,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A string or UUID that is the backing store key. The datatype of *key* must match the key method that was selected in the *flags* parameter to **dce_db_open()** when the backing store was created.

data A pointer to the data structure to be stored.

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_db_store()** routine stores the data structure pointed to by *data* into the backing store. The conversion function that was specified in the call to **dce_db_open()** serializes the structure so that it can be written to disk.

If the *key* value is the same as a key already stored, the new *data* replaces the previously stored data associated with that key.

Notes

Because the **dce_db_store()** routine uses the encoding services, and they in turn use **rpc_sm_allocate()**, all programs that call **dce_db_store()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_store()** must do **rpc_sm_enable_allocate()**, but in the server side of an RPC, this is already done.

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_bad_index_type
The *key*'s type is wrong, or else the backing store is not by name or by UUID.

db_s_readonly
The backing store was opened with the **db_c_readonly** flag, and cannot be written to.

db_s_store_failed
The data could not be stored into the backing store for some reason. The global variable **errno** may contain more information about the error.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Storing is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_fetch(3dce)**, **dce_db_open(3dce)**, **dce_db_store_by_name(3dce)**, **dce_db_store_by_uuid(3dce)**.

dce_db_store_by_name

Purpose

Stores data into a string-indexed backing store

Synopsis

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

voiddce_db_store_by_name(
        dce_db_handle_t handle,
        char * key,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A null-terminated string that is the backing store key.

data A pointer to the data structure to be stored.

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_db_store_by_name()** routine stores the data structure pointed to by *data* into the backing store. The conversion function that was specified in the call to **dce_db_open()** serializes the structure so that it can be written to disk.

This routine is specialized for storage into backing stores that are indexed by string, as selected by the **db_c_index_by_name** bit in the *flags* parameter to **dce_db_open()** when the backing store was created.

If the *key* value is the same as a key already stored, the new *data* replaces the previously stored data associated with that key.

Notes

Because the **dce_db_store_by_name()** routine uses the encoding services, and they in turn use **rpc_sm_allocate()**, all programs that call **dce_db_store_by_name()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_store_by_name()** must do **rpc_sm_enable_allocate()**, but in the server side of an RPC, this is already done.

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_bad_index_type
The backing store is not indexed by name.

db_s_readonly
The backing store was opened with the **db_c_readonly** flag, and cannot be written to.

db_s_store_failed
The data could not be stored into the backing store for some reason. The global variable **errno** may contain more information about the error.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Storing is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_open(3dce)**, **dce_db_store(3dce)**, **dce_db_store_by_uuid(3dce)**.

dce_db_store_by_uuid

Purpose

Stores data into a UUID-indexed backing store

Synopsis

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

voiddce_db_store_by_uuid(
        dce_db_handle_t handle,
        uuid_t * key,
        void * data,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

key A UUID that is the backing store key.

data A pointer to the data structure to be stored.

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_db_store_by_uuid()** routine stores the data structure pointed to by *data* into the backing store. The conversion function that was specified in the call to **dce_db_open()** serializes the structure so that it can be written to disk.

This routine is specialized for storage into backing stores that are indexed by UUID, as selected by the **db_c_index_by_uuid** bit in the *flags* parameter to **dce_db_open()** when the backing store was created.

If the *key* value is the same as a key already stored, the new *data* replaces the previously stored data associated with that key.

Notes

Because the **dce_db_store_by_uuid()** routine uses the encoding services, and they in turn use **rpc_sm_allocate()**, all programs that call **dce_db_store_by_uuid()** outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first. Indeed, every thread that calls **dce_db_store_by_uuid()** must do **rpc_sm_enable_allocate()**, but in the server side of an RPC, this is already done.

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_bad_index_type
The backing store is not indexed by UUID.

db_s_readonly
The backing store was opened with the **db_c_readonly** flag, and cannot be written to.

db_s_store_failed
The data could not be stored into the backing store for some reason. The global variable **errno** may contain more information about the error.

db_s_iter_not_allowed
The function was called while an iteration, begun by **dce_db_iter_start()**, was in progress. Storing is not allowed during iteration.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_open(3dce)**, **dce_db_store(3dce)**, **dce_db_store_by_name(3dce)**.

dce_db_unlock

Purpose

Releases the backing store lock

Synopsis

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

voiddce_db_unlock(
        dce_db_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle A handle, returned from **dce_db_open()**, that identifies the backing store being used.

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_db_unlock()** routine releases the lock associated with the 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.

db_s_not_locked
An attempt was made to unlock a backing store, but it was not locked.

error_status_ok
The call was successful.

Related Information

Functions: **dce_db_lock(3dce)**.

dce_error_inq_text

Purpose

Retrieves message text associated with a DCE error code

Synopsis

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

void dce_error_inq_text(
        error_status_t status_to_convert,
        dce_error_string_t error_text,
        int * status);
```

Parameters

Input

status_to_convert
: DCE status code for which text message is to be retrieved.

Output

error_text
: The message text associated with the *status_to_convert*.

status
: Returns the status code from this operation. The status code is set to 0 on success, and to -1 on failure.

Description

The **dce_error_inq_text()** routine retrieves from the installed DCE component message catalogs the message text associated with an error status code returned by a DCE library routine.

All DCE message texts are assigned a unique 32-bit message ID. The special value of all-bits-zero is reserved to indicate success.

The **dce_error_inq_text()** routine uses the message ID as a series of indices into the correct DCE component's message catalog; the text found by this indexing is the

message that explains the status code that was returned by the DCE or DCE application routine.

All messages for a given component are stored in a single message catalog generated by the **sams** utility when the component is built. (The messages may also be compiled into the component code, rendering the successful retrieval of message text independent of whether or not the message catalogs were correctly installed.)

If the user sets their **LANG** variable and has the correct message catalog files installed, the user can receive translated messages. That is, the text string returned by **dce_error_inq_text()** is dependant on the current locale.

Examples

The following code fragment shows how **dce_error_inq_text()** can be used to retrieve the message text describing the status code returned by a DCE RPC library routine:

```
dce_error_string_t error_string;
error_status_t status;
int print_status;
rpc_server_register_if(application_v1_0_s_ifspec, &type_uuid,
(rpc_mgr_epv_t)&manager_epv, &status);
if (status != rpc_s_ok) {
dce_error_inq_text(status, error_string, &print_status);
fprintf(stderr," Server: %s: %s\n", caller, error_string);
}
```

dce_msg_cat_close

Purpose

DCE message catalog close routine

Synopsis

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

voiddce_msg_cat_close(
        dce_msg_cat_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle The handle returned by **dce_msg_cat_open()** to the catalog that is to be closed.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_cat_close()** routine closes the message catalog which was opened with **dce_msg_cat_open()**. On error, it fills in *status* with an error code.

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_cat_get_msg(3dce)**, **dce_msg_cat_open(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_cat_msg(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_cat_get_msg

Purpose

DCE message text retrieval routine

Synopsis

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

unsigned char * dce_msg_cat_get_msg(
        dce_msg_cat_handle_t handle,
        unsigned32 message,
        error_status_t * status);
```

Parameters

Input

message The ID of the message to be retrieved.

handle A handle returned by **dce_msg_cat_open()** to an opened message catalog.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

Once the catalog has been opened with the **dce_msg_cat_open()** routine, the **dce_msg_cat_get_msg()** routine can be used to retrieve the text for a specified *message* (which is a 32-bit DCE message ID as described in **dce_error_inq_text(3dce)**). The space allocated for the message should not be freed. The output pointer is useable until a call to the **dce_msg_cat_get_msg()** or **dce_msg_cat_close()** routine. If the specified

message cannot be found in the catalog, the routine returns a NULL and fills in *status* with the appropriate error code.

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_cat_close(3dce)**, **dce_msg_cat_open(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_cat_msg(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_cat_open

Purpose

DCE message catalog open routine

Synopsis

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

dce_msg_cat_handle_tdce_msg_cat_open(
        unsigned32 message_ID,
        error_status_t * status);
```

Parameters

Input

message_ID The ID of the message to be retrieved.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_cat_open()** routine opens the message catalog that contains the specified *message_ID*. It returns a handle that can be used in subsequent calls to **dce_msg_cat_get_msg()**. On error, it returns NULL and fills in *status* with an appropriate error code.

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_cat_close(3dce)**, **dce_msg_cat_get_msg(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_cat_msg(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_define_msg_table

Purpose

Adds a message table to in-memory table

Synopsis

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

void dce_msg_define_msg_table(
        dce_msg_table_t * table,
        unsigned32 count,
        error_status_t * status);
```

Parameters

Input

table A message table structure (defined in a header file generated by **sams** during compilation (see the **EXAMPLES** section).

count The number of elements contained in the table.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

All messages for a given component are stored in a single message catalog generated by the **sams** utility when the component (application) is built.

However, the messages may also be compiled directly into the component code, thus rendering the successful retrieval of message text independent of whether or not the message catalogs were correctly installed. Generation of in-memory message tables is specified by the **incatalog** flag in the **sams** file in which the message text is defined

(see **sams(1dce)** for more information on **sams** files). If the messages have been generated at compile time with this option specified, the **dce_msg_define_msg_table()** routine can be called by the application to register an in-memory table containing the messages.

The *table* parameter to the call should identify a message table structure defined in a header file generated by **sams** during compilation (see the **EXAMPLES** section). The *count* parameter specifies the number of elements contained in the table. If an error is detected during the call, the routine will return an appropriate error code in the *status* parameter.

Examples

The following code fragment shows how an application (whose serviceability component name is *app*) would set up an in-memory message table:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include "dce app msg.h"   /* defines app_msg_table */
error_status_t status;
```

The following call adds the message table to the in-memory table. Note that you must include **<dce/dce_msg.h>**. You also have to link in **dce*app*msg.o** and **dce*app*svc.o** (object files produced by compiling **sams**-generated **.c** files), which contain the code for the messages and the table, respectively.

```
dce_msg_define_msg_table(app_msg_table,
sizeof(app_msg_table) / sizeof(app_msg_table[0]),
&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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_get(3dce)**, **dce_msg_get_default_msg(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_get

Purpose

Retrieves text of specified DCE message

Synopsis

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

unsigned char * dce_msg_get(
        unsigned32 message);
```

Parameters

Input

message ID of message to be retrieved.

Description

The **dce_msg_get()** routine is a convenience form of the **dce_msg_get_msg()** routine. Like **dce_msg_get_msg()**, **dce_msg_get()** retrieves the text for a specified *message* (which is a 32-bit DCE message ID as described in **dce_msg_intro(3dce)**). However, **dce_msg_get()** does not return a status code; it either returns the specified message successfully or fails (aborts the program) with an assertion error if the message could not be found or memory could not be allocated.

The routine implicitly determines the correct message catalog in which to access the specified message, and opens it; the caller only has to call this routine.

The routine first searches the appropriate message catalog for the message, and then (if it cannot find the catalog) searches the in-memory message table, if it exists.

The message, if found, is returned in allocated space to which the routine returns a pointer. The pointed-to space must be freed by the caller using **free()**.

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.

msg_s_bad_id
A message ID with an invalid technology or component was specified.

msg_s_no_cat_open
Could not open the message catalog for the specified message ID.

msg_s_no_cat_perm
Local file permissions prevented the program from opening the message catalog for the specified message ID.

msg_s_no_catalog
The message catalog for the specified message ID does not exist.

msg_s_no_default
Could not find the default message for the specified status code in the internal tables.

msg_s_no_memory
Could not allocate memory for message table, string copy, or other internal requirement.

msg_s_not_found
Could not find the text for the specified status code in either the in-core message tables or the message catalogs.

msg_s_ok_text
The operation was performed successfully.

Related Information

Functions: **dce_msg_define_msg_table(3dce)**, **dce_msg_get_default_msg(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_get_cat_msg

Purpose

Opens message catalog and retrieves message

Synopsis

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

unsigned char * dce_msg_get_cat_msg(
        unsigned32 message,
        error_status_t * status);
```

Parameters

Input

message ID of message to be retrieved.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_get_cat_msg()** routine is a convenience form of the **dce_msg_cat_get_msg()** routine. The difference between it and the latter routine is that **dce_msg_get_cat_msg()** does not require the message catalog to be explicitly opened; it determines the correct catalog from the *message* parameter (which is a 32-bit DCE message ID as described in **dce_error_inq_text(3dce)**), opens it, and returns a pointer to the message. If the message catalog is inaccessible, the routine returns an error. (See the routine **dce_msg_get()** for a description of the return value.) The space allocated for the message should not be freed. The

output pointer is useable until a call to another **dce_msg...** routine or a call to the **dce_error_inq_text()** routine.

The routine will fail if the message catalog is not correctly installed.

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_cat_close(3dce)**, **dce_msg_cat_get_msg(3dce),** **dce_msg_cat_open(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_get_default_msg

Purpose

Retrieves DCE message from in-memory tables

Synopsis

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

unsigned char * dce_msg_get_default_msg(
        unsigned32 message,
        error_status_t * status);
```

Parameters

Input

message ID of message to be retrieved.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_get_default_msg()** routine retrieves a message from the application's in-memory tables. It returns a pointer to static space that should not be freed. If the specified *message* (which is a 32-bit DCE message ID as described in **dce_error_inq_text(3dce)**) cannot be found in the in-memory tables, the routine returns NULL and fills in *status* with the appropriate error code.

This routine should be used only for message strings that will never have to be translated (see **dce_msg_translate_table(3dce)**).

All messages for a given component are stored in a single message catalog generated by the **sams** utility when the component is built. Messages may also be compiled

directly into the component code, thus rendering the successful retrieval of message text independent of whether or not the message catalogs were correctly installed. Generation of in-memory message tables is specified by the **incatalog** flag in the **sams** file in which the message text is defined. (See **sams(1dce)** for more information on **sams** files.) If the messages have been generated at compile time with this option specified, the **dce_msg_define_msg_table()** routine can be called by the application to set up an in-memory table containing the messages.

Examples

The following code fragment shows how **dce_msg_get_default_msg()** might be called to retrieve the in-memory copy of a message defined by a DCE application (whose serviceability component name is *app*):

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include "dce<app>msg.h"  /* test_msg is defined in this file  */
unsigned char       *my_msg;
error_status_t      status;
<. . .>
my_msg = dce_msg_get_default_msg(test_msg, &status);
printf("Message is: %s\n", my_msg);
```

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_define_msg_table(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_msg(3dce)**.

dce_msg_get_msg

Purpose

Retrieves a DCE message from its ID

Synopsis

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

unsigned char * dce_msg_get_msg(
        unsigned32 message,
        error_status_t * status);
```

Parameters

Input

message ID of message to be retrieved.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_get_msg()** routine retrieves the text for a specified *message* (which is a 32-bit DCE message ID as described in **dce_error_inq_text(3dce)**). The routine implicitly determines the correct message catalog in which to access the message, and opens it; the caller only has to call the routine.

The routine first searches the appropriate message catalog for the message, and then (if it cannot find the catalog) searches the in-memory message table. If the message cannot be found in either of these places, the routine returns a default string and fills in *status* with an error code. This routine thus always returns a string, even if there is an error (except for **msg_sno_memory**).

The message, if found, is returned in allocated space to which the routine returns a pointer. The pointed-to space must be freed by the caller using **free()**. If memory cannot be allocated, the routine returns NULL and fills in *status* with the **msg_s_no_memory** error code.

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.

See **dce_get_msg(3dce)**.

Related Information

Functions: **dce_msg_define_msg_table(3dce)**, **dce_msg_get(3dce)**, **dce_msg_get_default_msg(3dce)**.

dce_msg_translate_table

Purpose

Translates all in-memory messages in a table

Synopsis

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

void dce_msg_translate_table(
        dce_msg_table_t * table,
        unsigned32 count,
        error_status_t * status);
```

Parameters

Input

table A message table structure (defined in a header file generated by **sams** during compilation (see the **EXAMPLES** section), the contents of which are to be translated.

count The number of elements contained in the table.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_msg_translate_table()** routine overwrites the specified in-memory message table (that is, updates the in-memory table with the contents of a message table, which has changed for some reason; for example, because of a change in locale).

If any in-memory message is not found in the message catalog, all in-memory messages are left unchanged.

Examples

The following code fragment shows how **dce_msg_translate_table()** might be called (in an application whose serviceability component name is *app*) to translate a DCE application's in-memory message table, set up by an earlier call to **dce_msg_define_msg_table()**:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include "dce<app>msg.h"
char                *loc_return;
error_status_t      status;
<. . .>
dce_msg_translate_table(<app>_msg_table,
 sizeof(<app>_msg_table) / sizeof(<app>_msg_table[0]),
 &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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_msg_define_msg_table(3dce)**.

dce_pgm_printf, dce_pgm_fprintf, dce_pgm_sprintf

Purpose

Formatted DCE message output routines

Synopsis

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

int dce_pgm_printf(
        unsigned32 messageid,
        . . .);

int dce_pgm_fprintf(
        FILE * stream,
        unsigned32 messageid,
        . . .);

unsigned char *dce_pgm_sprintf(
        unsigned32 messageid,
        . . .);
```

Parameters

Input

messageid	The message ID, defined in the message's **code** field in the **sams** file.
stream	An open file pointer.
. . .	Any format arguments for the message string.

Description

The **dce_pgm_printf()** routine is equivalent to **dce_printf()**, except that it prefixes the program name to the message (in the standard style of DCE error messages), and appends a newline to the end of the message. The routine **dce_printf()** does neither. This allows clients (which do not usually use the serviceability interface) to produce

error (or other) messages which automatically include the originating application's name. Note that the application should call **dce_svc_set_progname()** first to set the desired application name. Otherwise, the default program name will be **PID#***nnnn*, where *nnnn* is the process ID of the application making the call.

The **dce_pgm_sprintf()** routine is similarly equivalent to **dce_sprintf()**, and the **dce_pgm_fprintf()** routine is similarly equivalent to **dce_fprintf()**.

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.

See **dce_msg_get(3dce)**.

Related Information

Functions: **dce_fprintf(3dce)**, **dce_msg_get_msg(3dce)**, **dce_printf(3dce)**, **dce_sprintf(3dce)**, **dce_svc_set_progname(3dce)**.

dce_printf, dce_fprintf, dce_sprintf

Purpose

Formatted DCE message output routines

Synopsis

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

int dce_printf(
        unsigned32 messageid,
        . . .);

int dce_fprintf(
        FILE * stream,
        unsigned32 messageid,
        . . .);

unsigned char *dce_sprintf(
        unsigned32 messageid,
        . . .);
```

Parameters

Input

messageid	The message ID, defined in the message's **code** field in the **sams** file.
stream	An open file pointer.
. . .	Any format arguments for the message string.

Description

The **dce_printf()** routine retrieves the message text associated with the specified *messageid*, and prints the message and its arguments on the standard output. The routine determines the correct message catalog and, if necessary, opens it. If the message catalog is inaccessible, and the message exists in an in-memory table, then

this message is printed. If neither the catalog nor the default message is available, a default message is printed.

The **dce_fprintf()** routine functions much like **dce_printf()**, except that it prints the message and its arguments on the specified stream.

The **dce_sprintf()** routine retrieves the message text associated with the specified *messageid*, and prints the message and its arguments into an allocated string that is returned. The routine determines the correct message catalog and, if necessary, opens it. If the message catalog is inaccessible, and the message exists in an in-memory table, then this message is printed. If neither the catalog nor the default message is available, a default message is printed. The **dce_pgm_printf()** routine is equivalent to **dce_printf()**, except that it prefixes the program name to the message (in the standard style of DCE error messages), and appends a newline to the end of the message. For more information, see the **dce_pgm_printf(3dce)** reference page.

Examples

Assume that the following message is defined in an application's **sams** file:

```
start
code          arg_msg
text          "This message has exactly %d, not %d argument(s)"
action        "None required"
explanation   "Test message with format arguments"
end
```

The following code fragment shows how **dce_sprintf()** might be called to write the message (with some argument values) into a string:

```
unsigned char     *my_msg;
my_msg = dce_sprintf(arg_msg, 2, 8);
puts(my_msg);
free(my_msg);
```

Of course, **dce_printf()** could also be called to print the message and arguments:

```
dce_printf(arg_msg, 2, 8);
```

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.

See **dce_msg_get(3dce)**.

Notes

The final formatted string generated by **dce_sprintf()** must not exceed 1024 bytes.

Related Information

Functions: **dce_msg_get_msg(3dce)**, **dce_svc_set_progname(3dce)**.

dce_server_disable_service

Purpose

Disables an individual service of a server

Synopsis

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

voiddce_server_disable_service(
        dce_server_handle_t server_handle,
        rpc_if_handle_t interface,
        error_status_t * status);
```

Parameters

Input

server_handle
: An opaque handle returned by **dce_server_register()**.

interface
: Specifies an opaque variable containing information the runtime uses to access interface specification data.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully. The only status code is **error_status_ok**.

Description

The **dce_server_disable_service()** routine disables an individual service that a server provides by unregistering the service's interface from the RPC runtime and marking the server's endpoints as disabled in the local **dced**'s endpoint mapper service.

For **dced** to recognize all of a server's services, a server should register all its application services using the **dce_server_register()** routine. If it later becomes necessary for the server to disable an interface, it can use the **dce_server_disable_service()** routine rather than unregistering the entire server.

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: **dce_server_enable_service(3dce)**, **dce_server_register(3dce)**, **rpc_intro(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_enable_service

Purpose

Enables an individual service for a server

Synopsis

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

voiddce_server_enable_service(
        dce_server_handle_t server_handle,
        rpc_if_handle_t interface,
        error_status_t * status);
```

Parameters

Input

server_handle
: An opaque handle returned by **dce_server_register()**.

interface
: Specifies an opaque variable containing information the runtime uses to access interface specification data.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully. The only status code is **error_status_ok**.

Description

The **dce_server_enable_service()** routine enables an individual service that a server provides by registering the service's interface with the RPC runtime, and registering the endpoints in the endpoint map. If the **dce_server_c_no_endpoints** flag was set with the **dce_server_register()** call prior to callibng this routine, the endpoints are not registered in the endpoint map.

A server commonly registers all its services with DCE at once by using the **dce_server_register()** routine. If necessary, a server can use the **dce_server_disable_service()** routine to disable individual services and then reenable them by using **dce_server_enable_service()**. However, suppose a server needs its services registered in a certain order, or it require application-specific activities between the registration of services. If a server requires this kind of control as services are registered, you can set the **server->services.list[i].flags** field of the **server_t** structure to **service_c_disabled** for individual services prior to calling **dce_server_register()**. Then, the server can call **dce_server_enable_service()** for each service when needed.

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: **dce_server_disable_service(3dce)**, **dce_server_register(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_inq_attr

Purpose

Obtains from **dced** the value of an attribute known to the server

Synopsis

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

voiddce_server_inq_attr(
        uuid_t attribute_uuid,
        sec_attr_t * value,
        error_status_t * status);
```

Parameters

Input

attribute_uuid
The UUID **dced** uses to identify an attribute.

Output

value Returns the attribute.

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

Description

The **dce_server_inq_attr()** routine obtains an attribute from the environment created by **dced** when it started the server. Each server maintains among other things, a list of attributes that are used to describe application-specific behavior.

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

dced_s_server_attr_not_found

dced_s_not_started_by_dced

Related Information

Functions: **dce_server_inq_server(3dce)** , **dce_server_inq_uuids(3dce)**, **dced_intro(3dce)**, **sec_intro(3sec)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_inq_server

Purpose

Obtains the server configuration data **dced** used to start the server

Synopsis

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

voiddce_server_inq_server(
        server_t ** server,
        error_status_t * status);
```

Parameters

Output

server Returns the structure that describes the server's configuration.

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

Description

The **dce_server_inq_server()** routine obtains the server configuration data (**srvrconf**) maintained by **dced** and used by **dced** to start the server. This routine is commonly called prior to registering the server to obtain the server data used as input to **dce_server_register()**.

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

dced_s_not_started_by_dced

dced_s_data_unavailable

Related Information

Functions: **dce_server_register(3dce)** , **dced_intro(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_inq_uuids

Purpose

Obtains the UUIDs that **dced** associates with the server's configuration and execution data

Synopsis

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

voiddce_server_inq_uuids(
        uuid_t * conf_uuid,
        uuid_t * exec_uuid,
        error_status_t * status);
```

Parameters

Output

conf_uuid Returns the UUID that **dced** uses to identify the server's configuration data. If a NULL value is input, no value is returned.

exec_uuid Returns the UUID that **dced** uses to identify the executing server. If a NULL value is input, no value is returned.

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

Description

The **dce_server_inq_uuids()** routine obtains the UUIDs that **dced** uses in its **srvrconf** and **srvrexec** services to identify the server's configuration and execution data. The server can then use **dced** API routines to access the data and perform other server management functions.

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

dced_s_not_started_by_dced

Related Information

Functions: **dce_server_inq_server(3dce)** , **dced_intro(3dce)**, **dced_*(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_register

Purpose

Registers a server with DCE

Synopsis

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

voiddce_server_register(
        unsigned32 flags,
        server_t * server,
        dce_server_register_data_t * data,
        dce_server_handle_t * server_handle,
        error_status_t * status);
```

Parameters

Input

flags Specifies options for server registration. Combinations of the following values may be used:

dce_server_c_no_protseqs

dce_server_c_no_endpoints

dce_server_c_ns_export

server Specifies the server data, commonly obtained from **dced** by calling **dce_server_inq_server()**. The **server_t** structure is described in **sec_intro(3sec)** .

data Specifies the array of data structures that contain the additional information required for the server to service requests for specific remote procedures. Each structure of the array includes the following:

- An interface handle (**ifhandle**) of type **rpc_if_handle_t**
- An entry point vector (**epv**) of type **rpc_mgr_epv_t**

- A number (*num_types*) of type **unsigned32** representing the number in the following array
- An array of server types (**types**) of type **dce_server_type_t**

The **dce_server_type_t** structure contains a UUID (**type**) of type **uuid_t** representing the object type, and a manager entry point vector (**epv**) of type **rpc_mgr_epv_t** representing the set of procedures implemented for the object type.

Output

server_handle
Returns a server handle, which is a pointer to an opaque data structure containing information about the server.

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

Description

By default, the **dce_server_register()** routine registers a DCE server by establishing a server's binding information for all valid protocol sequences, registering all the servers services with the RPC runtime, and entering the server's endpoints in **dced**'s endpoint mapper service.

Prior to calling the **dce_server_register()** routine, the server obtains the server configuration data from **dced** by calling **dce_server_inq_server()**. The server must also set up an array of registration data, where the size of the array represents all the server's services that are currently registered in the server configuration data of **dced** (**server->services.count**). If the memory for the array is dynamically allocated, it must not be freed until after the corresponding **dce_server_unregister()** routine is called.

You can modify the behavior of **dce_server_register()** Depending on the values of the *flags* parameter. If the flag has the value **dce_server_c_ns_export**, the the binding information is also exported to the namespace. The namespace entry is determined for each service by the **server->services.list[i].entryname** parameter. If this parameter has no value, the default value for the entire server is used (**server->entryname**). If the flag has the value **dce_server_c_no_endpoints**, the binding information is not registered with the endpoint map. Your application can use **rpc_ep_register()** to register specific binding information. If the flag has the value **dce_server_c_no_protseqs**, specific protocol sequences are used rather than all valid protocol sequences. Use of this flag

requires that the server first call **dce_server_use_protseq()** at least once for a valid protocol sequence.

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

rpc_s_no_memory

Related Information

Functions: **dce_server_inq_server(3dce)** , **dce_server_sec_begin(3dce)**, **dce_server_unregister(3dce)**, **dced_intro(3dce)**, **rpc_server_listen(3rpc)** .

Books: *DCE 1.2.2 Application Development Guide.*

dce_server_sec_begin

Purpose

Establishes a server to receive fully authenticated RPCs and to act as a client to do authenticated RPCs

Synopsis

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

voiddce_server_sec_begin(
        unsigned32 flags,
        error_status_t * status);
```

Parameters

Input

flags Flags are set to manage keys and setup a login context. Valid values include the following:

dce_server_c_manage_key

dce_server_c_login

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 **dce_server_sec_begin()** routine prepares a server to receive authenticated RPCs. It also sets up all that is required for the application, when behaving as a client to other servers, to do authenticated RPCs as a client. When authentication is required, this call must precede all other RPC and DCE server initialization calls, including

dce_server_register() . When your application is finished listening for RPCs, it should call the **dce_server_sec_done()** routine.

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

dced_s_need_one_server_prin

dced_s_not_started_by_dced

dced_s_no_server_keyfile

dced_s_cannot_create_key_mgmt_thr

dced_s_cannot_detach_key_mgmt_thr

Related Information

Functions: **dce_server_register(3dce)** , **dce_server_sec_done(3dce)**, **rpc_server_listen(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_sec_done

Purpose

Releases resources established for a server to receive (and when acting as a client, to send) fully authenticated RPCs

Synopsis

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

voiddce_server_sec_done(
        error_status_t * status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully. The only status code is **error_status_ok**.

Description

The **dce_server_sec_done()** routine releases the resources previously set up by a call to **dce_server_sec_begin()**. The **dce_server_sec_begin()** routine sets all that is needed for a server to receive authenticated RPCs and it also sets up all that is required for the application to do authenticated RPCs as a client. If this routine is used, it must follow all other server DCE and RPC initialization and cleanup calls.

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: **dce_server_sec_begin(3dce)**, **rpc_server_listen(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_unregister

Purpose

Unregisters a DCE server

Synopsis

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

voiddce_server_unregister(
        dce_server_handle_t * server_handle,
        error_status_t * status);
```

Parameters

Input

server_handle
: An opaque handle returned by **dce_server_register()**.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully. The only status code is **error_status_ok**.

Description

The **dce_server_unregister()** routine unregisters a DCE server by unregistering a server's services (interfaces) from the RPC runtime. When a server has stopped listening for remote procedure calls, it should call this routine.

The flags set with the corresponding **dce_server_register()** routine are part of the server handle's information used to determine what action to take or not take. These actions include removing the server's endpoints from the **dced**'s endpoint mapper service and unexporting binding information from the namespace.

Use the **dce_server_disable_service()** routine to disable specific application services rather than unregistering the whole server.

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: **dce_server_disable_service(3dce)**, **dce_server_register(3dce)**, **rpc_server_listen(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_server_use_protseq

Purpose

Tells DCE to use the specified protocol sequence for receiving RPCs

Synopsis

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

voiddce_server_use_protseq(
        dce_server_handle_t server_handle,
        unsigned char * protseq,
        error_status_t * status);
```

Parameters

Input

server_handle
: An opaque handle. Use the value of NULL.

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 **intro(3rpc)** reference page.)

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully. The only status code is **error_status_ok**.

Description

The **dce_server_use_protseq()** routine registers an individual protocol sequence with DCE. Typical servers use all valid protocol sequences, the default behavior for the **dce_server_register()** call, and so most servers do not need to call this **dce_server_use_protseq()** routine. However, this routine may be called prior to

dce_server_register(), to restrict the protocol sequences used. 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.

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: **dce_server_register(3dce)**, **rpc_intro(3rpc)**, **rpc_server_use_protseq(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dce_svc_components

Purpose

Returns registered component names

Synopsis

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

void dce_svc_components(
        dce_svc_stringarray_t * table,
        error_status_t * status);
```

Parameters

Output

table An array containing the names of all components that have been registered with the **dce_svc_register()** routine.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_components** routine returns an array containing the names of all components in the program that have been registered with the **dce_svc_register()** routine.

Examples

The following code fragment shows how the **dce_svc_components()** routine should be used in a DCE application's implementation of the serviceability remote interface. The function defined below is the implementation of the *app*_**svc_inq_components**

operation defined in the application's serviceability **.epv** file. Clients call this function remotely, and the function, when called, first checks the caller's authorization and then (if the client is authorized to perform the operation) calls the **dce_svc_components()** routine to perform the actual operation.

```
/*****
* app_svc_inq_components -- remote request for list of all
*  components registered by dce_svc_register().
*****/
static void
app_svc_inq_components(
handle_t h,
dce_svc_stringarray_t *table,
error_status_t *st)
{
int ret;
/* Check the client's permissions here, if insufficient,  */
/* deny the request. Otherwise, proceed with operation    */
dce_svc_components(table, 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. See **dce_svc_register(3dce)**.

Files

dce/service.idl

dce_svc_debug_routing

Purpose

Specifies how debugging messages are routed

Synopsis

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

void dce_svc_debug_routing(
        unsigned char * where,
        error_status_t * status);
```

Parameters

Input

where A four-field routing string, the format of which is described in **svcroute(5dce)**.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_debug_routing()** routine specifies both the level of an applications's serviceability debug messaging, and where the messages are routed. The *where* parameter is a four-field routing string, as described in **svcroute(5dce)**. All four fields are required.

The routine is used to specify the disposition of serviceability debug messages. If called before the component is registered (with **dce_svc_register()**), the disposition is stored until it is needed. In case of error, the *status* parameter is filled in with an error code.

To set only the debugging level for a component, use the **dce_svc_debug_set_levels()** routine.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_debug_set_levels(3dce)**.

Files: **svcroute(5dce)**.

dce_svc_debug_set_levels

Purpose

Sets the debugging level for a component

Synopsis

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

void dce_svc_debug_set_levels(
        unsigned char * where,
        error_status_t * status);
```

Parameters

Input

where A multifield string consisting of the component name separated by a colon from a comma-separated list of subcomponent/level pairs, as described in **svcroute(5dce)**.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_debug_set_levels()** routine sets serviceability debugging message level(s) for a component. The *where* parameter is a multifield string consisting of the component name separated by a colon from a comma-separated list of subcomponent/level pairs, as described in **svcroute(5dce)**. The subcomponents are specified by codes defined in the component's **sams** file; the levels are specified by single digits (1 through 9).

If the routine is called before the component is registered (with **dce_svc_register()**), the disposition is stored until it is needed. In case of error, the *status* parameter is filled in with an error code.

To set both the debug level and routing for a component, use the **dce_svc_debug_routing()** routine.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_debug_routing(3dce)**.

Files: **svcroute(5dce)**.

dce_svc_define_filter

Purpose

DCE serviceability filtering routines

Synopsis

#include <stdarg.h> #include <dce/dce.h> #include <pthread.h> #include <dce/svcfilter.h>

void dce_svc_define_filter(
dce_svc_handle_t *handle***,**
dce_svc_filter_proc_t *filter_function***,**
dce_svc_filterctl_proc_t *filter_ctl_function***,**
error_status_t * *status***);**

Description

The serviceability interface provides a hook into the message-output mechanism that allows applications to decide at the time of messaging whether the given message should be output or not. The application defines its own routine to perform whatever checking is desired, and installs the routine (the *filter_function* parameter) with a call to **dce_svc_define_filter()**.

The filter routine to be installed must have the signature defined by the **dce_svc_filter_proc_t** typedef. Once installed, the routine will be automatically invoked every time a serviceability routine is called to output a message. The filter receives a *prolog* argument which contains all the pertinent information about the message. If the filter returns TRUE, the message is output per the original serviceability call. If the filter returns FALSE, the message is not output. The information in the *prolog* allows such decisions to be made on the basis of severity level, subcomponent, message index, and so on. For details, see the header file **dce/svcfilter.h**.

In addition, an application that installs a message-filtering routine must also define a routine that can be called remotely to alter the operation of the filter routine. This procedure must have the signature defined by the **dce_svc_filterctl_proc_t** typedef. The routine will be invoked with an opaque byte array parameter (and its length), which

it is free to interpret in an appropriate manner. The remote-control routine is installed by the same call to **dce_svc_define_filter()** (as the *filter_ctl_function* parameter) in which the filter itself is installed. See **dce_svc_filter(3dce)**.

Examples

The following code fragment consists of example versions of an application's routines to filter serviceability messages, alter the behavior of the filter routine, and install the two routines.

```
/*****
* Filter routine-- this is the routine that's hooked into
* the serviceability mechanism when you install
* it by calling dce_svc_define_filter().
*****/
boolean app_filter(prolog, args)
dce_svc_prolog_t prolog;
va_list args;
{
if (filter_setting) {
printf("The value of filter_setting is TRUE\n");
printf("The progname is %s\n", prolog->progname);
if (prolog->attributes & svc_c_sev_notice)
printf("This is a Notice-type message\n");
switch (prolog->table_index) {
case app_s_server:
printf("Server subcomponent\n");
break;
case app_s_refmon:
printf("Refmon subcomponent\n");
break;
case app_s_manager:
printf("Manager subcomponent\n");
break;
}
}
return 1;
}
```

```
/*****
* Filter Control routine-- this is the entry point for
* the remote-control call to modify the filter
* routine's behavior.
*****/
void app_filter_control(arg_size, arg, status)
idl_long_int arg_size;
idl_byte *arg;
error_status_t *status;
{
if (strncmp(arg, "Toggle", arg_size) != 0)
return;
else {
filter_setting = (filter_setting == FALSE) ? TRUE : FALSE;
if (filter_setting)
printf("     FILTER IS TURNED ON\n");
else
printf("     FILTER IS TURNED OFF\n");
}
return;
}
/*****
* install_filters-- calls dce_svc_define_filter()
* to install the above 2 routines.
*****/
void install_filters()
{
unsigned32 status;
filter_setting = TRUE;
dce_svc_define_filter(app_svc_handle, app_filter,
dce_svc_filterctl_proc_t)app_filter_control, &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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_register(3dce)**, **DCE_SVC_DEFINE_HANDLE(3dce)**.

dce_svc_filter

Purpose

Controls behavior of serviceability filter

Synopsis

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

void dce_svc_filter(
        dce_svc_string_t component,
        idl_long_int arg_size,
        idl_byte * argument,
        error_status_t * status);
```

Parameters

Input

component The name of the serviceability-registered component, defined in the **component** field of the **sams** file.

arg_size The number of characters contained in *argument*.

argument A string value to be interpreted by the target component's filter control routine.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_filter()** routine controls the behavior of the serviceability message filtering routine, if one exists.

Along with the filter routine itself, the application also installs a filter control routine, whose purpose is to permit the behavior of the filter to be altered dynamically while the application is running. The **dce_svc_filter()** routine is the interface's call-in to such an installed filter control.

If an application has installed a serviceability filtering routine, and if filter remote control is desired, the application's filter routine (installed by the call to **dce_svc_define_filter()**) should be coded so that its operation can be switched to the various desired alternatives by the values of static variables to which it has access. These variables should also be accessible to the filter control routine. The filter control routine thus receives from **dce_svc_filter()** an argument string (which it uses to set the variables), the meaning of whose contents are entirely application-defined.

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.

See **dce_svc_register(3dce)**.

Files

dce/service.idl

dce_svc_log_close

Purpose

Closes an open log file

Synopsis

#include <dce/dce.h> #include <pthread.h> #include <dce/svclog.h>

void dce_svc_log_close(
dce_svc_log_handle_t *handle***,**
error_status_t * *status***);**

Parameters

Input

handle The handle (returned by **dce_svc_log_open()**) of the log file to be closed.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_log_close()** routine closes an open binary format serviceability log and releases all internal state associated with the 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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_log_get(3dce)**, **dce_svc_log_open(3dce)**, **dce_svc_log_rewind(3dce)**.

dce_svc_log_get

Purpose

Reads the next record from a binary log file

Synopsis

#include <dce/dce.h> #include <pthread.h> #include <dce/svclog.h>

void dce_svc_log_get(
dce_svc_log_handle_t *handle*,
dce_svc_log_prolog_t * *prolog*,
error_status_t * *status*);

Parameters

Input

handle The handle (returned by **dce_svc_log_open()**) of the log file to be read.

Output

prolog A pointer to a structure containing information read from the log file record.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_log_get()** routine reads the next entry from a binary format serviceability log, and fills in *prolog* with a pointer to a private data area containing the data read. The contents of the *prolog* structure are defined in **dce/svclog.h**.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_log_close(3dce)**, **dce_svc_log_open(3dce)**, **dce_svc_log_rewind(3dce)**.

dce_svc_log_open

Purpose

Opens binary log file

Synopsis

#include <dce/dce.h> #include <pthread.h> #include <dce/svclog.h>

void dce_svc_log_open(
const char * *name***,**
dce_svc_log_handle_t * *handle***,**
error_status_t * *status***);**

Parameters

Input

name The pathname of the log file to be opened.

Output

handle A filled-in handle to the opened log file specified by *name*.

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_log_open()** routine opens the binary log file specified by *name* for reading. If the call is successful, *handle* is filled in with a handle to be used with the other **dce_svc_log_** calls. On error, *status* will contain an error code.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_log_close(3dce)**, **dce_svc_log_get(3dce)**, **dce_svc_log_rewind(3dce)**.

dce_svc_log_rewind

Purpose

Rewinds binary log file to first record

Synopsis

```
#include
<dce/dce.h> #include <pthread.h> #include <dce/svclog.h>

void dce_svc_log_rewind(
        dce_svc_log_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle The handle (returned by **dce_svc_log_open()**) of the log file to be rewound.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_log_rewind()** routine rewinds the current reading position of the specified (by *handle*) binary log file to the first record.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_log_close(3dce)**, **dce_svc_log_get(3dce)**, **dce_svc_log_open(3dce)**.

dce_svc_printf

Purpose

Generates a serviceability message

Synopsis

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

voiddce_svc_printf(
        DCE_SVC(dce_svc_handle_t handle char * argtypes),
        const unsigned32 table_index,
        const unsigned32 attributes,
        const unsigned32 messageID,
        . . .);
```

Parameters

Input

handle	The caller's serviceability handle.
argtypes	Format string for the message.
table_index	The message's subcomponent name (defined in the **sams** file).
attributes	Any routing, severity, action, or debug attributes that are to associated with the generated message, OR'd together.
messageID	The message ID, defined in the message's **code** field in the **sams** file.
. . .	Any format arguments for the message string.

Description

The **dce_svc_printf()** routine is the normal call for writing or displaying serviceability messages. It cannot be called with a literal text argument. Instead, the message text is retrieved from a message catalog or an in-core message table. These are generated

by the **sams** utility, which in turn outputs sets of tables from which the messages are extracted for output.

There are two main ways in which to call the routine. If a message has been defined in the **sams** file with both **sub-component** and **attributes** specified, then the **sams** output will include a convenience macro for the message that can be passed as the single argument to **dce_svc_printf()**, for example:

```
dce_svc_printf(SIGN_ON_MSG);
```

The convenience macro's name will be generated from the uppercase version of the message's **code** value (as specified in the **sams** file), with the string **_MSG** appended.

If a convenience macro is not generated, or if you want to override some of the message's attributes at the time of output, then you must call the routine in its long form. An example of this form of the call looks as follows:

```
dce_svc_printf(DCE_SVC(app_svc_handle, ""), app_subcomponent,\
svc_c_sev_error | svc_c_route_stderr, messageID);
```

DCE_SVC() is a macro that *must* be passed as the first argument to **dce_svc_printf()** if a convenience macro is not being used. It takes two arguments:

- The caller's serviceability handle
- A format string for the message that is to be output

The format string is for use with messages that have been coded with argument specifiers. It is a character string consisting of the argument types as they would be passed to a **printf()** call. If the message is to be routed to a binary file, the format is extended to include a **%b** specifier; using **%b** in a different routing will give unpredictable results. The **%b** specifier takes two arguments: an integer size, and a buffer pointer.

The remaining arguments passed to **dce_svc_printf()** are as follows:

- Subcomponent table index

  This symbol is declared in the **sub-component** list coded in Part II of the **sams** file; its value is used to index into the subtable of messages in which the desired message is located.

- Message attribute(s)

  This argument consists of one or more attributes to be applied to the message that is to be printed. Note that you *must* specify at least one severity here. Multiple attributes are OR'd together, as shown in the following example.

  There are four categories of message attributes:

  Routing — The available routing attribute constants are as follows:

  - **svc_c_route_stderr**
  - **svc_c_route_nolog**

  However, most routing is done either by passing specially-formatted strings to **dce_svc_routing()** or by environment variable values. Note that using **svc_c_route_nolog** without using **svc_c_route_stderr** will result in no message being generated.

  Severity — The available severity attribute constants are as follows:

  - **svc_c_sev_fatal**
  - **svc_c_sev_error**
  - **svc_c_sev_warning**
  - **svc_c_sev_notice**
  - **svc_c_sev_notice_verbose**

  Action — The available message action attribute constants are as follows:

  - **svc_c_action_abort**
  - **svc_c_action_exit_bad**
  - **svc_c_action_exit_ok**
  - **svc_c_action_brief**
  - **svc_c_action_none**

  Note that **svc_c_action_brief** is used to suppress the standard prolog.

  Debug Level — Nine different debug levels can be specified (**svc_c_debug1**...**svc_c_debug9** or **svc_c_debug_off**).

- Message ID

  This argument consists of the message's **code**, as declared in the **sams** file.

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 routine has no return value.

Related Information

Functions: **dce_svc_register(3dce)**, **DCE_SVC_DEFINE_HANDLE(3dce)**.

dce_svc_register

Purpose

Registers a serviceability message table

Synopsis

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

dce_svc_handle_t dce_svc_register(
        dce_svc_subcomp_t * table,
        const idl_char * component_name,
        error_status_t * status);
```

Parameters

Input

table A message table structure (defined in a header file generated by **sams** during compilation).

component_name
The serviceability name of the component, defined in the **component** field of the **sams** file.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_register()** routine registers a serviceability message table. An application must call either it (or the **DCE_SVC_DEFINE_HANDLE()** macro) in order to set up its table(s) and obtain the serviceability handle it must have in order to use the serviceability interface.

Two parameters are required for the call: *table* is a pointer to the application's serviceability table, defined in a file called **dce***app***svc.h** generated by the **sams** utility. *component_name* is a string whose value is *app*, which is defined in the **component** field of the **sams** file in which the serviceability messages are defined.

On error, this routine returns NULL and fills in *status* with an error code.

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 following serviceability status codes are defined:

svc_s_assertion_failed
A programmer-developed compile-time assertion failed.

svc_s_at_end
No more data is available.

svc_s_bad_routespec
See **svcroute(5dce)** for information on routing specification format.

svc_s_cantopen
Permission denied or file does not exist; consult **errno**.

svc_s_no_filter
Attempted to send data to the filter-control handle for a component that does not have a filter registered.

svc_s_no_memory
Could not allocate memory for message table, string copy or other internal requirement.

svc_s_no_stats
The definition of the return value has not been specified.

svc_s_ok Operation performed.

svc_s_unknown_component
Could not find the service handle for a component.

Related Information

Functions: **dce_svc_debug_routing(3dce)**, **dce_svc_debug_set_levels(3dce)**, **dce_svc_define_filter(3dce)**, **dce_svc_routing(3dce)**, **dce_svc_unregister(3dce)**.

dce_svc_routing

Purpose

Specifies routing of serviceability messages

Synopsis

#include <dce/dce.h> #include <dce/svcremote.h>

void dce_svc_routing(
unsigned char * *where***,**
error_status_t * *status***);**

Parameters

Input

where A three-field routing string, as described in **svcroute(5)**.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_routing()** routine specifies how normal (non-debug) serviceability messages are routed. The *where* parameter is a three-field routing string, as described in **svcroute(5)**. For convenience, the first field of the routing specifier (which indicates the message severity type to which the routing is to be applied) may be an ***** (asterisk) to indicate that all messages, whatever their severity, should be routed as specified.

If the routine is called before the component is registered (with the **dce_svc_register()** routine), the routing is stored until it is needed. In case of error, the *status* parameter is filled in with an error code.

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.

See **dce_svc_register(3dce)**.

Files

dce/service.idl

dce_svc_set_progname

Purpose

Sets an application's program name

Synopsis

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

void dce_svc_set_progname(
        char *program_name,
        error_status_t * status);
```

Parameters

Input

program_name
: A string containing the name that is to be included in the text of all serviceability messages that the application generates during the session.

Output

status
: Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

This function sets the application's *program name*, which is included in serviceability messages. This allows serviceability messages from more than one application to be written to the same file and still be distinguishable as to their separate origins.

If **dce_svc_set_progname()** is not called, the application's generated serviceability messages will be identified by its process ID.

Examples

Suppose an application sets its program name to be **demo_program** , as follows:

```
dce_svc_set_progname("demo_program", &status);
```

Serviceability messages generated by the program will as a result look like the following:

```
1994-04-05-20:13:34.500+00:00I-- --- demo_program NOTICE app
main.c 123 0xa444e208 message text
```

If the application does not set its program name, its generated serviceability messages will have the following form:

```
1994-04-05-20:13:34.500+00:00I-- --- PID#9467 NOTICE app
main.c 123 0xa444e208 message text
```

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. See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_printf(3dce)** , **dce_svc_printf(3dce)**, **DCE_SVC_DEBUG(3dce)**.

dce_svc_table

Purpose

Returns a registered component's subcomponent table

Synopsis

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

voiddce_svc_table(
        dce_svc_string_t component,
        dce_svc_subcomparray_t * table,
        error_status_t * status);
```

Parameters

Input

component The name of the serviceability-registered component, defined in the **component** field of the application's **sams** file.

Output

table An array of elements, each of which describes one of the component's serviceability subcomponents (as defined in its **sams** file).

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_table** routine returns the serviceability subcomponent table registered with the specified component. The returned table consists of an array of elements, each of which describes one subcomponent. Each element consists of four fields, which contain the subcomponent name, its description, its message catalog ID, and the current value of its debug message level.

The first three of these values are specified in the **sams** file which is processed during the application's compilation, and from which the application's message catalogs and other serviceability and message files are generated.

Examples

The following code fragment shows how the remote operation might be called from an application's client side, and how the results might be printed out:

```
#include <dce/rpc.h>
#include <dce/service.h>
handle_t svc_bind_handle;
dce_svc_string_t component;
dce_svc_subcomparray_t subcomponents_table;
error_status_t remote_status;
int i;
dce_svc_inq_table(svc_bind_handle, component, &subcomponents_table,
&remote_status);
fprintf(stdout, "Subcomponent table size received is: %d...\n",
subcomponents_table.tab_size);
fprintf(stdout, "Subcomponent table contents are:\n");
for (i = 0; i < subcomponents_table.tab_size; i++)
{
fprintf(stdout, "Name: %s\n",
subcomponents_table.table[i].sc_name);
fprintf(stdout, "Desc: %s\n",
subcomponents_table.table[i].sc_descr);
fprintf(stdout, "Msg Cat ID: 0x%8.8lx\n",
(long) subcomponents_table.table[i].sc_descr_msgid);
fprintf(stdout, "Active debug level: %d\n\n",
subcomponents_table.table[i].sc_level);
}
```

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.

See **dce_svc_register(3dce)**.

Files

dce/service.idl

dce_svc_unregister

Purpose

Destroys a serviceability handle

Synopsis

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

voiddce_svc_unregister(
        dce_svc_handle_t handle,
        error_status_t * status);
```

Parameters

Input

handle The application's serviceability handle, originally returned by a call to **dce_svc_register()**, or filled in by the **DCE_SVC_DEFINE_HANDLE()** macro.

Output

status Returns the status code from this operation. The status code is a value that indicates whether the routine completed successfully and if not, why not.

Description

The **dce_svc_unregister()** routine destroys a serviceability handle. Calling it closes any open serviceability message routes and frees all allocated resources associated with the handle.

The *handle* parameter is the serviceability handle that was originally returned by the call to **dce_svc_register()**, or filled in by the **DCE_SVC_DEFINE_HANDLE()** macro. On error, the routine fills in *status* with an error code.

Note that it is not usually necessary to call this routine, since the normal process exit will perform the required cleanup.

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.

See **dce_svc_register(3dce)**.

Related Information

Functions: **dce_svc_register(3dce)**.

dced_binding_create

Purpose

Establishes a **dced** binding to one of the host services of a remote (or the local) **dced**

Synopsis

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

voiddced_binding_create(
        dced_string_t service,
        unsigned32 binding_flags,
        dced_binding_handle_t * dced_bh,
        error_status_t * status);
```

Parameters

Input

service A character string that specifies a host daemon service name and an optional remote host. A service name is specified with one of the following: **hostdata**, **srvrconf**, **srvrexec** , **secval**, or **keytab**. The format of a complete service and host specification is one of the following:

service_name
A service at the local host. Pre-existing defined values include

dced_c_service_hostdata

dced_c_service_srvrconf

dced_c_service_srvrexec

dced_c_service_secval

dced_c_service_keytab

*service_name***@hosts**/*host_name*
A service at a host anywhere in the local namespace.

/.:/hosts/*host_name* **/config/***service_name*
A complete specification for *service_name* **@***host*, where the host is anywhere in the local namespace.

/...*/cell* **/hosts/** *host_name***/config/** *service_name*
A service at a host anywhere in the global namespace.

binding_flags
The only valid flag value for this parameter is **dced_c_binding_syntax_default** .

Output

dced_bh Returns a **dced** binding handle which is a pointer to an opaque data structure containing information about an RPC binding, the host, the host service, and a local cache.

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

Description

The **dced** on each DCE host maintains the host services and provides a remote interface to them. The host services include the following:

- endpoint mapper
- host data management (**hostdata**)
- server management, including server configuration (**srvrconf**) and server execution (**srvrexec**)
- security validation (**secval**)
- key table management (**keytab**)

The **dced_binding_create()** routine establishes a dced binding to a **dced** service and it (or **dced_binding_from_rpc_binding()**)must be the first **dced** API routine called before an application can access one of the host services with other **dced** API routines. When an application is finished with the service, it should call the **dced_binding_free()** routine to free resources. To establish a **dced** binding to your local host's **dced**, you can use the service name by itself, and do not need to specify a host.

To access the endpoint map directly, use **rpc_mgmt_ep_elt_inq_begin()** and associated routines.

Examples

The following example establishes a **dced** binding to the server configuration service on the host **patrick**.

```
dced_binding_handle_t dced_bh;
error_status_t        status;
dced_binding_create("srvrconf@hosts/patrick",
dced_c_binding_syntax_default,
&dced_bh,
&status);
.
. /* Other routines including dced API routines. */
.
dced_binding_free(dced_bh, &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

dce_cf_e_no_mem

dced_s_invalid_arg

dced_s_no_memory

dced_s_unknown_service

rpc_s_entry_not_found

rpc_s_incomplete_name

rpc_s_invalid_object

rpc_s_name_service_unavailable

rpc_s_no_memory

rpc_s_no_more_bindings

rpc_s_no_ns_permission

Related Information

Functions: **dced_binding_free(3dce)** , **dced_binding_from_rpc_binding(3dce)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_binding_free

Purpose

Releases the resources associated with a **dced** binding handle

Synopsis

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

voiddced_binding_free(
        dced_binding_handle_t dced_bh,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies a **dced** binding handle to free for a **dced** service on a specific host.

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 **dced_binding_free()** routine frees resources used by a **dced** binding handle and referenced information. Use this routine when your application is finished with a host service to break the communication between your application and the **dced** . The **dced** binding handle and referenced information is created with the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_binding_from_rpc_binding

Purpose

Establishes a **dced** binding to one of the host services on the host specified in an existing RPC binding handle

Synopsis

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

voiddced_binding_from_rpc_binding(
        dced_service_type_t service,
        handle_t rpc_bh,
        dced_binding_handle_t * dced_bh,
        error_status_t * status);
```

Parameters

Input

service A variable that specifies one of the host services. A valid variable name includes one of the following:

dced_e_service_type_hostdata

dced_e_service_type_srvrconf

dced_e_service_type_srvrexec

dced_e_service_type_secval

dced_e_service_type_keytab

rpc_bh An RPC binding handle to some server.

Output

dced_bh Returns a **dced** binding handle which is a pointer to an opaque data structure containing information about an RPC binding, the host, the host service, and a local cache.

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

Description

The **dced** on each DCE host maintains the host services and provides a remote interface to the services. The **dced_binding_from_rpc_binding()** routine establishes a **dced** binding to a **dced** service, and it (or **dced_binding_create()**) must be the first **dced** API routine called before an application can access one of the host services with other **dced** routines. When an application is finished with the service, it should call the **dced_binding_free()** routine to free resources.

Prior to using the RPC binding in this routine, make a copy of the binding by using the **rpc_binding_copy()** routine. This is necessary if the application needs to continue using the RPC binding, because otherwise the **dced** binding takes over the RPC binding.

The RPC binding may be obtained from a call to specific RPC runtime routines such as the routines **rpc_binding_from_string_binding(3rpc)** , **rpc_ns_binding_import_next(3rpc)** , or **rpc_ns_binding_lookup_next(3rpc)** .

Examples

This example obtains an RPC binding from a string binding, and it later makes a copy of the RPC binding for use in the **dced_binding_from_rpc_binding()** call.

```
handle_t                rpc_bh, binding_handle;
dced_binding_handle_t   dced_bh;
dced_service_type_t     service_type;
error_status_t          status;
unsigned_char_t         string_binding[STRINGLEN];
.
.
.
rpc_binding_from_string_binding(string_binding, &binding_handle,
&status);
.
.
```

```
.
rpc_binding_copy(binding_handle, &rpc_bh, &status);
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh,
&status);
.
. /* Other routines including dced API routines. */
.
dced_binding_free(dced_bh, &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

dced_s_no_memory

dced_s_unknown_service

ept_s_cant_perform_op

ept_s_database_invalid

ept_s_invalid_context

ept_s_invalid_entry

rpc_s_comm_failure

rpc_s_fault_context_mismatch

rpc_s_invalid_arg

rpc_s_invalid_binding

rpc_s_no_more_elements

rpc_s_wrong_kind_of_binding

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_free(3dce)**, **rpc_binding_copy(3rpc)**, **rpc_binding_from_string_binding(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_binding_set_auth_info

Purpose

Sets authentication and authorization information for a **dced** binding handle

Synopsis

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

voiddced_binding_set_auth_info(
        dced_binding_handle_t dced_bh,
        unsigned32 protect_level,
        unsigned32 authn_service,
        rpc_auth_identity_handle_t authn_identity,
        unsigned32 authz_service,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for which to set the authentication and authorization information.

protect_level Specifies the protection level for **dced** API calls that will use the **dced** binding handle *dced_bh*.

authn_service
Specifies the authentication service to use for **dced** API calls that will use the **dced** binding handle *dced_bh*.

authn_identity
Specifies a handle for the data structure that contains the calling application's authentication and authorization credentials appropriate for the selected *authn_service* and *authz_service* services.

Specify NULL to use the default security login context for the current address space.

authz_service
Specifies the authorization service to be implemented by **dced** for the host service 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 **dced_binding_set_auth_info()** routine sets up the **dced** binding handle so it can be used for authenticated calls that include authorization information. The **rpc_binding_set_auth_info()** routine performs in the same way as this one. See it for details of the parameters and values. Prior to calling this routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

This example establishes a **dced** binding to a host's key table service, and then it calls **dced_binding_set_auth_info()** so that the application is authorized to access remote key tables by using additional calls to the key table service.

```
dced_binding_handle_t   dced_bh;
error_status_t          status;
dced_binding_create((dced_string_t)"keytab@hosts/patrick",
dced_c_binding_syntax_default,
&dced_bh,
&status);
dced_binding_set_auth_info(dced_bh,
rpc_c_protect_level_default,
rpc_c_authn_pkt_privacy,
NULL,
rpc_c_authz_dce,
&status);
.
. /* Other routines including dced API routines. */
```

.

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

dced_s_bad_binding

dced_s_no_support

ept_s_not_registered

rpc_s_authn_authz_mismatch

rpc_s_binding_incomplete

rpc_s_comm_failure

rpc_s_invalid_binding

rpc_s_mgmt_op_disallowed

rpc_s_rpcd_comm_failure

rpc_s_unknown_authn_service

rpc_s_unsupported_protect_level

rpc_s_wrong_kind_of_binding

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **rpc_binding_set_auth_info(3rpc)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_entry_add

Purpose

Adds a **keytab** or **hostdata** entry to a host's **dced** for an existing file on that host

Synopsis

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

voiddced_entry_add(
        dced_binding_handle_t dced_bh,
        dced_entry_t * entry,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

Input/Output

entry Specifies the data entry to add to the service.

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 **dced_entry_add()** routine adds a data entry to a **dced** service. The data it refers to must already exist in a file on the **dced**'s host. You can only add **hostdata** or **keytab** entries.

A service's data entries do not contain the actual data. Instead, they contain a UUID, a name for the entry, a brief description of the item, and a storage tag that describes the location of the actual data. In the cases of the **hostdata** and **keytab** services, the

data for each entry is stored in a file. The **dced** uses this two-level scheme so that it can manipulate different kinds of data in the same way and so names are independent of local file system requirements.

The **hostdata** and **keytab** services each have their respective routines to create new data and at the same time, add a new entry to the appropriate service. These routines are **dced_hostdata_create()** and **dced_keytab_create().**

Prior to calling the **dced_entry_add()** routine, the application must have established a valid **dced** binding handle for the **hostdata** or **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example shows how to add a printer configuration file to the **hostdata** service. The example creates a **dced** binding to the local **hostdata** service, an entry data structure is filled in with the storage tag containing the full path of the existing configuration file, and finally, the **dced_entry_add()** routine is called.

```
dced_binding_handle_t dced_bh;
error_status_t        status;
dced_entry_t          entry;
dced_binding_create(dced_c_service_hostdata,
dced_c_binding_syntax_default,
&dced_bh,
&status);
uuid_create(&(entry.id), &status);
entry.name = (dced_string_t)("NEWERprinter");
entry.description = (dced_string_t)("Configuration for a new printer.");
entry.storage_tag = (dced_string_t)("/etc/NEWprinter");
dced_entry_add(dced_bh, &entry, &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

db_s_readonly

db_s_store_failed

dced_s_already_exists

dced_s_bad_binding

dced_s_import_cant_access

dced_s_no_support

rpc_s_b inding_has_no_auth

sec_acl_invalid_permission

uuid_s_no_address

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_remove(3dce)**, **dced_hostdata_create(3dce)**, **dced_keytab_create(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_entry_get_next

Purpose

Obtains one data entry from a list of entries of a **dced** service

Synopsis

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

voiddced_entry_get_next(
        dced_cursor_t cursor,
        dced_entry_t ** entry,
        error_status_t * status);
```

Parameters

Input/Output

cursor Specifies the entry list's cursor that points to an entry, and returns the cursor advanced to the next entry in the list.

Output

entry Returns a pointer to an entry.

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

Description

The **dced_entry_get_next()** routine obtains a pointer to a data entry, and advances the cursor to the next entry in the list. This routine is commonly used in a loop to traverse a host service's entry list. The data is obtained in an undetermined order. Prior to using this routine, the application must call **dced_initialize_cursor()** to obtain a list of entries and to establish the beginning of the cursor. When the application is finished traversing the entry list, it should call **dced_release_cursor()** to release resources.

A data entry does not contain the actual data, but it contains the name, identity, description, and storage location of the data. In the cases of **hostdata** and **keytab** services, the data for each entry is stored in a file. In the cases of **srvrconf** and **srvrexec** services, data is stored in memory. The **dced** uses this two-level scheme so that it can manipulate different kinds of data in the same way.

Prior to using the **dced_entry_get_next()** routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

In the followin g example, a **dced** binding is obtained from a service type and an existing rpc binding handle. After establishing an entry list cursor, the **dced_entry_get_next()** routine obtains an entry, one at a time, and the name and description of each entry is displayed until the entry list is exausted.

```
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh, &status);
dced_initialize_cursor(dced_bh, &cursor, &status);
for( ; ; ) { /* forever loop */
dced_entry_get_next(cursor, &entry, &status);
if(status != error_status_ok) break;
display(entry->name, entry->description); /* application specific */
}
dced_release_cursor(&cursor, &status);
dced_binding_free( dced_bh, &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

dced_s_no_more_entries

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_initialize_cursor(3dce)**, **dced_release_cursor(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_entry_remove

Purpose

Removes a **hostdata** or **keytab** data entry from a **dced** service's list of entries

Synopsis

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

voiddced_entry_remove(
        dced_binding_handle_t dced_bh,
        uuid_t * entry_uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

entry_uuid Specifies the UUID of the entry to be removed from the service.

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 **dced_entry_remove()** routine removes an entry from the **hostdata** or **keytab** service entry list of **dced**. It does not remove the actual data stored in the file, but makes it inaccessible from a remote host by way of the **dced**'s user interfaces which include the **dced** API and the DCE control program, **dcecp**. Each host service that maintains data also maintains a list of data entries. A data entry contains a name, a UUID, a brief description, and a storage tag indicating the location of the actual data.

To delete both the data and entry for the **hostdata**, **keytab**, or **srvrconf** services, use **dced_hostdata_delete()**, **dced_keytab_delete()**, or **dced_server_delete()**, respectively. (The **srvrexec** service is maintained only by **dced** and the **secval** service does not maintain data, so you cannot remove data for these services.)

Applications commonly obtain an entry by traversing the entry list using the **dced_entry_get_next()** routine with its associated cursor routines.

Prior to calling the **dced_entry_remove()** routine, the application must have established a valid **dced** binding handle to the **hostdata** or **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_del_failed

db_s_key_not_found

db_s_readonly

dced_s_bad_binding

dced_s_no_support

dced_s_not_found

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_hostdata_delete(3dce)**, **dced_initialize_cursor(3dce)**, **dced_keytab_delete(3dce)**, **dced_server_delete(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_hostdata_create

Purpose

Creates a **hostdata** item and the associated entry in **dced** on a specific host

Synopsis

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

voiddced_hostdata_create(
        dced_binding_handle_t dced_bh,
        dced_entry_t * entry,
        dced_attr_list_t * data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the host data service on a specific host.

Input/Output

entry Specifies the **hostdata** entry to create. You supply a name (**entry->name**), description (**entry->description**), and file name (**entry->storage_tag**), in the form of **dced** strings. You can supply a UUID (**entry->id**) for **dced** to use or you can use a NULL value and **dced** will generate a new UUID for the entry.

Input

data Specifies the data created and written to a file on the host. The **dced_attr_list_t** consists of a count of the number of attributes, and an array of attributes of type **sec_attr_t**. The reference OSF implementation has one attribute for a **hostdata** item (file contents). However some vendors may provide multiple attributes.

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 **dced_hostdata_create()** routine creates a new host data item in a file on the host to which the **dced** binding handle refers, and it generates the associated **hostdata** entry in the host's **dced**.

If data that you want to add to the host data service already exists on the host (in a file), you can add it to the service by calling **dced_entry_add()**, which only creates the new data entry for **dced**.

Prior to calling the **dced_hostdata_create()** routine, the application must have established a valid **dced** binding handle to the **hostdata** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example creates a binding to the host data service on the local host, creates the entry data, and fills in the data structure for one attribute to a hypothetical printer configuration. The attribute represents a plain-text file containing one string of data.

```
dced_binding_handle_t    dced_bh;
error_status_t           status;
dced_entry_t             entry;
dced_attr_list_t         data;
int                      num_strings, str_size;
sec_attr_enc_str_array_t *attr_array;
dced_binding_create(dced_c_service_hostdata,
dced_c_binding_syntax_default,
&dced_bh,
&status);
/*Create an Entry. */
uuid_create(&entry.id, &status);
```

```
entry.name        = (dced_string_t)("NEWERprinter");
entry.description = (dced_string_t)("Configuration for a new printer.");
entry.storage_tag = (dced_string_t)("/etc/NEWprinter");
/* create the attributes */
data.count  = 1;
num_strings = 1;
data.list   = (sec_attr_t *)malloc( data.count * sizeof(sec_attr_t) );
data.list->attr_id = dced_g_uuid_fileattr;
data.list->attr_value.attr_encoding = sec_attr_enc_printstring_array;
str_size    = sizeof(sec_attr_enc_str_array_t) +
num_strings * sizeof(sec_attr_enc_printstring_p_t);
attr_array  = (sec_attr_enc_str_array_t *)malloc(str_size);
data.list->attr_value.tagged_union.string_array = attr_array;
attr_array->num_strings = num_strings;
attr_array->strings[0]=
(dced_string_t)("New printer configuration data");
dced_hostdata_create(dced_bh, &entry, &data, &status);
dced_binding_free( dced_bh, &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

db_s_key_not_found

db_s_readonly

db_s_store_failed

dced_s_already_exists

dced_s_bad_binding

dced_s_cant_open_storage_file

dced_s_import_already_exists

dced_s_unknown_attr_type

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_add(3dce)**, **dced_hostdata_read(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_hostdata_delete

Purpose

Deletes a **hostdata** item from a specific host and removes the associated entry from **dced**

Synopsis

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

voiddced_hostdata_delete(
        dced_binding_handle_t dced_bh,
        uuid_t * entry_uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **hostdata** service on a specific host.

entry_uuid Specifies the UUID of the **hostdata** entry (and associated data) to delete.

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 **dced_hostdata_delete()** routine deletes a **hostdata** item (a file) from a specific host, and removes the associated entry from the host data service of that host's **dced**.

If you want to only make the data inaccessible remotely but not delete it, use the **dced_entry_remove()** routine which only removes the data's **hostdata** entry.

Prior to calling the **dced_hostdata_delete()** routine, the application must have established a valid **dced** binding handle for the **hostdata** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Warnings

Do not delete the standard **hostdata** items such as *cell_name*, **cell_aliases**, *host_name*, **post_processors**, or **dce_cf.db**. This will cause operational problems for the host.

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

db_s_bad_index_type

db_s_del_failed

db_s_iter_not_allowed

db_s_key_not_found

dced_s_bad_binding

dced_s_cant_remove_storage_file

dced_s_not_found

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_remove(3dce)**, **dced_hostdata_read(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_hostdata_read

Purpose

Reads a **hostdata** item maintained by **dced** on a specific host

Synopsis

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

voiddced_hostdata_read(
        dced_binding_handle_t dced_bh,
        uuid_t * entry_uuid,
        uuid_t * attr_uuid,
        sec_attr_t ** data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **hostdata** service on a specific host.

entry_uuid Specifies the **hostdata** entry UUID associated with the data to read.

attr_uuid Specifies the UUID associated with an attribute of the data. The attribute is either plain text (**dced_g_uuid_fileattr**) or binary (**dced_g_uuid_binfileattr**). Some vendors may allow other attributes.

Output

data Returns the data for the item. See the **sec_intro(3sec)** reference page for details on the **sec_attr_t** data type.

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

Description

The **dced_hostdata_read()** routine returns a **hostdata** item maintained by **dced** on a specific host. The standard data items include the cell name, a list of cell aliases, the host name, a list of UUID-program pairs (post_processors), and the DCE configuration database, among other items.

For programming convenience, the following global variables are defined for the *entry_uuid* of some standard data items:

dced_g_uuid_cell_name

dced_g_uuid_cell_aliases

dced_g_uuid_host_name

dced_g_uuid_hostdata_post_proc

dced_g_uuid_dce_cf_db

dced_g_uuid_pe_site

dced_g_uuid_svc_routing

Other host-specific data items may also be maintained by the **hostdata** service. The UUIDs for these are established when the data item is created (see **dced_hostdata_create()**). After the application reads host data and when it is done with the data, it should call the **dced_objects_release()** routine to release the resources allocated.

Each **hostdata** item for a specific host is stored in a local file. The name of an item's storage file is indicated in the storage tag field of each **dced hostdata** entry.

You can also use the **dced_object_read()** routine to read the text of a **hostdata** item. You might use this routine if your application needs to read data for other host services (**srvrconf**, **srvrexec**, or **keytab**) in addition to data for the **hostdata** service.

Prior to calling the **dced_hostdata_read()** routine, the application must have established a valid **dced** binding handle to the **hostdata** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_key_not_found

dce_cf_e_file_open

dce_cf_e_no_match

dce_cf_e_no_mem

dced_s_bad_binding

dced_s_cant_open_storage_file

dced_s_invalid_attr_type

dced_s_no_memory

sec_acl_invalid_permission

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_object_read(3dce)**, **dced_objects_release(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_hostdata_write

Purpose

Replaces an existing **hostdata** item maintained by **dced** on a specific host

Synopsis

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

voiddced_hostdata_write(
        dced_binding_handle_t dced_bh,
        uuid_t * entry_uuid,
        dced_attr_list_t * data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the host data service on a specific host.

entry_uuid Specifies the **hostdata** entry UUID to associate with the data to be written.

data Specifies the data to write. The **dced_attr_list_t** consists of a count of the number of attributes, and an array of attributes of type **sec_attr_t**. The reference OSF implementation has one attribute for a hostdata item (file contents). However some vendors may require multiple attributes.

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 **dced_hostdata_write()** routine replaces existing data for a **hostdata** item maintained by **dced** on a specific host. If the *entry_uuid* is not one maintained by **dced**, an error is returned and a new entry is *not* created. Use **dced_hostdata_create()** to create a new entry.

Prior to calling the **dced_hostdata_write()** routine, the application must have established a valid **dced** binding handle to the **hostdata** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_key_not_found

dced_s_bad_binding

dced_s_cant_open_storage_file

dced_s_no_postprocessors

dced_s_postprocessor_file_fail

dced_s_postprocessor_spawn_fail

dced_s_unknown_attr_type

sec_acl_invalid_permission

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_hostdata_create(3dce)**, **dced_hostdata_read(3dce)**.

Books: *DCE 1.2.2 Application Development Guide.*

dced_initialize_cursor

Purpose

Sets a cursor to the start of a cached list of data entries for a **dced** service

Synopsis

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

voiddced_initialize_cursor(
        dced_binding_handle_t dced_bh,
        dced_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

Output

cursor Returns the cursor used to traverse the list of data entries, one at a time. The cursor is an opaque data structure that is used to keep track of the entries between invocations of the **dced_entry_get_next()** 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 **dced_initialize_cursor()** routine sets a cursor at the start of a DCE host service's list of data entries. The cursor is then used in subsequent calls to **dced_entry_get_next()** to obtain individual data entries. When the application is finished traversing the entry list, it should call **dced_release_cursor()** to free the resources allocated for the cursor.

The valid services for this routine that have entry lists include **hostdata**, **srvrconf**, **srvrexec**, and **keytab**.

If a service's entry list is small, it may be more efficient to obtain the entire list using the **dced_list_get()** routine, rather than using cursor routines. This is because **dced_list_get()** guarantees that the list is obtained with one remote procedure call. However, your application is scalable if you use the cursor routines. This is because when an entry list is very large, it may be more efficient (or even necessary) to obtain the list in chunks with more than one remote procedure call.

Prior to calling the **dced_initialize_cursor()** routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_key_not_found

dced_s_bad_binding

dced_s_no_memory

dced_s_no_support

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_get_next(3dce)**, **dced_list_get(3dce)**, **dced_release_cursor(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_inq_id

Purpose

Obtains the entry UUID that **dced** associates with a name

Synopsis

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

void dced_inq_id(
        dced_binding_handle_t dced_bh,
        dced_string_t name,
        uuid_t * uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

name Specifies the name for which to obtain the *uuid*.

Output

uuid returns the UUID associated with the *name* input.

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

Description

The **dced_inq_id()** routine obtains the UUID associated with a name in a service of a specific host's **dced**. Applications and administrators use strings maintained by **dced** to identify data, but **dced** and its API must associate each data entry with a UUID. This routine is valid for the **hostdata**, **srvrconf** , **srvrexec**, and **keytab** services.

Prior to calling this routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example establishes a **dced** binding to a host's server configuration service. The example then obtains the UUID of some known server in order to read the server's configuration data.

```
dced_binding_handle_t dced_bh;
server_t              conf;
dced_string_t         server_name;
uuid_t                srvrconf_id;
error_status_t        status;
dced_binding_create("srvrconf@hosts/patrick",
dced_c_binding_syntax_default,
&dced_bh,
&status);
dced_inq_id(dced_bh, server_name, &srvrconf_id, &status);
dced_object_read(dced_bh, &srvrconf_id, (void**)&(conf), &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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_key_not_found

dced_s_not_found

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_inq_name(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_inq_name

Purpose

Obtains the entry name that **dced** associates with a UUID

Synopsis

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

void dced_inq_name(
        dced_binding_handle_t dced_bh,
        uuid_t * uuid,
        dced_string_t * name,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

uuid Specifies the UUID for which to obtain the *name*.

Output

name Returns the name associated with the *uuid* input.

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

Description

The **dced_inq_name()** routine obtains the name associated with a UUID in a service of a specific host's **dced**.

A name is a label for each data entry to help applications and administrators identify all data maintained by **dced**. The **dced** requires UUIDs to keep track of the data it maintains. But it also maintains a mapping of UUIDs to names so that other applications and administrators can more easily access the data by using a recognizable

name rather than a cumbersome UUID. A name is a label for **hostdata** items, **srvrconf** and **srvrexec** servers, and **keytab** tables.

Prior to calling this routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example establishes a **dced** binding handle to the local host data service, reads an entry, and uses **dced_inq_name()** to get the name associated with the attribute ID.

```
dced_binding_handle_t dced_bh;
uuid_t                entry_uuid;
sec_attr_t            *data_ptr;
error_status_t        status;
.
.
.
dced_binding_create(dced_c_service_hostdata,
dced_c_binding_syntax_default,
&dced_bh,
&status);
dced_hostdata_read(dced_bh,
&entry_uuid,
&dced_g_uuid_fileattr,
&data_ptr,
&status);
dced_inq_name(dced_bh, data_ptr->sec_attr.attr_id, &name, &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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_key_not_found

dced_s_not_found

sec_acl_invalid_permission

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_inq_id(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_add_key

Purpose

Adds a key (server password) to a specified key table on a specific host

Synopsis

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

voiddced_keytab_add_key(
        dced_binding_handle_t dced_bh,
        uuid_t * keytab_uuid,
        dced_key_t * key,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

keytab_uuid Specifies the UUID that **dced** uses to identify the key table to which the key is to be added.

Input/Output

key Specifies the key to be added. Some fields are completed by **dced**. See **dced_intro(3dce)**.

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 **dced_keytab_add_key()** routine adds a key to a server's key table (file) on a specific host, without changing the key in the security registry. (Servers use **sec_key_mgmt_set_key(3sec)** to do this for their own local key table.)

Most management applications use the **dced_keytab_change_key()** routine to remotely change a key because it also changes the key in the security registry.

Managing the same key in multiple key tables is a more complex process. The security registry needs a copy of a server's key, so that during the authentication process, it can encrypt tickets that only a server with that key can later decrypt. Part of updating a key in the security registry also includes automatic version number updating. When servers share the same principle identity they use the same key. If these servers are on different hosts, then the key must be in more than one key table. (Even if the servers are on the same host, it is possible for their keys to be in different key tables, although this is not a recommended key management practice.) When the same keys in different tables need changing, one (perhaps the master server or busiest one) is changed using **dced_keytab_change_key()** which also causes an automatic version update. However, all other copies of the key must be changed using the **dced_keytab_add_key()** routine so that the version number does not change again.

Prior to calling **dced_keytab_add_key()** the application must have established a valid **dced** binding handle to the **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_key_not_found

dced_s_bad_binding

dced_s_key_v0_not_allowe

dced_s_key_version_mismatch

dced_s_need_privacy

dced_s_random_key_not_allowed

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_key_unsupported

sec_key_mgmt_e_key_ve rsion_exists

sec_key_mgmt_e_unauthorized

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_keytab_change_key(3dce)**, **sec_key_mgmt_set_key(3sec)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_change_key

Purpose

Changes a key (server password) in both a key table and in the security registry

Synopsis

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

voiddced_keytab_change_key(
        dced_binding_handle_t dced_bh,
        uuid_t * keytab_uuid,
        dced_key_t * key,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

keytab_uuid Specifies the UUID **dced** uses to identify the key table in which the key is to be changed.

Input/Output

key Specifies the new key. Some fields are modified by **dced**.

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 **dced_keytab_change_key()** routine updates a key in both the key table on a specific host and in the security registry. Managembent applications change keys

remotely with this routine. (Servers can change their own keys locally with the **sec_key_mgmt_change_key()** routine.)

The security registry needs a copy of a server's current key, so that during the authentication process, it can encrypt tickets that only a server with that key can later decrypt. When a management application calls **dced_keytab_change_key()**, **dced** first tries to make the modification in the security registry, and, if successful, it then modifies the key in the key table. The old key is not really replaced, but a new version and key is established for all new authenticated communication. The old version is maintained in the key table (and registry too) for a time, so that existing clients with valid tickets can still communicate with the server. The old key is removed depending on the local cell's change policy and whether the server calls **sec_key_mgmt_garbage_collect()** to purge its old keys explicitly, or calls **sec_key_mgmt_manage_key()** to purge them implicitly.

When more than one server shares the same principal identity, the servers use the same key. If you need to change the same key in more than one key table, use **decd_keytab_change_key()** for one change and then use the **dced_keytab_add_key()** routine for all others.

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

db_s_bad_index_type

db_s_key_not_found

dced_s_bad_binding

dced_s_key_version_mismatch

dced_s_need_privacy

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_authn_unavailable

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_key_unsupported

sec_key_mgmt_e_key_version_exists

sec_key_mgmt_e_not_implemented

sec_key_mgmt_e_unauthorized

sec_rgy_object_not_found

sec_rgy_server_unavailable

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_keytab_add_key(3dce)**, **sec_key_mgmt_change_key(3sec)**.

Books: *DCE 1.2.2 Application Development Guide.*

dced_keytab_create

Purpose

Creates a key table with a list of keys (server passwords) in a new file on a specific host

Synopsis

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

voiddced_keytab_create(
        dced_binding_handle_t dced_bh,
        dced_entry_t * keytab_entry,
        dced_key_list_t * keys,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

Input/Output

keytab_entry Specifies the **keytab** entry to create for **dced** .

keys Specifies the list of keys to be written to the key table 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 **dced_keytab_create()** routine creates a new key table file on a specific host, and it generates the associated **keytab** service entry in **dced**. This routine is used by

management applications to remotely create a key table. Servers typically create their own key table locally using the **sec_key_mgmt_set_key()** routine. However, if several servers on different hosts share the same principal, each host requires a local copy of the key table.

If a key table that you want to add to the **keytab** service already exists on the host, you can add it to the service by calling **dced_entry_add()**. This routine creates a new **keytab** service entry by associating the existing key table file with a new UUID in **dced**.

Prior to calling the **dced_keytab_create()** routine, the application must have established a valid **dced** binding handle to the **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_header_type

db_s_bad_index_type

db_s_bad_index_type

db_s_iter_not_allowed

db_s_key_not_found

db_s_readonly

db_s_store_failed

dced_s_already_exists

dced_s_bad_binding

dced_s_import_already_exists

dced_s_need_privacy

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_key_unsupported

sec_key_mgmt_e_key_version_exists

sec_key_mgmt_e_unauthorized

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_add(3dce)**, **sec_key_mgmt_set_key(3sec)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_delete

Purpose

Deletes a key table file from a specific host

Synopsis

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

voiddced_keytab_delete(
        dced_binding_handle_t dced_bh,
        uuid_t * keytab_uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

keytab_uuid Specifies the UUID of the **keytab** entry and associated key table to be deleted.

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 **dced_keytab_delete()** routine deletes a key table (file) from a specific host and removes the associated entry from the **keytab** service of that host's **dced**. A key table is a file containing a list of server keys (passwords). This routine is used by management applications to remotely delete a key table.

To remove individual keys from a remote key table, use the **dced_keytab_remove_key()** routine. If you only want to make the key table inaccessible remotely (via **dced**), but not to delete it, use the **dced_entry_remove()** routine. This routine only removes the key table's **keytab** entry from **dced**.

Prior to calling the **dced_keytab_delete()** routine, the application must have established a valid **dced** binding handle to the **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_del_failed

db_s_iter_not_allowed

db_s_key_not_found

dced_s_bad_binding

dced_s_cant_remove_storage_file

dced_s_need_privacy

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_remove(3dce)**, **dced_keytab_remove_key(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_get_next_key

Purpose

Returns a key from a cached list and advances the cursor in the list

Synopsis

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

voiddced_keytab_get_next_key(
        dced_keytab_cursor_t cursor,
        dced_key_t ** key,
        error_status_t * status);
```

Parameters

Input/Output

cursor Specifies the cursor that points to a key, and returns the cursor advanced to the next key in the list.

Output

key Returns the current key to which the *cursor* points.

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

Description

The **dced_keytab_get_next_key()** routine obtains the current key to which the key-list cursor points. This routine is commonly used in a loop to traverse a key table's keys. The keys are returned in an undetermined order. Prior to using this routine in the loop, the application must call **dced_keytab_initialize_cursor()** to obtain the key list and establish the beginning of the cursor. When the application is finished traversing the key list, it should call **dced_keytab_release_cursor()** to release the resources allocated.

Management applications use **dced_keytab_get_next_key()** to remotely access a server's individual keys. Servers use **sec_key_mgmt_get_next_key()** to access their own local keys individually.

You can also use the **dced_object_read()** routine to read an entire key table. You might use **dced_object_read()** if your application needs to bind to and read data for other host services (**srvrconf**, **srvrexec**, or **hostdata**) in addition to data for the **keytab** service.

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

dced_s_no_more_entries

Related Information

Functions: **dced_keytab_initialize_cursor(3dce)** , **dced_keytab_release_cursor(3dce)** , **dced_object_read(3dce)**, **sec_key_mgmt_get_next_key(3sec)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_initialize_cursor

Purpose

Obtains a list of keys from a key table and sets a cursor at the beginning of the list

Synopsis

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

voiddced_keytab_initialize_cursor(
        dced_binding_handle_t dced_bh,
        uuid_t * keytab_uuid,
        dced_keytab_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

keytab_uuid Specifies the **keytab** entry **dced** associates with a key table.

Output

cursor Returns the cursor that is used to traverse the list of keys.

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

Description

The **dced_keytab_initialize_cursor()** routine obtains the complete list of keys from a remote key table and sets a cursor at the beginning of the cached list keys. In order to minimize the security risks of keys exposed to the network, the entire set of keys are encrypted and transferred in one remote procedure call rather than individually or in

chunks. The cursor is then used in subsequent calls to **dced_keytab_get_next_key()** to obtain individual keys. When the application is finished traversing the key list, it should call **dced_keytab_release_cursor()** to release the resources previously allocated.

Management applications use **dced_keytab_initialize_cursor()** and its associated routines to remotely access server keys. Servers use **sec_key_mgmt_initialize_cursor()** and its associated routines to manage their own keys locally.

Prior to calling the **dced_keytab_initialize_cursor()** routine, the application must have established a valid **dced** binding handle to the **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

dced_s_bad_binding

dced_s_need_privacy

dced_s_no_memory

dced_s_no_support

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_unauthorized

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_keytab_get_next_key(3dce)** , **dced_keytab_release_cursor(3dce)** , **sec_key_mgmt_initialize_cursor(3sec)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_release_cursor

Purpose

Releases the resources of a cursor that traverses a key table's list of keys (server passwords)

Synopsis

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

voiddced_keytab_release_cursor(
        dced_keytab_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input/Output

cursor Specifies the cursor for which resources are released.

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 **dced_keytab_release_cursor()** routine releases the cursor and resources initially set by the **dced_keytab_initialize_cursor()** routine and used by the **dced_keytab_get_next_key()** routine.

Prior to calling this routine, the application must have first established a valid **dced** binding handle by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()** , and then the application must have called the **dced_keytab_initialize_cursor()** routine.

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

dced_s_bad_binding

dced_s_no_support

Related Information

Functions: **dced_keytab_get_next_key(3dce)** , **dced_keytab_initialize_cursor(3dce)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_keytab_remove_key

Purpose

Removes a key (server password) from a specified key table on a specific host

Synopsis

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

voiddced_keytab_remove_key(
        dced_binding_handle_t dced_bh,
        uuid_t * keytab_uuid,
        dced_key_t * key,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **keytab** service on a specific host.

keytab_uuid Specifies the UUID **dced** maintains to identify the key table from which the key is to be removed.

key Specifies the key to be removed from the key table.

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 **dced_keytab_remove_key()** routine removes a key from a key table (file) on a specific host. The key table is specified with a **keytab** entry UUID from the host's **dced** . Management applications use **dced_keytab_remove_key()** to remotely remove

server keys from key tables. Typically, servers delete their own keys from their local key tables implicitly by calling **sec_key_mgmt_manage_key()**, or explicitly by calling **sec_key_mgmt_delete_key()**. Applications can delete an entire key table file using the **dced_keytab_delete()** routine.

Prior to calling this routine, the application must have established a valid **dced** binding handle to the **keytab** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

db_s_bad_index_type

db_s_key_not_found

dced_s_bad_binding

dced_s_need_privacy

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_unauthorized

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_keytab_delete(3dce)**, **sec_key_mgmt_delete_key(3sec)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_list_get

Purpose

Returns the list of data entries maintained by a **dced** service on a specific host

Synopsis

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

void dced_list_get(
        dced_binding_handle_t dced_bh,
        dced_entry_list_t * list,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

Output

list Returns a list of data entries for the service.

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

Description

The **dced_list_get()** routine obtains all the data entries for a **dced** service on a specific host. The list of data entries obtained is not the actual data. Each entry contains a UUID, name, description, and storage tag that describes where the data is located (for example, a filename or memory location). Call the **dced_list_release()** routine when your application is finished with the entry list to release resources allocated with **dced_list_get()** routine.

If a service's entry list is small, it may be efficient to obtain the entire list using the **dced_list_get()** routine, because this guarantees that the list is obtained

with one remote procedure call. However, to make your application scalable, use the **dced_initialize_cursor()** , **dced_entry_get_next()**, and **dced_release_cursor()** routines, because if an entry list is very large, it may be more efficient (or even necessary) to obtain the list in chunks with more than one remote procedure call.

Prior to calling this routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

In the following example, a **dced** binding is obtained from a service type and an existing RPC binding handle. The list of entries for the service is obtained with the **dced_list_get()** routine and each entry's name and description are displayed.

```
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh,
&status);
dced_list_get(dced_bh, &entries, &status);
for(i=0; i<entries.count; i++)
display(&entries);  /* application specific */
dced_list_release(dced_bh, &entries, &status);
dced_binding_free( dced_bh, &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

dced_s_bad_binding

dced_s_no_memory

dced_s_no_support

sec_acl_invalid_permission

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_initialize_cursor(3dce)**, **dced_list_release(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_list_release

Purpose

Releases the resources for a list of entries of a **dced** service

Synopsis

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

voiddced_list_release(
        dced_binding_handle_t dced_bh,
        dced_entry_list_t * list,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

InputOutput

list Specifies a list of data entries for the service.

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 **dced_list_release()** routine releases the resources allocated for a list of data entries previously retrieved by the **dced_list_get()** routine.

Prior to calling this routine, the application must have first established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine, and then the application must have called the **dced_list_get()** routine.

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

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_list_get(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_object_read

Purpose

Reads a data item of a **dced** service on a specific host

Synopsis

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

voiddced_object_read(
        dced_binding_handle_t dced_bh,
        uuid_t * entry_uuid,
        void ** data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

entry_uuid Specifies the UUID of the **dced** service's data entry associated with the data item.

Output

data Returns the data read. The data returned is one of the following structures, depending on the service:

Service	Data Type Returned
hostdata	**sec_attr_t**
srvrconf	**server_t**
srvrexec	**server_t**
keytab	**dced_key_list_t**

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

Description

The **dced_object_read()** routine reads the data for a specified entry of a **dced** service. When the application is done with the data, it should call the **dced_objects_release()** routine with a value of 1 for the *count* parameter.

The valid services for which you can read data include **hostdata** , **srvrconf**, **srvrexec**, and **keytab**. These host services each have a list of data entries maintained by **dced**. The entries do not contain the actual data, but contain the data's identity and an indicator of the location of the data item. The **hostdata** service also has the **dced_hostdata_read()** routine to read data, and the **keytab** service has a series of routines that traverse over the keys in a key table. (See the **dced_keytab_initialize_cursor()** routine.) The **secval** and **endpoint** services do not have data items to read with this routine.

Applications can also read the data for all entries of a service using one call to **dced_objects_read_all()**.

Prior to reading the actual data, an application commonly obtains the entries to read using the series of cursor routines that begin with **dced_entry_initialize_cursor()**.

Prior to calling the **dced_object_read()** routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example creates a **dced** binding to a **dced** service based on a service type and host in an RPC binding handle. The example then obtains the service's entry list and reads the data associated with each entry.

```
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh,
&status);
dced_list_get(dced_bh, &entries, &status);
for(i=0; i<entries.count; i++) {
dced_object_read(dced_bh, &entries.list[i].id, &data, &status);
```

```
.
.
.
dced_objects_release(dced_bh, 1, data, &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

db_s_bad_index_type

db_s_key_not_found

dce_cf_e_file_open

dce_cf_e_no_match

dce_cf_e_no_mem

dced_s_bad_binding

dced_s_need_privacy

dced_s_no_memory

dced_s_no_support

dced_s_not_found

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_unauthorized

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_hostdata_read(3dce)**, **dced_initialize_cursor(3dce)**, **dced_keytab_initialize_cursor(3dce)**, **dced_objects_read_all(3dce)**, **dced_objects_release(3dce)** .

Books: *DCE 1.2.2 Application Development Guide*.

dced_object_read_all

Purpose

Reads all the data for a service of **dced** on specific host

Synopsis

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

voiddced_object_read_all(
        dced_binding_handle_t dced_bh,
        unsigned32 * count,
        void ** data_list,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

Output

count Returns the count of the number of data items read.

data_list Returns the list of data items read. The data returned is an array of one of the following types, depending on the service:

Service	Data Type of Array Returned
hostdata	**sec_attr_t**
srvrconf	**server_t**
srvrexec	**server_t**
keytab	**dced_key_list_t**

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

Description

The **dced_object_read_all()** routine reads all the data for a specified host service on a specific host. When the application is done with the data, it should call the **dced_objects_release()** routine. Applications can also read individual data objects for a service using the **dced_object_read()** routine.

The valid services for which you can read data include **hostdata** , **srvrconf**, **srvrexec**, and **keytab**.

Prior to calling the **dced_object_read_all()** routine, the application must have established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

Examples

The following example reads and displays all the data for a particular **dced** service.

```
dced_binding_handle_t   dced_bh;
dced_string_t           host_service;
void                    *data_list;
unsigned32              count;
error_status_t          status;
dced_binding_create(host_service, dced_c_binding_syntax_default,
&dced_bh, &status);
dced_object_read_all(dced_bh, &count, &data_list, &status);
display(host_service, count, &data_list); /* application specific */
dced_objects_release(dced_bh, count, data_list, &status);
dced_binding_free( dced_bh, &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

db_s_bad_index_type

db_s_key_not_found

dce_cf_e_file_open

dce_cf_e_no_match

dce_cf_e_no_mem

dced_s_bad_binding

dced_s_need_privacy

dced_s_no_memory

dced_s_no_support

dced_s_not_found

rpc_s_binding_has_no_auth

rpc_s_invalid_binding

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

sec_key_mgmt_e_authn_invalid

sec_key_mgmt_e_key_unavailable

sec_key_mgmt_e_unauthorized

sec_s_no_memory

uuid_s_bad_version

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_object_read(3dce)**, **dced_objects_release(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_objects_release

Purpose

Releases the resources allocated for data read from a **dced** service

Synopsis

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

voiddced_objects_release(
        dced_binding_handle_t dced_bh,
        unsigned32 count,
        void * data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for a **dced** service on a specific host.

count Specifies the number of data items previously read and now to be released.

Input/Output

data Specifies the data for which resources are released.

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 **dced_objects_release()** routine releases the resources allocated when data for **dced** is read. Applications should call **dced_objects_release()** when finished with data allocated by the following **dced** API routines:

- **dced_object_read_all()**
- **dced_object_read()**
- **dced_hostdata_read()**

If the data being released was read by using **dced_object_read_all()** , the *count* returned from this routine is used as input to the **dced_objects_release()** routine. If the data being released was read by using **dced_object_read()** or **dced_hostdata_read()**, the *count* value required as input for the **dced_objects_release()** routine is **1**.

Examples

In the following example, a binding is created to a **dced** service on some host for a service that stores data, and the service's entry list is obtained. For each entry, the data is read, displayed, and released.

```
dced_binding_handle_t   dced_bh;
dced_entry_list_t       entries;
unsigned32              i;
void                    *data;
error_status_t          status;
dced_binding_create(host_service, dced_c_binding_syntax_default,
&dced_bh, &status);
dced_list_get(dced_bh, &entries, &status);
for(i=0; i<entries.count; i++) {
dced_object_read(dced_bh, &(entries.list[i].id), &data, &status);
display(host_service, 1, &data);      /* application specific */
dced_objects_release(dced_bh, 1, data, &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

dced_s_bad_binding

dced_s_no_support

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_hostdata_read(3dce)**, **dced_object_read(3dce)**, **dced_object_read_all(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_release_cursor

Purpose

Releases the resources of a cursor which traverses a **dced** service's list of entries

Synopsis

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

voiddced_release_cursor(
        dced_cursor_t * cursor,
        error_status_t * status);
```

Parameters

Input/Output

cursor Specifies the cursor for which resources are released.

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 **dced_release_cursor()** routine releases the resources of a cursor initially set by the **dced_initilalize_cursor()** routine and used by the **dced_entry_get_next()** routine.

Prior to calling this routine, the application must have first established a valid **dced** binding handle by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine, and then the application must have called the **dced_initialize_cursor()** routine.

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

Related Information

Functions: **dced_binding_create(3dce)** , **dced_binding_from_rpc_binding(3dce)** , **dced_entry_get_next(3dce)**, **dced_initialize_cursor(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_secval_start

Purpose

Starts the security validation service of a specific host's **dced**

Synopsis

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

voiddced_secval_start(
        dced_binding_handle_t dced_bh,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **secval** service on a specific host.

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 **dced_secval_start()** routine starts the security validation service of a specific host's **dced**. This routine is typically used by management applications.

The security validation service (**secval**) has two major functions:

- Maintains a login context for the host's *self* identity.
- Validates and certifies to applications (usually login programs) that the DCE security daemon (**secd**) is legitimate.

The **secval** program is commonly started by default when **dced** starts. See the **dced_secval_stop()** routine for a discussion of when to use the combination of **dced_secval_stop()** and **dced_secval_start()**.

Prior to calling this routine, the application must have established a valid **dced** binding handle to the **secval** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

dced_s_bad_binding

dced_s_sv_already_enabled

sec_acl_invalid_permission

Related Information

Commands: **dced(8dce)**, the **secval(8dce)** object of **dcecp**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_secval_stop(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_secval_status

Purpose

Indicates whether or not a specific host's security validation service of **dced** is running

Synopsis

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

voiddced_secval_status(
        dced_binding_handle_t dced_bh,
        boolean32 * secval_active,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **secval** service on a specific host.

Output

secval_active
Returns a value of TRUE if the security validation service is running and FALSE if it is not running.

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

Description

The **dced_secval_status()** routine sets a parameter to TRUE or FALSE depending on whether the security validation service has been activated or deactivated.

Prior to calling this routine, the application must have established a valid **dced** binding handle to the **secval** service by calling either the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine.

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

dced_s_bad_binding

Related Information

Commands: **dced(8dce)**, the **secval(8dce)** object of **dcecp**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_secval_start(3dce)**, **dced_secval_stop(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_secval_stop

Purpose

Stops the security validation service of a specific host's **dced**

Synopsis

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

voiddced_secval_stop(
        dced_binding_handle_t dced_bh,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **secval** service on a specific host.

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 **dced_secval_stop()** routine stops the security validation service (**secval**) of a specific host's **dced**. This routine is typically used by management applications.

The **secval** service is commonly started by default when **dced** starts. The main use of **dced_secval_stop()** and **dced_secval_start()** is to force a refresh of the host principal credentials. This is the only way to force certain registry changes made by the host principal (such as **groupset** membership) to be seen by processes running on the host.

You can easily stop and then start the **secval** service with the following operations:

dcecp -c secval deactivate
dcecp -c secval activate

It is not a good idea to remove the machine principal **self** credentials for an extended period of time because processes running as **self** will fail in their attempts to perform authenticated operations.

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

dced_s_bad_binding

dced_s_sv_not_enabled

sec_acl_invalid_permission

Related Information

Commands: **dced(8dce)**, the **secval(8dce)** object of **dcecp**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_secval_start(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_secval_validate

Purpose

Validates that the **secd** used by a specific host is legitimate

Synopsis

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

voiddced_secval_validate(
        dced_binding_handle_t dced_bh,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **secval** service on a specific host.

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 **dced_secval_validate()** routine validates and certifies for a specific host that the DCE security daemon (**secd**)is legitimate. Typically, a login program uses the security validation service when it uses the security service's login API (routines that begin with **sec_login**). However, if a management application trusts some remote host, it can use **dced_secval_validate()** to validate **secd**, without logging in to the host.

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

dced_s_bad_binding

ept_s_not_registered

rpc_s_comm_failure

rpc_s_invalid_binding

rpc_s_rpcd_comm_failure

rpc_s_wrong_kind_of_binding

sec_login_s_no_current_context

Related Information

Commands: **dced(8dce)**, the **secval(8dce)** object of **dcecp**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_secval_start(3dce)**, **sec_login_*** **(3sec)** API.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_create

Purpose

Creates a DCE server's configuration data for the host's **dced**

Synopsis

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

void dced_server_create(
        dced_binding_handle_t dced_bh,
        server_t * conf_data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrconf** service on a specific host.

Input/Output

conf_data Specifies the configuration data for the server. The **dced_intro(3dce)** reference page describes the **server_t** structure.

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 **dced_server_create()** routine creates a server's configuration data. This routine is used by management installation applications to remotely (or locally) establish the data used to control how a DCE server starts. However, this routine does not create the program or start it. Since this activity is typically part of a server's installation, you can also use **dcecp**'s **server create** operation.

Management applications use the **dced_object_read()** routine to read the configuration data.

Prior to calling **dced_server_create()**, the application must have established a valid **dced** binding handle to the **srvrconf** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()**.

Examples

The following example shows how to fill in some of the fields of a **server_t** structure and then create the configuration in **dced**.

```
dced_binding_handle_t dced_bh;
server_t              conf;
error_status_t        status;
dced_binding_create("srvrconf@hosts/katharine",
dced_c_binding_syntax_default,
&dced_bh,
&status);
/* setup a server_t structure */
uuid_create(&conf.id, &status);
conf.name           = (dced_string_t)"application";
conf.entryname      = (dced_string_t)"/.:/development/new_app";
conf.services.count = 1;
/* service_t structure(s) */
conf.services.list = malloc(conf.services.count * sizeof(service_t));
rpc_if_inq_id(application_v1_0_c_ifspec,
&(conf.services.list[0].ifspec), &status);
conf.services.list[0].ifname     = (dced_string_t)"application";
conf.services.list[0].annotation = (dced_string_t)"A new application";
conf.services.list[0].flags      = 0;
/* server_fixedattr_t structure */
conf.fixed.startupflags = server_c_startup_explicit |
server_c_startup_on_failure;
conf.fixed.flags = 0;
conf.fixed.program = (dced_string_t)"/usr/users/bin/new_app";
dced_server_create(dced_bh, &conf, &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

db_s_bad_header_type

db_s_bad_index_type

db_s_iter_not_allowed

db_s_key_not_found

db_s_readonly

db_s_store_failed

dced_s_already_exists

dced_s_bad_binding

dced_s_name_missing

sec_acl_invalid_permission

Related Information

dcecp objects: **server(8dce)**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_object_read(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_delete

Purpose

Deletes a DCE server's configuration data from **dced**

Synopsis

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

void dced_server_delete(
        dced_binding_handle_t dced_bh,
        uuid_t * conf_uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrconf** service on a specific host.

conf_uuid Specifies the UUID that **dced** uses to identify the server's configuration data to be deleted.

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 **dced_server_delete()** routine deletes a server's configuration data from the server's **dced**. This routine removes a server from DCE control by making it incapable of starting via **dced**. The routine does not delete the program from disk nor does it affect the server if the server is currently running.

Prior to using **dced_server_delete()**, the server configuration data must be created by an administrator using the **dcecp server create** operation or by an application using **dced_server_create()**.

Prior to calling **dced_server_delete()**, the application must have established a valid **dced** binding handle to the **srvrconf** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()**.

Examples

In the following example, a **dced** binding is created to the server configuration service on a host, and then an inquiry is made as to the UUID associated with a particular server. The **dced_server_delete()** routine is then used to delete the configuration.

```
dced_binding_handle_t dced_bh;
dced_string_t         server_name;
uuid_t                srvrconf_id;
error_status_t        status;
name_server(&server_name); /* application specific */
dced_binding_create("srvrconf@hosts/katharine",
dced_c_binding_syntax_default, &dced_bh, &status);
dced_inq_id(dced_bh, server_name, &srvrconf_id, &status);
dced_server_delete(dced_bh, &srvrconf_id, &status);
dced_binding_free(dced_bh, &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

db_s_bad_index_type

db_s_del_failed

db_s_iter_not_allowed

dced_s_bad_binding

dced_s_not_found

sec_acl_invalid_permission

Related Information

dcecp Objects: **server(8dce)**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_server_create(3dce)**, **dced_server_modify_attributes(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_disable_if

Purpose

Disables a service (RPC interface) provided by a specific server on a specific host

Synopsis

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

voiddced_server_disable_if(
        dced_binding_handle_t dced_bh,
        uuid_t * exec_uuid,
        rpc_if_id_t * interface,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrexec** service on a specific host.

exec_uuid Specifies the UUID that **dced** uses to identify the running server.

interface Specifies the RPC interface identifier that represents the service to be disabled. The interface identifier is generated when **idl** compiles an interface definition file. The interface identifier is an *rpc_if_id_t* structure that contains the interface UUID (**uuid**) of type **uuid_t**, and numbers of type **unsigned16** representing the major (*vers_major*) and minor (*vers_minor*) version numbers for the interface.

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 **dced_server_disable_if()** routine disables a service provided by a server on a specific host. A service is represented by an RPC interface identifier. Management applications use this routine to remotely disable an interface so it is inaccessible by clients, without completely stopping the entire server.

When a server starts and initializes itself, it must call the **dce_server_register()** routine to enable all of its services. The server can then disable its own individual services by using **dce_server_disable_service()**. This routine unregisters the service's interface from the RPC runtime and marks the interface as disabled in the endpoint map. As an alternative, a management application can use **dced_server_disable_if()** to disable individual services. However, this routine only affects the endpoint map in **dced** by marking the interface as disabled and does not affect the server's runtime.

A management application can reenable a service again by calling the **dced_server_enable_if()** routine. (Servers reenable their own services using the **dce_server_enable_if()** routine.)

Prior to calling **dced_server_disable_if()**, the application must have established a valid **dced** binding handle to the **srvrexec** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()** .

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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_readonly

db_s_store_failed

dced_s_bad_binding

dced_s_not_found

sec_acl_invalid_permission

Related Information

dcecp Objects: **server(8dce)**.

Functions: **dce_server_disable_if(3dce)**, **dce_server_enable_if(3dce)** , **dce_server_register(3dce)**, **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)**, **dced_server_enable_if(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_enable_if

Purpose

Enables a service (RPC interface) of a specific server on a specific host

Synopsis

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

voiddced_server_enable_if(
        dced_binding_handle_t dced_bh,
        uuid_t * exec_uuid,
        rpc_if_id_t * interface,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrexec** service on a specific host.

exec_uuid Specifies the UUID that **dced** uses to identify the running server.

interface Specifies the RPC interface identifier that represents the service to be enabled. The interface identifier is generated when **idl** compiles an interface definition file. The interface identifier is a structure that contains the interface UUID (**interface->uuid**), and the major (**interface->vers_major**) and minor (**interface->vers_minor**) version numbers for the interface.

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 **dced_server_enable_if()** routine enables a service (or reenables a previously disabled service) that a specific server provides. Management applications use this routine. A service is represented by an RPC interface identifier.

When a server starts and initializes itself, it typically calls the **dce_server_register()** routine to enable all of its services. The services can then be disabled and reenabled, as needed. A server enables and disables its own services by using the routines **dce_server_enable_service()** and **dce_server_disable_service()** . A management application enables and disables a remote server's service using the routines **dced_server_enable_if()** and **dced_server_disable_if()**.

The **dce_server*** routines affect both the RPC runtime and the local endpoint map by registering (or unregistering) with the runtime and setting a flag for the interface in the the endpoint map as enabled (or disabled). The **dced_server_enable_if()** and **dced_server_disable_if()** routines affect only the remote endpoint map by setting the flag.

Prior to calling **dced_server_enable_if()**, the application must have established a valid **dced** binding handle to the **srvrexec** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()**.

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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_readonly

db_s_store_failed

dced_s_bad_binding

dced_s_not_found

sec_acl_invalid_permission

Related Information

dcecp Objects: **server(8dce)**.

Functions: **dce_server_disable_if(3dce)**, **dce_server_enable_if(3dce)** , **dce_server_register(3dce)**, **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)**, **dced_server_disable_if(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_modify_attributes

Purpose

Modifies attributes for a DCE server's configuration data

Synopsis

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

voiddced_server_modify_attributes(
        dced_binding_handle_t dced_bh,
        uuid_t * conf_uuid,
        dced_attr_list_t * data,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrconf** service on a specific host.

conf_uuid Specifies the UUID that **dced** uses to identify a server's configuration data to be modified.

data Specifies the attributes to be modified.

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 **dced_server_modify_attributes()** routine replaces a server's attributes of its configuration data maintained by **dced** on a specific host. This routine is typically called after a configuration is created with the **dced_server_create()** routine.

A server's configuration is manipulated in a **server_t** data structure, and the **dced_server_modify_attributes()** routine affects only the **attributes** member of this structure. To change other server configuration data, you must first delete the configuration by using **dced_server_delete()**, and then create the configuration again by using **dced_server_create()** .

Prior to calling **dced_server_modify_attributes()** , the application must have established a valid **dced** binding handle to the **srvrconf** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()** .

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

db_s_bad_index_type

db_s_iter_not_allowed

db_s_readonly

db_s_store_failed

dced_s_bad_binding

dced_s_not_found

sec_acl_invalid_permission

Related Information

dcecp Objects: **server(8dce)**.

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_object_read(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*

dced_server_start

Purpose

Starts a DCE-configured server on a specified host

Synopsis

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

void dced_server_start(
        dced_binding_handle_t dced_bh,
        uuid_t * conf_uuid,
        dced_attr_list_t * attributes,
        uuid_t * exec_uuid,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrconf** service on a specific host.

conf_uuid Specifies the UUID that **dced** uses to identify the server to start. If the value input is that of a server that is already running, **dced** starts a new instance.

attributes Specifies the configuration attributes to use to start the server. If the value is NULL, the default configuration defined in **dced** is used.

Input/Output

exec_uuid Specifies a new UUID for **dced** to use to identify the running server. If a nil UUID is input, a new UUID is created and returned. If the value input is that of a server that is already running, **dced** starts a new instance and returns a new 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 **dced_server_start()** routine starts DCE-configured servers on a specific remote host (or the local host). The configuration data is stored in an object in the **srvrconf** service of **dced**. When the server starts, **dced** uses the server configuration object and creates a server execution object in the **srvrexec** service. A server execution object consists of data that describes the executing server.

Management applications create the configuration data by using the **dced_server_create()** and the **dced_object_read()** routine to read the configuration or execution data.

Prior to calling **dced_server_start()**, the application must have established a valid **dced** binding handle to the **srvrconf** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()**.

Examples

The following example starts a configured server using a nil UUID as input for the executing server.

```
dced_binding_handle_t conf_bh;
dced_string_t         server_name;
uuid_t                srvrconf_id, srvrexec_id;
error_status_t        status;
dced_binding_create("srvrconf@hosts/patrick",
dced_c_binding_syntax_default,
&conf_bh,
&status);
dced_inq_id(conf_bh, server_name, &srvrconf_id, &status);
uuid_create_nil(&srvrexec_id, &status);
dced_server_start(conf_bh, &srvrconf_id, NULL, &srvrexec_id,
&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

db_s_bad_header_type

db_s_iter_not_allowed

db_s_key_not_found

db_s_readonly

db_s_store_failed

dced_s_bad_binding

dced_s_no_support

dced_s_not_found

dced_s_sc_cant_fork

dced_s_sc_invalid_attr_type

dced_s_sc_open_file_failed

sec_acl_invalid_permission

uuid_s_bad_version

Related Information

Commands: **server(8dce)** .

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)** , **dced_server_create(3dce)**, **dced_server_stop(3dce)**.

Books: *DCE 1.2.2 Application Development Guide*.

dced_server_stop

Purpose

Stops a DCE-configured server running on a specific host

Synopsis

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

void dced_server_stop(
        dced_binding_handle_t dced_bh,
        uuid_t * exec_uuid,
        srvrexec_stop_method_t method,
        error_status_t * status);
```

Parameters

Input

dced_bh Specifies the **dced** binding handle for the **srvrexec** service on a specific host.

exec_uuid Specifies a UUID that **dced** uses to identify the running server. If the value input is **dced_g_uuid_all_servers**, **dced** attempts to stop all the DCE servers running on that host.

method Specifies the method **dced** uses to stop a server. A method is represented by one of the following values:

srvrexec_stop_rpc
Uses the **rpc_mgmt_stop_server_listening()** routine. This is the cleanest way to stop a server, because it waits for outstanding remote procedure calls to finish before making the server's **rpc_server_listen()** routine return.

srvrexec_stop_soft
Uses a soft local host mechanism (such as the **TERM** signal in UNIX)

srvrexec_stop_hard
Uses a hard local host mechanism (such as the **KILL** signal in UNIX)

srvrexec_stop_error
Uses a mechanism that saves the program state (such as the **ABORT** signal in UNIX)

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 **dced_server_stop()** routine stops DCE-configured servers on specific hosts. When the server is completely stopped and no longer a running process, **dced** deletes the associated execution data it maintained.

Administrators use the **dcecp** operations **server create** and **server start** to configure and start a server, and management applications use the associated **dced_server_create()** and **dced_server_start()** routines.

Prior to calling **dced_server_stop()**, the application must have established a valid **dced** binding handle to the **srvrexec** service by calling either **dced_binding_create()** or **dced_binding_from_rpc_binding()**.

Cautions

Using the value **dced_g_uuid_all_servers** for the *exec_uuid* parameter causes **dced** to shutdown all servers *including itself*.

Examples

The following example obtains **dced** binding handles to the server configuration and execution services of **dced** on the host **patrick**. The example then checks to see if the server is running by seeing if **dced** has a UUID and entry for the executing server. However, the server may be in the process of starting up or stopping, so the example also checks to be sure the instance UUID of the running server matches the UUID of the configuration for that server. If there is a match, the server is running. Finally, the

example stops the server by calling **dced_server_stop()** with the **srvrexec_stop_rpc** parameter.

```
dced_binding_handle_t conf_bh, exec_bh;
dced_string_t         server_name;
void                  *data;
server_t              *exec_ptr;
uuid_t                srvrconf_id, srvrexec_id;
error_status_t        status;
.
.
.
dced_binding_create("srvrconf@hosts/patrick",
dced_c_binding_syntax_default,
&conf_bh,
&status);
dced_binding_create("srvrexec@hosts/patrick",
dced_c_binding_syntax_default,
&exec_bh,
&status);
/* is server running? */
dced_inq_id(exec_bh, server_name, &srvrexec_id, &status);
/* also check to be sure server is not coming up or going down */
dced_object_read(exec_bh, &srvrexec_id, &data, &status);
exec_ptr = (server_t*)data;
dced_inq_id(conf_bh, server_name, &srvrconf_id, &status);
if(uuid_equal(&srvrconf_id,
&exec_ptr->exec_data.tagged_union.running_data.instance,
&status) ) {
dced_server_stop(exec_bh, &srvrexec_id, srvrexec_stop_rpc, &status);
}
dced_objects_release(exec_bh, 1, data, &status);
dced_binding_free(conf_bh, &status);
dced_binding_free(exec_bh, &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

dced_s_bad_binding

dced_s_no_support

dced_s_not_found

rpc_s_binding_incomplete

rpc_s_comm_failure

rpc_s_invalid_binding

rpc_s_mgmt_op_disallowed

rpc_s_unknown_if

rpc_s_wrong_kind_of_binding

sec_acl_invalid_permission

uuid_s_bad_version

Related Information

dcecp Objects: **server(8dce)**.

Functions: **dced_binding_create(3dce)**, **dce d_binding_from_rpc_binding(3dce)** , **dced_server_create(3dce) dced_server_start(3dce)**, **rpc_mgmt_stop_server_listening(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide*.

DCE_SVC_DEBUG

Purpose

Macro to output a serviceability debug message

Synopsis

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

DCE_SVC_DEBUG((
        dce_svc_handle_t handle,
        const unsigned32 table_index,
        unsigned32 debug_level,
        char * format,
        ...));
```

Parameters

Input

handle	The caller's serviceability handle.
table_index	The message's subcomponent name (defined in the **sams** file).
debug_level	Serviceability debug level for the message.
format	The message string.
. . .	Format arguments, if any.

Description

The **DCE_SVC_DEBUG** macro is used to generate debugging output. Because it is a macro that takes a variable number of arguments, the entire parameter list must be enclosed in two sets of parentheses. The *handle* and *table_index* parameters are as described for **dce_svc_printf()**.

In contrast to the normal operation of the serviceability interface, **DCE_SVC_DEBUG** requires the caller to specify the message as a string literal in the call, rather than by defining it in the application's **sams** file specifying the message by a message ID.

The *debug_level* argument indicates the level of detail associated with this message and must be in the range **svc_c_debug1** to **svc_c_debug9**.

Thus the value of *debug_level* associates the message with one of nine levels, and whether or not the message is actually generated at run time will depend on what debugging level has been set for the application. The level can be set by the application itself by a call to **dce_svc_debug_set_levels()** or **dce_svc_debug_routing()**. The level can also be set by the value of an environment variable or a serviceability routing file. See **svcroute(5dce)** for further information.

The significance of the various levels is application-defined, but in general the higher levels (numbers) imply more detail in debugging output.

The *format* and **. . .** parameters are passed directly to **fprintf()** or its equivalent.

Related Information

Functions: **dce_svc_debug_routing(3dce)**, **dce_svc_debug_set_levels(3dce)**, **dce_svc_printf(3dce)**, **dce_svc_routing(3dce)**.

Files: **svcroute(5dce)**.

DCE_SVC_DEBUG_ATLEAST

Purpose

Macro to test a component's serviceability debug level

Synopsis

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

DCE_SVC_DEBUG_ATLEAST(
        dce_svc_handle_t handle,
        const unsigned32 table_index,
        unsigned32 debug_level);
```

Parameters

Input

handle The caller's serviceability handle.

table_index The subcomponent name (defined in the **sams** file) whose debug level is being tested.

debug_level The debug level being tested.

Description

If serviceability debug code was enabled (by defining **DCE_DEBUG**)during compilation, the **DCE_SVC_DEBUG_ATLEAST** and **DCE_SVC_DEBUG_IS** macros can be used to test the debug level of a subcomponent (specified by *table_index*) for the specified *handle*. **DCE_SVC_DEBUG_ATLEAST** tests whether the debug level is at least at the specified level. **DCE_SVC_DEBUG_IS** tests for an exact match with the specified level. In either case, the specified level should be a number between 1 and 9.

Related Information

Functions: **DCE_SVC_DEBUG(3dce)**, **DCE_SVC_DEBUG_IS(3dce)**, **DCE_SVC_LOG(3dce)**.

DCE_SVC_DEBUG_IS

Purpose

Macro to test a component's serviceability debug level

Synopsis

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

DCE_SVC_DEBUG_IS(
        dce_svc_handle_t handle,
        const unsigned32 table_index,
        unsigned32 debug_level);
```

Parameters

Input

handle The caller's serviceability handle.

table_index The name of the subcomponent name (defined in the **sams** file) whose debug level is to be tested.

debug_level The serviceability debug level being tested.

Description

If serviceability debug code was enabled (by defining **DCE_DEBUG**)during compilation, the **DCE_SVC_DEBUG_ATLEAST** and **DCE_SVC_DEBUG_IS** macros can be used to test the debug level of a subcomponent (specified by *table_index*) for the specified *handle*. **DCE_SVC_DEBUG_ATLEAST** tests whether the debug level is at least at the specified level. **DCE_SVC_DEBUG_IS** tests for an exact match with the specified level. In either case, the specified level should be a number between 1 and 9.

Related Information

Functions: **DCE_SVC_DEBUG(3dce)**, **DCE_SVC_DEBUG_ATLEAST(3dce)**, **DCE_SVC_LOG(3dce)**.

DCE_SVC_DEFINE_HANDLE

Purpose

Macro to create a serviceability handle

Synopsis

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

DCE_SVC_DEFINE_HANDLE(
        dce_svc_handle_t handle,
        dce_svc_subcomp_t * table,
        const idl_char * component_name);
```

Parameters

Input

table A message table structure (defined in a header file generated by **sams** during compilation).

component_name
The serviceability name of the component, defined in the **component** field of the **sams** file.

Output

handle A serviceability handle structure that will be filled in by the macro.

Description

There are two ways to register a serviceability table preparatory to using the serviceability interface in an application. The first is to create a global variable using the **DCE_SVC_DEFINE_HANDLE** macro. The first parameter is the serviceability handle, the second is a pointer to the component's message table, and the third is the name of the serviceability component (application). The macro creates a skeleton

variable that will be completed the first time the handle is used. This can be useful when writing library code that has no explicit initialization routine.

The second method is to call the **dce_svc_register()** routine.

Related Information

Functions: **dce_svc_register(3dce)**.

DCE_SVC_LOG

Purpose

Macro to output a binary form of a serviceability debug message

Synopsis

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

DCE_SVC_LOG((
        dce_svc_handle_t handle,
        const unsigned32 table_index,
        unsigned32 debug_level,
        const unsigned32 messageid,
        char * format,
        . . .));
```

Parameters

Input

handle	The caller's serviceability handle.
table_index	The message's subcomponent name (defined in the **sams** file).
debug_level	Serviceability debug level for the message.
messageid	A message ID, defined in the message's **code** field in the **sams** file.
format	A message format specifier string (used if *messageid* cannot be found).
. . .	Any format arguments for the message string.

Description

The **DCE_SVC_LOG** macro is used to generate debugging output based on a message defined in an application's **sams** file (in this respect it is unlike **DCE_SVC_DEBUG**, in which the message is specified as a literal string parameter). Because it is a macro that takes a variable number of arguments, the entire parameter list must be enclosed

in two sets of parentheses. The *handle* and *table_index* parameters are as described for **dce_svc_printf()**.

The message can be specified in either one of two ways: by *messageid*, identifying a message defined in the normal way in the application's **sams** file; or as a string literal paramater (*format*). The *format* string is used only if the specified *messageid* cannot be found.

DCE_SVC_LOG generates a record in the serviceability binary format, not a conventional serviceability message as such. The complete message text is not normally written; instead, only the message ID (the *messageid* specified in the macro parameter), and its format arguments (if any) are written. When the binary log is read (see **svcdumplog(8dce)**), the text of the message is reconstructed from the application's installed message catalog. However, if the original message was generated from the *format* argument, then the entire message text is written to the binary record.

The *debug_level* argument indicates the level of detail associated with the message and must be in the range **svc_c_debug1** to **svc_c_debug9**.

Thus the value of *debug_level* associates the message with one of nine levels, and whether or not the message is actually generated at run time will depend on what debugging level has been set for the application. The level can be set by the application itself by a call to **dce_svc_debug_set_levels()** or **dce_svc_debug_routing()**. The level can also be set by the value of an environment variable or a serviceability routing file. See **svcroute(5dce)** for further information.

The significance of the various levels is application-defined, but in general the higher levels (numbers) imply more detail in debugging output.

Related Information

Functions: **DCE_SVC_DEBUG(3dce)**, **DCE_SVC_DEBUG_ATLEAST(3dce)**, **DCE_SVC_DEBUG_IS(3dce)**.

svcroute

Purpose

Routing control file for DCE serviceability messages

Description

The file **svc/routing** specifies routing for serviceability messages. The default location for **svc/routing** is the DCE local data directory (typically **/opt/dcelocal/var**). The environment variable **DCE_SVC_ROUTING_FILE**, if set, specifies a different location for the file.

The file consists of a series of text lines. Blank lines and lines that begin with a **#** (number sign) character are ignored when the file's contents are interpreted. All other lines must consist of either three or four fields, each separated by a **:** (colon). Leading whitespace is ignored.

Lines consisting of three fields specify routing for nondebug serviceability messages. Their format is as follows:

*sev***:***out_form***:***dest*[**;***out_form***:***dest* . . .] [**GOESTO:**{*sev* | *comp*}]

The *sev* (severity) field specifies one of the following message severities:

FATAL Fatal error exit: An unrecoverable error (such as database corruption) has occurred and will probably require manual intervention to be corrected. The program usually terminates immediately after such an error.

ERROR Error detected: An unexpected event that is nonterminal (such as a timeout), or is correctable by human intervention, has occurred. The program will continue operation, although some functions or services may no longer be available. This severity level may also be used to indicate that a particular request or action could not be completed.

WARNING Correctible error: An error occurred that was automatically corrected (for example, a configuration file was not found, and default values were used instead). This severity level may also be used to indicate a condition that *may* be an error if the effects are undesirable (for example, removing all files as a side-effect of removing a nonempty directory). This severity level may also be used to indicate a condition that, if not

corrected, will eventually result in an error (for example, a printer's running low on paper).

NOTICE Informational notice: A significant routine major event has occurred; for example, a server has started.

NOTICE_VERBOSE
Verbose information notice: A significant routine event has occurred; for example, a directory entry was removed.

The *out_form* (output form) field specifies how the messages of a given severity level should be processed, and must be one of the following:

BINFILE Write these messages as a binary log entry to the specified file.

TEXTFILE Write these messages as human-readable text.

FILE Equivalent to **TEXTFILE**.

DISCARD Do not record messages of this severity level.

STDOUT Write these messages as human-readable text to standard output.

STDERR Write these messages as human-readable text to standard error.

Files written as **BINFILE**s can be read and manipulated with a set of log file functions (for more information, see **dce_svc_log_open()** and **dce_svc_log_get()**), or by the **svcdumplog** command (see **svcdumplog(1dce)**).

The *out_form* specifier may be followed by a two-number specifier of the form

.gens.count

where

gens is an integer that specifies the number of files (that is, generations) that should be kept

count is an integer specifying how many entries (that is, messages) should be written to each file

The multiple files are named by appending a dot to the simple specified name, *dest*, followed by the current generation number. When the number of entries in a file reaches the maximum specified by *count*, the file is closed, the generation number is incremented, and the next file is opened. When the maximum generation number files have been created and filled, the generation number is reset to 1, and a new file with that number is created and written to (thus overwriting the already-existing file

with the same name), and so on, as long as messages are being written. Thus the files wrap around to their beginning, and the total number of log files never exceeds *gens*, although messages continue to be written as long as the program continues writing them. Note that when a program starts, the generation starts at 1.

The *dest* (destination) field specifies where the message should be sent, and is a pathname. The field can be left blank if the *out_form* specified is **DISCARD**, **STDOUT**, or **STDERR**. The field can also contain a **%ld** string in the filename which, when the file is written, will be replaced by the process ID of the program that wrote the message(s). Filenames may *not* contain colons or periods.

Multiple routings for the same severity level can be specified by simply adding the additional desired routings as semicolon-separated *out_form***:***dest* strings.

For example, the following strings specify that

- Fatal error messages should be sent to the console.
- Warnings should be discarded.
- Notices should be written both to standard error and as binary entries in files located in the **/tmp** directory. No more than 50 files should be written, and there should be no more than 100 messages written to each file. The files will have names of the form **/tmp/log***process_id***.***n*, where *process_id* is the process ID of the program originating the messages, and *n* is the generation number of the file (expressed with only as many digits as needed).

```
FATAL:TEXTFILE:/dev/console
WARNING:DISCARD:–
NOTICE:STDERR:-;BINFILE.50.100:/tmp/log%ld
```

The **GOESTO** specifier allows messages for the severity whose routing specification it appears in to be routed to the same destination (and in the same output form) as those for the other, specified, severity level (or component name). For example, the following specification means that **WARNING** messges should show up in three places: twice to **stderr**, and then once to the file **/tmp/foo**:

```
WARNING:STDERR:;GOESTO:FATAL
FATAL:STDERR:;FILE:/tmp/foo
```

Note that a **GOESTO** specification should be the last element in a multidestination route specification.

Routing Serviceability Messages by Environment Variable

Serviceability message routing can also be specified by the values of certain environment variables. If environment variables are used, the routings they specify will override any conflicting routes specified by the routing file.

The routes are specified on the basis of severity level by putting the desired routing instructions in their corresponding environment variables:

- **SVC_FATAL**
- **SVC_ERROR**
- **SVC_WARNING**
- **SVC_NOTICE**
- **SVC_NOTICE_VERBOSE**

Each variable should contain a single string in the format

*out_form***:***dest*[**;***out_form***:***dest* . . .]

where *out_form* and *dest* have the same meanings and form as in the preceding syntax line. Multiple routings can be specified with semicolon-separated additional substrings specifying the additional routes, as shown.

Setting Serviceability Debug Message Levels

Nine serviceability debug message levels (specified respectively by single digits from 1 to 9) are available. The precise meaning of each level varies with the application or DCE component in question, but the general notion is that ascending to a higher level (for example, from **2** to **3**) increases the level of informational detail in the messages.

Setting debug messaging at a certain level means that all levels up to and including the specified level are enabled. For example, if the debug level is set at **4**, then the **1**, **2**, and **3** levels are enabled as well.

The general format for the debug level specifier string is

*component***:***sub_comp***.***level***,***sub_comp***.***level***,**. . .

where

component is the three-character serviceability component code for the program whose debug message levels are being specified.

*sub_comp***.***level*
is a serviceability subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Note that multiple subcomponent/level pairs can be specified in the string.

If there are multiple subcomponents and it is desired to set the debug level to be the same for all of them, then the following form will do this (where ***** specifies all subcomponents):

*component***:*.***level*

Routing Serviceability Debug Messages

Routing for serviceability debug messages can be specified in either of the two following ways:

- By the contents of the **SVC**_*COMP*_**DBG** environment variable (where *COMP* is the code of the component, converted to upper case, whose debug message routing is to be set).
- By the contents of the **/svc/routing** routing file.

The routing is specified by the contents of a specially-formatted string that is either included in the value of the environment variable or the contents of the routing file.

The general format for the debug routing specifier string is

*component***:***sub_comp***.***level***,**. . .**:***out_form***:***dest*[**;***out_form***:***dest* . . .] \
[**GOESTO:**{*sev* | *component*}]

where *component*, *sub_comp.level*, *out_form*, *dest*, and *sev* have the same meanings as defined earlier in this reference page.

For example, consider the following string value:

```
hel:*.4:STDERR:-;TEXTFILE:/tmp/hel_debug_log_%ld
```

This value, when assigned to the **SVC_HEL_DBG** environment variable, would set the debug level and routing for all **hel** subcomponents. A debug level of **4** is specified, and all debug messages of that level or lower will be written both to standard error,

and in text form to the file **/tmp/hel_debug_log_***process_ID*, where *process_ID* is the process ID of the program writing the messages.

Chapter 2

DCE Threads

thr_intro

Purpose

Introduction to DCE Threads

Description

DCE Threads is a set of routines that you can call to create a multithreaded program. Multithreading is used to improve the performance of a program. Routines implemented by DCE Threads that are not specified by Draft 4 of the POSIX 1003.4a standard are indicated by an **_np** suffix on the name. These routines are new primitives.

The threads routines sort alphabetically in the reference pages; however, the tables in this introduction list the routines in the following functional groups:

- Threads routines
- Routines that implicitly initialize threads package
- Attributes object routines
- Mutex routines
- Condition variable routines
- Thread-specific data routines
- Threads cancellation routines
- Threads priority and scheduling routines
- Cleanup routines
- The **atfork()** routine
- Signal handling routines

Threads Routines	
Routine	**Description**
pthread_create()	Creates a thread

pthread_delay_np()	Causes a thread to wait for a period of time
pthread_detach()	Marks a thread for deletion
pthread_equal()	Compares one thread identifier to another thread identifier
pthread_exit()	Terminates the calling thread
pthread_join()	Causes the calling thread to wait for the termination of a specified thread
pthread_once()	Calls an initialization routine to be executed only once
pthread_self()	Obtains the identifier of the current thread
pthread_yield()	Notifies the scheduler that the current thread will release its processor to other threads of the same or higher priority

The following DCE Threads routines will, when called, implicitly perform any necessary initialization of the threads package. Thus any application using DCE Threads should call one of the following routines before calling any other threads routines, in order to ensure that the package is properly initialized.

Routines that Implicitly Perform Threads Initialization	
Routine	**Description**
pthread_attr_create()	Creates a thread attributes object
pthread_create()	Creates a thread
pthread_self()	Obtains the identifier of the current thread
pthread_setprio()	Changes the scheduling priority attribute
pthread_getprio()	Obtains the scheduling priority attribute
pthread_setscheduler()	Changes the scheduling policy attribute
pthread_getscheduler()	Obtains the scheduling policy attribute

pthread_once()	Calls an initialization routine to be executed only once
pthread_keycreate()	Generates a unique thread-specific data key value
pthread_mutexattr_create()	Creates a mutex attributes object
pthread_mutex_init()	Creates a mutex
pthread_condattr_create()	Creates a condition variable attributes object
pthread_cond_init()	Creates a condition variable
pthread_testcancel()	Requests delivery of a pending cancel
pthread_setcancel()	Enables or disables the current thread's general cancelability
pthread_setasynccancel()	Enables or disables the current thread's asynchronous cancelability
pthread_delay_np()	Causes a thread to wait for a period of time

Attributes Object Routines	
Routine	**Description**
pthread_attr_create()	Creates a thread attributes object
pthread_attr_delete()	Deletes a thread attributes object
pthread_attr_getinheritsched()	Obtains the inherit scheduling attribute
pthread_attr_getprio()	Obtains the scheduling priority attribute
pthread_attr_getsched()	Obtains the scheduling policy attribute
pthread_attr_getstacksize()	Obtains the stacksize attribute
pthread_attr_setinheritsched()	Changes the inherit scheduling attribute
pthread_attr_setprio()	Changes the scheduling priority attribute
pthread_attr_setsched()	Changes the scheduling policy attribute
pthread_attr_setstacksize()	Changes the stacksize attribute

pthread_condattr_create()	Creates a condition variable attributes object
pthread_condattr_delete()	Deletes a condition variable attributes object
pthread_mutexattr_create()	Creates a mutex attributes object
pthread_mutexattr_delete()	Deletes a mutex attributes object
pthread_mutexattr_getkind_np()	Obtains the mutex type attribute
pthread_mutexattr_setkind_np()	Changes the mutex type attribute

Mutex Routines	
Routine	**Description**
pthread_lock_global_np()	Locks a global mutex
pthread_mutex_destroy()	Deletes a mutex
pthread_mutex_init()	Creates a mutex
pthread_mutex_lock()	Locks a mutex and waits if the mutex is already locked
pthread_mutex_trylock()	Locks a mutex and returns if the mutex is already locked
pthread_mutex_unlock()	Unlocks a mutex
pthread_unlock_global_np()	Unlocks a global mutex

Condition Variable Routines	
Routine	**Description**
pthread_cond_broadcast()	Wakes all threads waiting on a condition variable
pthread_cond_destroy()	Deletes a condition variable
pthread_cond_init()	Creates a condition variable
pthread_cond_signal()	Wakes one thread waiting on a condition variable

pthread_cond_timedwait()	Causes a thread to wait for a specified period of time for a condition variable to be signaled or broadcast
pthread_cond_wait()	Causes a thread to wait for a condition variable to be signaled or broadcast
pthread_get_expiration_np()	Obtains a value representing a desired expiration time

Thread-Specific Data	
Routine	**Description**
pthread_getspecific()	Obtains the thread-specific data associated with the specified key
pthread_keycreate()	Generates a unique thread-specific data key value
pthread_setspecific()	Sets the thread-specific data associated with the specified key

Threads Cancellation Routines	
Routine	**Description**
pthread_cancel()	Allows a thread to request termination
pthread_setasynccancel()	Enables or disables the current thread's asynchronous cancelability
pthread_setcancel()	Enables or disables the current thread's general cancelability
pthread_signal_to_cancel_np()	Cancels a thread if a signal is received by the process
pthread_testcancel()	Requests delivery of a pending cancel

Threads Priority and Scheduling Routines	
Routine	**Description**
pthread_getprio()	Obtains the current priority of a thread

pthread_getscheduler()	Obtains the current scheduling policy of a thread
pthread_setprio()	Changes the current priority of a thread
pthread_setscheduler()	Changes the current scheduling policy and priority of a thread

Cleanup Routines	
Routine	**Description**
pthread_cleanup_pop()	Removes a cleanup handler from the stack
pthread_cleanup_push()	Establishes a cleanup handler

The atfork() Routine	
Routine	**Description**
atfork()	Arranges for fork cleanup handling

Signal Handling Routines	
Routine	**Description**
sigaction()	Specifies action to take on receipt of signal
sigpending()	Examines pending signals
sigprocmask()	Sets the current signal mask
sigwait()	Causes thread to wait for asynchronous signal

datatypes

Purpose

Data types used by DCE Threads

Description

The DCE Threads data types can be divided into two broad categories: primitive system and application level.

Primitive System Data Types

The first category consists of types that represent structures used by (and internal to) DCE Threads. These types are defined as being primitive system data types.

- **pthread_attr_t**
- **pthread_cond_t**
- **pthread_condattr_t**
- **pthread_key_t**
- **pthread_mutex_t**
- **pthread_mutexattr_t**
- **pthread_once_t**
- **pthread_t**

Although applications must know about these types, passing them in and receiving them from various DCE Threads routines, the structures themselves are opaque: they cannot be directly modified by applications, and they can be manipulated only (and only in some cases) through specific DCE Threads routines. (The **pthread_key_t** type is somewhat different from the others in this list, in that it is essentially a handle to a thread-private block of memory requested by a call to **pthread_keycreate()**.)

Application Level Data Types

The second category of DCE Threads data consists of types used to describe objects that originate in the application:

- **pthread_addr_t**

- **pthread_destructor_t**
- **pthread_initroutine_t**
- **pthread_startroutine_t**
- **sigset_t**

All of the above types, with the exception of the last, are various kinds of memory addresses that must be passed by callers of certain DCE Threads routines. These types are extensions to POSIX. They permit DCE Threads to be used on platforms that are not fully ANSI C compliant. While being extensions to permit the use of compilers that are not ANSI C compatible, they are fully portable data types.

The last data type, **sigset_t**, exhibits properties of both primitive system and application level data types. While objects of this type originate in the application, the data type is opaque. A set of functions is provided to manipulate objects of this type.

For further information, see the following descriptions, listed in sorted order.

Data Type Descriptions

Following are individual descriptions of each of the DCE Threads data types. The descriptions include the routines where the data type is modified, such as, created, changed or deleted/destroyed, but not the routines referencing or using them that do not change them.

- **pthread_addr_t**

  A miscellaneous data type representing an address value that must be passed by the caller of various threads routines. Usually the **pthread_addr_t** value is the address of an area which contains various parameters to be made accessible to an implicitly called routine. For example, when the **pthread_create()** routine is called, one of the parameters passed is a **pthread_addr_t** value that contains an address which will be passed to the *start_routine* which the thread is being created to execute; presumably the routine will extract necessary parameters from the area referenced by this address.

- **pthread_attr_t**

  Threads attribute object, used to specify the attributes of a thread when it is created by a call to **pthread_create()**. The object is created by a call to **pthread_attr_create()**, then modified as desired by calls to

  — **pthread_attr_setinheritsched()**

— **pthread_attr_setprio()**

— **pthread_attr_setsched()**

— **pthread_attr_setstacksize()**

(Note that there are **_get** versions of these four calls, which can be used to retrieve the respective values.)

- **pthread_cond_t**

  Data type representing a threads condition variable. The variable is created by a call to **pthread_cond_init()**, and destroyed by a call to **pthread_cond_destroy()**.

- **pthread_condattr_t**

  Data type representing a threads condition variable attributes object. Created by a call to **pthread_condattr_create()**. The range of possible modifications to a condition variable attributes object is not great: creation (via **pthread_condattr_create()**) and deletion (via **pthread_condattr_delete()**) are all. The object is created with default values.

- **pthread_destructor_t**

  Data type, passed in a call to **pthread_keycreate()**, representing the address of a procedure to be called to destroy a data value associated with a unique thread-specific data key value when the thread terminates.

- **pthread_initroutine_t**

  Data type representing the address of a procedure that performs a one-time initialization for a thread. It is passed in a call to **pthread_once()**. The **pthread_once()** routine, when called, executes the initialization routine. The specified routine is *guaranteed to be executed only once*, even though the **pthread_once()** call occurs in multithreaded code.

- **pthread_key_t**

  Data type representing a thread-specific data key, created by a call to **pthread_keycreate()**. The key is an address of memory. Associating a static block of memory with a specific thread in this way is an alternative to using stack memory for the thread. The key is destroyed by the application-supplied procedure specified by the routine specified using the **pthread_destructor_t** data type in the call to **pthread_keycreate()**.

- **pthread_mutex_t**

Data type representing a mutex object. It is created by a call to **pthread_mutex_init()** and destroyed by a call to **pthread_mutex_destroy()**. Care should be taken not to attempt to destroy a locked object.

- **pthread_mutexattr_t**

  Data type representing an attributes object which defines the characteristics of a mutex. Created by a call to **pthread_mutexattr_create()**; modified by calls to **pthread_mutexattr_setkind_np()** (which allows you to specify fast, recursive, or nonrecursive mutexes); passed to **pthread_mutex_init()** to create the mutex with the specified atttributes. The only other modification allowed is to destroy the mutex attributes object, with **pthread_mutexattr_delete()**.

- **pthread_once_t**

  A data structure that defines the characteristics of the one-time initialization routine executed by calling **pthread_once()**. The structure is opaque to the application, and cannot be modified by it, but it must be explicitly declared by the client code, and initialized by a call to **pthread_once_init()**. The **pthread_once_t** type must not be an array.

- **pthread_startroutine_t**

  Data type representing the address of the application routine or other routine, whatever it is, that a new thread is created to execute as its start routine.

- **pthread_t**

  Data type representing a thread handle, created by a call to **pthread_create()**. The thread handle is used thenceforth to identify the thread to calls such as **pthread_cancel()**, **pthread_detach()**, **pthread_equal()** (to which two handles are passed for comparison).

- **sigset_t**

  Data type representing a set of signals. It is always an integral or structure type. If a structure, it is intended to be a simple structure, such as, a set of arrays as opposed to a set of pointers. It is opaque in that a set of functions called the **sigsetops** primitives is provided to manipulate signal sets. They operate on signal set data objects addressable by the application, not on any objects known to the system.

  The primitives are **sigemptyset()** and **sigfillset()** which initialize the set as either empty or full, **sigaddset()** and **sigdelset()** which add or delete signals from the set, and **sigismember()** which permits the application to check if a object (signal)

of type **sigset_t** is a member of the signal set. Applications must call at least one of the initialization primitives at least once for each object of type **sigset_t** prior to any other use of that object (signal set).

The object, or objects, represented by this data type when used by **sigaction()** is (are) used in conjunction with a **sigaction** structure by the **sigaction** function to describe an action to be taken with (a) specified **sigset_t**-type object(s).

atfork

Purpose

Arranges for fork cleanup handling

Synopsis

```
#include <pthread.h>

void atfork(
        void (*user_state)(),
        void (*pre_fork)(),
        void (*parent_fork)(),
        void (*child_fork)());
```

Parameters

user_state	Pointer to the user state that is passed to each routine.
pre_fork	Routine to be called before performing the fork.
parent_fork	Routine to be called in the parent after the fork.
child_fork	Routine to be called in the child after the fork.

Description

The **atfork()** routine allows you to register three routines to be executed at different times relative to a fork. The different times and/or places are as follows:

- Just prior to the fork in the parent process.
- Just after the fork in the parent process.
- Just after the fork in the created (child) process.

Use these routines to clean up just prior to **fork**(), to set up after **fork**(), and to perform locking relative to **fork**(). You are allowed to provide one parameter to be used in conjunction with all the routines. This parameter must be *user_state*.

Return Values

The **atfork**() routine does not return a value. Instead, an exception is raised if there is insufficient table space to record the handler addresses.

Related Information

Functions: **fork(2)**.

exceptions

Purpose

Exception handling in DCE Threads

Description

DCE Threads provides the following two ways to obtain information about the status of a threads routine:

- The routine returns a status value to the thread.
- The routine raises an exception.

Before you write a multithreaded program, you must choose only one of the preceding two methods of receiving status. These two methods cannot be used together in the same code module.

The POSIX P1003.4a (pthreads) draft standard specifies that errors be reported to the thread by setting the external variable **errno** to an error code and returning a function value of −1. The threads reference pages document this status value-returning interface. However, an alternative to status values is provided by DCE Threads in the exception-returning interface.

Access to exceptions from the C language is defined by the macros in the **exc_handling.h** file. The **exc_handling.h** header file is included automatically when you include **pthread_exc.h.**

To use the exception-returning interface, replace **#include <pthread.h>** with the following include statement:

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

The following example shows the syntax for handling exceptions:

```
TRY
    try_block
```

```
[CATCH (exception_name)
    handler_block]...
[CATCH_ALL
    handler_block]
ENDTRY
```

pthread_attr_create

Purpose

Creates a thread attributes object

Synopsis

```
#include <pthread.h>

int pthread_attr_create(
        pthread_attr_t *attr);
```

Parameters

attr	Thread attributes object created.

Description

The **pthread_attr_create()** routine creates a thread attributes object that is used to specify the attributes of threads when they are created. The attributes object created by this routine is used in calls to **pthread_create()**.

The individual attributes (internal fields) of the attributes object are set to default values. (The default values of each attribute are discussed in the descriptions of the following services.) Use the following routines to change the individual attributes:

- **pthread_attr_setinheritsched()**
- **pthread_attr_setprio()**
- **pthread_attr_setsched()**
- **pthread_attr_setstacksize()**

When an attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new thread. Attributes objects perform in a manner similar to additional parameters. Changing individual attributes does not affect any threads that were previously created using the attributes object.

Return Values

If the function fails, -1 is returned and **errno** may be set to one of the following values:

Return	Error	Description
−1	[**ENOMEM**]	Insufficient memory exists to create the thread attributes object.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_delete(3thr)**, **pthread_attr_setinheritsched(3thr)**, **pthread_attr_setprio(3thr)**, **pthread_attr_setsched(3thr)**, **pthread_attr_setstacksize(3thr)**, **pthread_create(3thr)**.

pthread_attr_delete

Purpose

Deletes a thread attributes object

Synopsis

```
#include <pthread.h>

int pthread_attr_delete(
        pthread_attr_t *attr);
```

Parameters

attr	Thread attributes object deleted.

Description

The **pthread_attr_delete()** routine deletes a thread attributes object and gives permission to reclaim storage for the thread attributes object. Threads that were created using this thread attributes object are not affected by the deletion of the thread attributes object.

The results of calling this routine are unpredictable if the value specified by the *attr* parameter refers to a thread attributes object that does not exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**.

pthread_attr_getinheritsched

Purpose

Obtains the inherit scheduling attribute

Synopsis

```
#include <pthread.h>

int pthread_attr_getinheritsched(
        pthread_attr_t attr);
```

Parameters

attr	Thread attributes object whose inherit scheduling attribute is obtained.

Description

The **pthread_attr_getinheritsched()** routine obtains the value of the inherit scheduling attribute in the specified thread attributes object. The inherit scheduling attribute specifies whether threads created using the attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to **pthread_create()**.

The default value of the inherit scheduling attribute is **PTHREAD_INHERIT_SCHED**.

Return Values

On successful completion, this routine returns the inherit scheduling attribute value.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Inherit scheduling attribute		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_setinheritsched(3thr)**, **pthread_create(3thr)**.

pthread_attr_getprio

Purpose

Obtains the scheduling priority attribute

Synopsis

```
#include <pthread.h>

int pthread_attr_getprio(
        pthread_attr_t attr);
```

Parameters

attr Thread attributes object whose priority attribute is obtained.

Description

The **pthread_attr_getprio()** routine obtains the value of the scheduling priority of threads created using the thread attributes object specified by the *attr* parameter.

Return Values

On successful completion, this routine returns the scheduling priority attribute value.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Scheduling priority attribute		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_setprio(3thr)**, **pthread_create(3thr)**.

pthread_attr_getsched

Purpose

Obtains the value of the scheduling policy attribute

Synopsis

```
#include <pthread.h>

int pthread_attr_getsched(
        pthread_attr_t attr);
```

Parameters

attr Thread attributes object whose scheduling policy attribute is obtained.

Description

The **pthread_attr_getsched()** routine obtains the scheduling policy of threads created using the thread attributes object specified by the *attr* parameter. The default value of the scheduling attribute is **SCHED_OTHER**.

Return Values

On successful completion, this routine returns the value of the scheduling policy attribute.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Scheduling policy attribute		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_setsched(3thr)**, **pthread_create(3thr)**.

pthread_attr_getstacksize

Purpose

Obtains the value of the stacksize attribute

Synopsis

```
#include <pthread.h>

long pthread_attr_getstacksize(
        pthread_attr_t attr);
```

Parameters

attr	Thread attributes object whose stacksize attribute is obtained.

Description

The **pthread_attr_getstacksize()** routine obtains the minimum size (in bytes) of the stack for a thread created using the thread attributes object specified by the *attr* parameter.

Return Values

On successful completion, this routine returns the stacksize attribute value.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Stacksize attribute		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_setstacksize(3thr)**, **pthread_create(3thr)**.

pthread_attr_setinheritsched

Purpose

Changes the inherit scheduling attribute

Synopsis

```
#include <pthread.h>

int pthread_attr_setinheritsched(
        pthread_attr_t attr,
        int inherit);
```

Parameters

attr Thread attributes object to be modified.

inherit New value for the inherit scheduling attribute. Valid values are as follows:

PTHREAD_INHERIT_SCHED
This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling **pthread_create()**.

PTHREAD_DEFAULT_SCHED
The created thread starts execution with the priority and scheduling policy stored in the thread attributes object.

Description

The **pthread_attr_setinheritsched()** routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the specified thread attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the thread attributes object that is passed to **pthread_create()**.

The first thread in an application that is not created by an explicit call to **pthread_create()** has a scheduling policy of **SCHED_OTHER**. (See the **pthread_attr_setprio()** and **pthread_attr_setsched()** routines for more information on valid priority values and valid scheduling policy values, respectively.)

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

Return Values

If the function fails, -1 is returned, and **errno** may be set to one of the following values:

Return	Error	Description
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**EINVAL**]	The value specified by *inherit* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_getinheritsched(3thr)**, **pthread_attr_setprio(3thr)**, **pthread_attr_setsched(3thr)**, **pthread_create(3thr)**.

pthread_attr_setprio

Purpose

Changes the scheduling priority attribute of thread creation

Synopsis

```
#include <pthread.h>

int pthread_attr_setprio(
        pthread_attr_t *attr,
        int priority);
```

Parameters

attr Thread attributes object modified.

priority New value for the priority attribute. The priority attribute depends on scheduling policy. Valid values fall within one of the following ranges:

- **PRI_OTHER_MIN** <= *priority* <= **PRI_OTHER_MAX** (use with the **SCHED_OTHER** policy)
- **PRI_FIFO_MIN** <= *priority* <= **PRI_FIFO_MAX** (use with the **SCHED_FIFO** policy)
- **PRI_RR_MIN** <= *priority* <= **PRI_RR_MAX** (use with the **SCHED_RR** policy)
- **PRI_FG_MIN_NP** <= *priority* <= **PRI_FG_MAX_NP** (use with the **SCHED_FG_NP** policy)
- **PRI_BG_MIN_NP** <= *priority* <= **PRI_BG_MAX_NP** (use with the **SCHED_BG_NP** policy)

The default priority is the midpoint between **PRI_OTHER_MIN** and **PRI_OTHER_MAX**. To specify a minimum or maximum priority, use the appropriate symbol; for example, **PRI_FIFO_MIN** or **PRI_FIFO_MAX**. To specify a value between the minimum and maximum, use an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and

maximum for the Round Robin scheduling policy, specify the following concept using your programming language's syntax:

```
pri_rr_mid = (PRI_RR_MIN + PRI_RR_MAX + 1)/2
```

If your expression results in a value outside the range of minimum to maximum, an error results when you attempt to use it.

Description

The **pthread_attr_setprio()** routine sets the execution priority of threads that are created using the attributes object specified by the *attr* parameter.

By default, a created thread inherits the priority of the thread calling **pthread_create()**. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Before calling this routine and **pthread_create()**, call **pthread_attr_setinheritsched()** and specify the value **PTHREAD_DEFAULT_SCHED** for the *inherit* parameter.

An application specifies priority only to express the urgency of executing the thread relative to other threads. Priority is not used to control mutual exclusion when accessing shared data.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *attr* is invalid.
–1	[**ERANGE**]	One or more parameters supplied have an invalid value.
–1	[**EPERM**]	The caller does not have the appropriate privileges to set the priority of the specified thread.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_getprio(3thr)**, **pthread_attr_setinheritsched(3thr)**, **pthread_create(3thr)**.

pthread_attr_setsched

Purpose

Changes the scheduling policy attribute of thread creation

Synopsis

```
#include <pthread.h>

int pthread_attr_setsched(
        pthread_attr_t *attr,
        int scheduler);
```

Parameters

attr The thread attributes object modified.

scheduler The new value for the scheduling policy attribute. Valid values are as follows:

SCHED_FIFO
First In, First Out—The highest-priority thread runs until it blocks. If there is more than one thread with the same priority, and that priority is the highest among other threads, the first thread to begin running continues until it blocks.

SCHED_RR
Round Robin—The highest-priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. Timeslicing is a process in which threads alternate making use of available processors.

SCHED_OTHER
Default—All threads are timesliced. **SCHED_OTHER** ensures that all threads, regardless of priority, receive some scheduling so that no thread is completely denied execution time. (However, **SCHED_OTHER** threads

can be denied execution time by **SCHED_FIFO** or **SCHED_RR** threads.)

SCHED_FG_NP
Foreground—Same as **SCHED_OTHER**. Threads are timesliced and priorities can be modified dynamically by the scheduler to ensure fairness.

SCHED_BG_NP
Background—Ensures that all threads, regardless of priority, receive some scheduling. However, **SCHED_BG_NP** can be denied execution by **SCHED_FIFO** or **SCHED_RR** threads.

Description

The **pthread_attr_setsched()** routine sets the scheduling policy of a thread that is created by using the attributes object specified by the *attr* parameter. The default value of the scheduling attribute is **SCHED_OTHER**.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *attr* is invalid.
–1	[**EINVAL**]	The value specified by *scheduler* is invalid.
–1	[**EPERM**]	The caller does not have the appropriate privileges to set the scheduling policy attribute in the specified threads attribute object.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_getsched(3thr)**, **pthread_attr_setinheritsched(3thr)**, **pthread_create(3thr)**.

pthread_attr_setstacksize

Purpose

Changes the stacksize attribute of thread creation

Synopsis

```
#include <pthread.h>

int pthread_attr_setstacksize(
        pthread_attr_t *attr,
        long stacksize);
```

Parameters

attr	Thread attributes object modified.
stacksize	New value for the stacksize attribute. The *stacksize* parameter specifies the minimum size (in bytes) of the stack needed for a thread.

Description

The **pthread_attr_setstacksize()** routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* parameter. Use this routine to adjust the size of the writable area of the stack. The default value of the stacksize attribute is machine specific.

A thread's stack is fixed at the time of thread creation. Only the main or initial thread can dynamically extend its stack.

Most compilers do not check for stack overflow. Ensure that your thread stack is large enough for anything that you call from the thread.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**EINVAL**]	The value specified by *stacksize* is invalid.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_attr_getstacksize(3thr)**, **pthread_create(3thr)**.

pthread_cancel

Purpose

Allows a thread to request that it or another thread terminate execution

Synopsis

```
#include <pthread.h>

int pthread_cancel(
        pthread_t thread);
```

Parameters

thread Thread that receives a cancel request.

Description

The **pthread_cancel()** routine sends a cancel to the specified thread. A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

Because of communications delays, the calling thread can only rely on the fact that a cancel eventually becomes pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending cancel is to be delivered because delivery is controlled by the designated thread.

Termination processing when a cancel is delivered to a thread is similar to **pthread_exit()**. Outstanding cleanup routines are executed in the context of the target thread, and a status of −1 is made available to any threads joining with the target thread.

This routine is preferred in implementing Ada's **abort** statement and any other language (or software-defined construct) for requesting thread cancellation.

The results of this routine are unpredictable if the value specified in *thread* refers to a thread that does not currently exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The specified thread is invalid.
−1	[**ERSCH**]	The specified thread does not refer to a currently existing thread.

Related Information

Functions: **pthread_exit(3thr)**, **pthread_join(3thr)**, **pthread_setasynccancel(3thr)**, **pthread_setcancel(3thr)**, **pthread_testcancel(3thr)**.

pthread_cleanup_pop

Purpose

Removes the cleanup handler at the top of the cleanup stack and optionally executes it

Synopsis

```
#include <pthread.h>

void pthread_cleanup_pop(
        int execute);
```

Parameters

execute Integer that specifies whether the cleanup routine that is popped should be executed or just discarded. If the value is nonzero, the cleanup routine is executed.

Description

The **pthread_cleanup_pop()** routine removes the routine specified in **pthread_cleanup_push()** from the top of the calling thread's cleanup stack and executes it if the value specified in *execute* is nonzero.

This routine and **pthread_cleanup_push()** are implemented as macros and must be displayed as statements and in pairs within the same lexical scope. You can think of the **pthread_cleanup_push()** macro as expanding to a string whose first character is a **{** (left brace) and **pthread_cleanup_pop** as expanding to a string containing the corresponding **}** (right brace).

Return Values

This routine must be used as a statement.

Related Information

Functions: **pthread_cleanup_push(3thr)**.

pthread_cleanup_push

Purpose

Establishes a cleanup handler

Synopsis

```
#include <pthread.h>

void pthread_cleanup_push(
        void routine,
        pthread_addr_t arg);
```

Parameters

routine	Routine executed as the cleanup handler.
arg	Parameter executed with the cleanup routine.

Description

The **pthread_cleanup_push()** routine pushes the specified routine onto the calling thread's cleanup stack. The cleanup routine is popped from the stack and executed with the *arg* parameter when any of the following actions occur:

- The thread calls **pthread_exit()**.
- The thread is canceled.
- The thread calls **pthread_cleanup_pop()** and specifies a nonzero value for the *execute* parameter.

This routine and **pthread_cleanup_pop()** are implemented as macros and must be displayed as statements and in pairs within the same lexical scope. You can think of the **pthread_cleanup_push()** macro as expanding to a string whose first character is a **{** (left brace) and **pthread_cleanup_pop()** as expanding to a string containing the corresponding **}** (right brace).

Return Values

This routine must be used as a statement.

Related Information

Functions: **pthread_cancel(3thr)**, **pthread_cleanup_pop(3thr)**, **pthread_exit(3thr)**, **pthread_testcancel(3thr)**.

pthread_cond_broadcast

Purpose

Wakes all threads that are waiting on a condition variable

Synopsis

```
#include <pthread.h>

int pthread_cond_broadcast(
        pthread_cond_t *cond);
```

Parameters

cond Condition variable broadcast.

Description

The **pthread_cond_broadcast()** routine wakes all threads waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for one or more waiting threads to proceed. If any one waiting thread might be able to proceed, call **pthread_cond_signal()**.

Call this routine when the associated mutex is either locked or unlocked.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *cond* is invalid.

Related Information

Functions: **pthread_cond_destroy(3thr)**, **pthread_cond_init(3thr)**, **pthread_cond_signal(3thr)**, **pthread_cond_timedwait(3thr)**, **pthread_cond_wait(3thr)**.

pthread_cond_destroy

Purpose

Deletes a condition variable

Synopsis

```
#include <pthread.h>

int pthread_cond_destroy(
        pthread_cond_t *cond);
```

Parameters

cond	Condition variable deleted.

Description

The **pthread_cond_destroy()** routine deletes a condition variable. Call this routine when a condition variable is no longer referenced. The effect of calling this routine is to give permission to reclaim storage for the condition variable.

The results of this routine are unpredictable if the condition variable specified in *cond* does not exist.

The results of this routine are also unpredictable if there are threads waiting for the specified condition variable to be signaled or broadcast when it is deleted.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *cond* is invalid.
–1	[**EBUSY**]	A thread is currently executing a **pthread_cond_timedwait()** routine or **pthread_cond_wait()** on the condition variable specified in *cond*.

Related Information

Functions: **pthread_cond_broadcast(3thr)**, **pthread_cond_init(3thr)**, **pthread_cond_signal(3thr)**, **pthread_cond_timedwait(3thr)**, **pthread_cond_wait(3thr)**.

pthread_cond_init

Purpose

Creates a condition variable

Synopsis

```
#include <pthread.h>

int pthread_cond_init(
        pthread_cond_t *cond,
        pthread_condattr_t attr);
```

Parameters

cond	Condition variable that is created.
attr	Condition variable attributes object that defines the characteristics of the condition variable created. If you specify **pthread_condattr_default**, default attributes are used.

Description

The **pthread_cond_init()** routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state. The state is defined by a Boolean expression called a predicate.

A condition variable is signaled or broadcast to indicate that a predicate might have become true. The broadcast operation indicates that all waiting threads need to resume and reevaluate the predicate. The signal operation is used when any one waiting thread can continue.

If a thread that holds a mutex determines that the shared data is not in the correct state for it to proceed (the associated predicate is not true), it waits on a condition variable associated with the desired state. Waiting on the condition variable automatically

releases the mutex so that other threads can modify or examine the shared data. When a thread modifies the state of the shared data so that a predicate might be true, it signals or broadcasts on the appropriate condition variable so that threads waiting for that predicate can continue.

It is important that all threads waiting on a particular condition variable at any time hold the *same* mutex. If they do not, the behavior of the wait operation is unpredictable (an implementation can use the mutex to control internal access to the condition variable object). However, it is legal for a client to store condition variables and mutexes and later reuse them in different combinations. The client must ensure that no threads use the condition variable with the old mutex. At any time, an arbitrary number of condition variables can be associated with a single mutex, each representing a different predicate of the shared data protected by that mutex.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EAGAIN**]	The system lacks the necessary resources to initialize another condition variable.
−1	[**EINVAL**]	Invalid attributes object.
−1	[**ENOMEM**]	Insufficient memory exists to initialize the condition variable.

Related Information

Functions: **pthread_cond_broadcast(3thr)**, **pthread_cond_destroy(3thr)**, **pthread_cond_signal(3thr)**, **pthread_cond_timedwait(3thr)**, **pthread_cond_wait(3thr)**.

pthread_cond_signal

Purpose

Wakes one thread that is waiting on a condition variable

Synopsis

```
#include <pthread.h>

int pthread_cond_signal(
        pthread_cond_t *cond);
```

Parameters

cond	Condition variable signaled.

Description

The **pthread_cond_signal()** routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it is possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true, but only one thread needs to proceed.

The scheduling policy determines which thread is awakened. For policies **SCHED_FIFO** and **SCHED_RR** a blocked thread is chosen in priority order.

Call this routine when the associated mutex is either locked or unlocked.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *cond* is invalid.

Related Information

Functions: **pthread_cond_broadcast(3thr)**, **pthread_cond_destroy(3thr)**, **pthread_cond_init(3thr)**, **pthread_cond_timedwait(3thr)**, **pthread_cond_wait(3thr)**.

pthread_cond_timedwait

Purpose

Causes a thread to wait for a condition variable to be signaled or broadcast

Synopsis

```
#include <pthread.h>

int pthread_cond_timedwait(
        pthread_cond_t *cond,
        pthread_mutex_t *mutex,
        struct timespec *abstime);
```

Parameters

cond	Condition variable waited on.
mutex	Mutex associated with the condition variable specified in *cond*.
abstime	Absolute time at which the wait expires, if the condition has not been signaled or broadcast. (See the **pthread_get_expiration_np()** routine, which you can use to obtain a value for this parameter.)

Description

The **pthread_cond_timedwait()** routine causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcast.
- The current system clock time is greater than or equal to the time specified by the *abstime* parameter.

This routine is identical to **pthread_cond_wait()** except that this routine can return before a condition variable is signaled or broadcast—specifically, when a specified time expires.

If the current time equals or exceeds the expiration time, this routine returns immediately, without causing the current thread to wait.

Call this routine after you lock the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The value specified by *cond*, *mutex*, or *abstime* is invalid.
–1	[**EAGAIN**]	The time specified by *abstime* expired.
–1	[**EDEADLK**]	A deadlock condition is detected.

Related Information

Functions: **pthread_cond_broadcast(3thr)**, **pthread_cond_destroy(3thr)**, **pthread_cond_init(3thr)**, **pthread_cond_signal(3thr)**, **pthread_cond_wait(3thr)**, **pthread_get_expiration_np(3thr)**.

pthread_cond_wait

Purpose

Causes a thread to wait for a condition variable to be signaled or broadcast

Synopsis

```
#include <pthread.h>

int pthread_cond_wait(
        pthread_cond_t *cond,
        pthread_mutex_t *mutex);
```

Parameters

cond	Condition variable waited on.
mutex	Mutex associated with the condition variable specified in *cond*.

Description

The **pthread_cond_wait()** routine causes a thread to wait for a condition variable to be signaled or broadcast. Each condition corresponds to one or more predicates based on shared data. The calling thread waits for the data to reach a particular state (for the predicate to become true).

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine automatically releases the mutex and causes the calling thread to wait on the condition. If the wait is satisfied as a result of some thread calling **pthread_cond_signal()** or **pthread_cond_broadcast()**, the mutex is reacquired and the routine returns.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either **pthread_cond_signal()** or **pthread_cond_broadcast()** for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcast. When a spurious wakeup occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, enclose this routine in a loop that checks the predicate.)

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *cond* or *mutex* is invalid.
−1	[**EDEADLK**]	A deadlock condition is detected.

Related Information

Functions: **pthread_cond_broadcast(3thr)**, **pthread_cond_destroy(3thr)**, **pthread_cond_init(3thr)**, **pthread_cond_signal(3thr)**, **pthread_cond_timedwait(3thr)**.

pthread_condattr_create

Purpose

Creates a condition variable attributes object

Synopsis

```
#include <pthread.h>

int pthread_condattr_create(
        pthread_condattr_t *attr);
```

Parameters

attr	Condition variable attributes object that is created.

Description

The **pthread_condattr_create()** routine creates a condition variable attributes object that is used to specify the attributes of condition variables when they are created. The condition variable attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a condition variable attributes object is used to create a condition variable, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional parameters to object creation. Changing individual attributes does not affect objects that were previously created using the attributes object.

Return Values

The created condition variable attributes object is returned to the *attr* parameter.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**ENOMEM**]	Insufficient memory exists to create the condition variable attributes object.

Related Information

Functions: **pthread_cond_init(3thr)**, **pthread_condattr_delete(3thr)**.

pthread_condattr_delete

Purpose

Deletes a condition variable attributes object

Synopsis

```
#include <pthread.h>

int pthread_condattr_delete(
        pthread_condattr_t *attr);
```

Parameters

attr	Condition variable attributes object deleted.

Description

The **pthread_condattr_delete()** routine deletes a condition variable attributes object. Call this routine when a condition variable attributes object created by **pthread_condattr_create()** is no longer referenced.

This routine gives permission to reclaim storage for the condition variable attributes object. Condition variables that are created using this attributes object are not affected by the deletion of the condition variable attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* parameter refers to an attributes object that does not exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_condattr_create(3thr)**.

pthread_create

Purpose

Creates a thread object and thread

Synopsis

```
#include <pthread.h>

int pthread_create(
        pthread_t *thread,
        pthread_attr_t attr,
        pthread_startroutine_t start_routine,
        pthread_addr_t arg);
```

Parameters

thread	Handle to the thread object created.
attr	Thread attributes object that defines the characteristics of the thread being created. If you specify **pthread_attr_default**, default attributes are used.
start_routine	Function executed as the new thread's start routine.
arg	Address value copied and passed to the thread's start routine.

Description

The **pthread_create()** routine creates a thread object and a thread. A *thread* is a single, sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. A thread object defines and controls the executing thread.

Creating a Thread

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.

- The associated executable thread is created with attributes specified by the *attr* parameter (or with default attributes if **pthread_attr_default** is specified).
- The *thread* parameter receives the new thread.
- The *start_routine* function is called. This may occur before this routine returns successfully.

Thread Execution

The thread is created in the ready state and therefore might immediately begin executing the function specified by the *start_routine* parameter. The newly created thread begins running before **pthread_create()** completes if the new thread follows the **SCHED_RR** or **SCHED_FIFO** scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which with sufficient processors might also be before **pthread_create()** returns.

The *start_routine* parameter is passed a copy of the *arg* parameter. The value of the *arg* parameter is unspecified.

The thread object exists until the **pthread_detach()** routine is called or the thread terminates, whichever occurs last.

The synchronization between the caller of **pthread_create()** and the newly created thread is through the use of the **pthread_join()** routine (or any other mutexes or condition variables they agree to use).

Terminating a Thread

A thread terminates when one of the following events occurs:

- The thread returns from its start routine.
- The thread exits (within a routine) as the result of calling the **pthread_exit()** routine.
- The thread is canceled.

When a Thread Terminates

The following actions are performed when a thread terminates:

- If the thread terminates by returning from its start routine or calling **pthread_exit()**, the return value is copied into the thread object. If the start routine returns normally and the start routine is a procedure that does not return a value, then the result obtained by **pthread_join()** is unpredictable. If the thread has been cancelled, a return value of −1 is copied into the thread object. The

return value can be retrieved by other threads by calling the **pthread_join()** routine.

- A destructor for each thread-specific data point is removed from the list of destructors for this thread and then is called. This step destroys all the thread-specific data associated with the current thread.
- Each cleanup handler that has been declared by **pthread_cleanup_push()** and not yet removed by **pthread_cleanup_pop()** is called. The most recently pushed handler is called first.
- A flag is set in the thread object indicating that the thread has terminated. This flag must be set in order for callers of **pthread_join()** to return from the call.
- A broadcast is made so that all threads currently waiting in a call to **pthread_join()** can return from the call.
- The thread object is marked to indicate that it is no longer needed by the thread itself. A check is made to determine if the thread object is no longer needed by other threads; that is, if **pthread_detach()** has been called. If that routine is called, then the thread object is deallocated.

Return Values

Upon successful completion, this routine stores the identifier of the created thread at *thread* and returns 0. Otherwise, a value of -1 is returned and no thread is created, the contents of *thread* are undefined, and **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EAGAIN**]	The system lacks the necessary resources to create another thread.
–1	[**ENOMEM**]	Insufficient memory exists to create the thread object. This is not a temporary condition.

Related Information

Functions: **pthread_attr_create(3thr)**, **pthread_cancel(3thr)**, **pthread_detach(3thr)**, **pthread_exit(3thr)**, **pthread_join(3thr)**.

pthread_delay_np

Purpose

Causes a thread to wait for a specified period

Synopsis

```
#include <pthread.h>

int pthread_delay_np(
        struct timespec *interval);
```

Parameters

interval Number of seconds and nanoseconds that the calling thread waits before continuing execution. The value specified must be greater than or equal to 0 (zero).

Description

The **pthread_delay_np()** routine causes a thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the *interval* parameter.

Specifying an interval of 0 (zero) seconds and 0 (zero) nanoseconds is allowed and can result in the thread giving up the processor or delivering a pending cancel.

The **struct timespec** structure contains two fields, as follows:

- The **tv_sec** field is an integer number of seconds.
- The **tv_nsec** field is an integer number of nanoseconds.

This routine is a new primitive.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *interval* is invalid.

Related Information

Functions: **pthread_yield(3thr)**.

pthread_detach

Purpose

Marks a thread object for deletion

Synopsis

```
#include <pthread.h>

int pthread_detach(
        pthread_t *thread);
```

Parameters

thread Thread object marked for deletion.

Description

The **pthread_detach()** routine indicates that storage for the specified thread is reclaimed when the thread terminates. This includes storage for the *thread* parameter's return value. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

Call this routine when a thread object is no longer referenced. Additionally, call this routine for every thread that is created to ensure that storage for thread objects does not accumulate.

You cannot join with a thread after the thread has been detached.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.
−1	[**ESRCH**]	The value specified by *thread* does not refer to an existing thread.

Related Information

Functions: **pthread_cancel(3thr)**, **pthread_create(3thr)**, **pthread_exit(3thr)**, **pthread_join(3thr)**.

pthread_equal

Purpose

Compares one thread identifier to another thread identifier.

Synopsis

```
#include <pthread.h>

boolean32 pthread_equal(
        pthread_t *thread1,
        pthread_t *thread2);
```

Parameters

thread1	The first thread identifier to be compared.
thread2	The second thread identifier to be compared.

Description

This routine compares one thread identifier to another thread identifier. (This routine does not check whether the objects that correspond to the identifiers currently exist.) If the identifiers have values indicating that they designate the same object, 1 (true) is returned. If the values do not designate the same object, 0 (false) is returned.

This routine is implemented as a C macro.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Values of thread1 and thread2 do not designate the same object.
1		Values of thread1 and thread2 designate the same object.

Related Information

Functions: **pthread_create(3thr)**

pthread_exit

Purpose

Terminates the calling thread

Synopsis

```
#include <pthread.h>

void pthread_exit(
        pthread_addr_t status);
```

Parameters

status Address value copied and returned to the caller of **pthread_join()**.

Description

The **pthread_exit()** routine terminates the calling thread and makes a status value available to any thread that calls **pthread_join()** and specifies the terminating thread.

An implicit call to **pthread_exit()** is issued when a thread returns from the start routine that was used to create it. The function's return value serves as the thread's exit status. If the return value is −1, an error exit is forced for the thread instead of a normal exit. The process exits when the last running thread calls **pthread_exit()**, with an undefined exit status.

Restrictions

The **pthread_exit()** routine does not work in the main (initial) thread because DCE Threads relies on information at the base of thread stacks; this information does not exist in the main thread.

Return Values

No value is returned.

Related Information

Functions: **pthread_create(3thr)**, **pthread_detach(3thr)**, **pthread_join(3thr)**.

pthread_get_expiration_np

Purpose

Obtains a value representing a desired expiration time

Synopsis

```
#include <pthread.h>

int pthread_get_expiration_np(
        struct timespec *delta,
        struct timespec *abstime);
```

Parameters

delta	Number of seconds and nanoseconds to add to the current system time. The result is the time when a timed wait expires.
abstime	Value representing the expiration time.

Description

The **pthread_get_expiration_np()** routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time is used as the expiration time in a call to **pthread_cond_timedwait()**. This routine is a new primitive.

The **struct timespec** structure contains two fields, as follows:

- The **tv_sec** field is an integer number of seconds.
- The **tv_nsec** field is an integer number of nanoseconds.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *delta* is invalid.

Related Information

Functions: **pthread_cond_timedwait(3thr)**.

pthread_getprio

Purpose

Obtains the current priority of a thread

Synopsis

```
#include <pthread.h>

int pthread_getprio(
        pthread_t thread);
```

Parameters

thread Thread whose priority is obtained.

Description

The **pthread_getprio()** routine obtains the current priority of a thread. The current priority is different from the initial priority of the thread if the **pthread_setprio()** routine is called.

The exact effect of different priority values depends upon the scheduling policy assigned to the thread.

Return Values

The current priority value of the thread specified in *thread* is returned. (See the **pthread_setprio()** reference page for valid values.)

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Priority value		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.
−1	[**ESRCH**]	The value specified by *thread* does not refer to an existing thread.

Related Information

Functions: **pthread_attr_setprio(3thr)**, **pthread_setprio(3thr)**, **pthread_setscheduler(3thr)**.

pthread_getscheduler

Purpose

Obtains the current scheduling policy of a thread

Synopsis

```
#include <pthread.h>

int pthread_getscheduler(
        pthread_t thread);
```

Parameters

thread Thread whose scheduling policy is obtained.

Description

The **pthread_getscheduler()** routine obtains the current scheduling policy of a thread. The current scheduling policy of a thread is different from the initial scheduling policy if the **pthread_setscheduler()** routine is called.

Return Values

The current scheduling policy value of the thread specified in *thread* is returned. (See the **pthread_setscheduler()** reference page for valid values.)

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Current scheduling policy		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.
−1	[**ESRCH**]	The value specified by *thread* does not refer to an existing thread.

Related Information

Functions: **pthread_attr_setscheduler(3thr)**, **pthread_setscheduler(3thr)**.

pthread_getspecific

Purpose

Obtains the thread-specific data associated with the specified key

Synopsis

```
#include <pthread.h>

int pthread_getspecific(
        pthread_key_t key,
        pthread_addr_t *value);
```

Parameters

key Context key value that identifies the data value obtained. This key value must be obtained from **pthread_keycreate()**.

value Address of the current thread-specific data value associated with the specified key.

Description

The **pthread_getspecific()** routine obtains the thread-specific data associated with the specified key for the current thread.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
–1	[**EINVAL**]	The key value is invalid.

Related Information

Functions: **pthread_keycreate(3thr)**, **pthread_setspecific(3thr)**.

pthread_join

Purpose

Causes the calling thread to wait for the termination of a specified thread

Synopsis

```
#include <pthread.h>

int pthread_join(
        pthread_t thread,
        pthread_addr_t *status);
```

Parameters

thread	Thread whose termination is awaited by the caller of this routine.
status	Status value of the terminating thread when that thread calls **pthread_exit()**.

Description

The **pthread_join()** routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

Any number of threads can call this routine. All threads are awakened when the specified thread terminates.

If the current thread calls this routine to join with itself, an error is returned.

The results of this routine are unpredictable if the value for *thread* refers to a thread object that no longer exists.

Return Values

If the thread terminates normally, the exit status is the value that is is optionally returned from the thread's start routine.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.
−1	[**ESRCH**]	The value specified by *thread* does not refer to a currently existing thread.
−1	[**EDEADLK**]	A deadlock is detected.

Related Information

Functions: **pthread_create(3thr)**, **pthread_detach(3thr)**, **pthread_exit(3thr)**.

pthread_keycreate

Purpose

Generates a unique thread-specific data key value

Synopsis

```
#include <pthread.h>

int pthread_keycreate(
        pthread_key_t *key,
        void (*destructor) (void *value));
```

Parameters

key	Value of the new thread-specific data key.
destructor	Procedure to be called to destroy a data value associated with the created key when the thread terminates.

Description

The **pthread_keycreate()** routine generates a unique thread-specific data key value. This key value identifies a thread-specific data value, which is an address of memory generated by the client containing arbitrary data of any size.

Thread-specific data allows client software to associate information with the current thread.

For example, thread-specific data can be used by a language runtime library that needs to associate a language-specific thread-private data structure with an individual thread. The thread-specific data routines also provide a portable means of implementing the class of storage called thread-private static, which is needed to support parallel decomposition in the FORTRAN language.

This routine generates and returns a new key value. Each call to this routine within a process returns a key value that is unique within an application invocation. Calls to **pthread_keycreate()** must occur in initialization code guaranteed to execute only

once in each process. The **pthread_once()** routine provides a way of specifying such code.

When multiple facilities share access to thread-specific data, the facilities must agree on the key value that is associated with the context. The key value must be created only once and needs to be stored in a location known to each facility. (It may be desirable to encapsulate the creation of a key, and the setting and getting of context values for that key, within a special facility created for that purpose.)

When a thread terminates, thread-specific data is automatically destroyed. For each thread-specific data currently associated with the thread, the *destructor* routine associated with the key value of that context is called. The order in which per-thread context destructors are called at thread termination is undefined.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *key* is invalid.
−1	[**EAGAIN**]	An attempt was made to allocate a key when the key namespace is exhausted. This is not a temporary condition.
−1	[**ENOMEM**]	Insufficient memory exists to create the key.

Related Information

Functions: **pthread_getspecific(3thr)**, **pthread_setspecific(3thr)**.

pthread_lock_global_np

Purpose

Locks the global mutex

Synopsis

```
#include <pthread.h>

void pthread_lock_global_np();
```

Description

The **pthread_lock_global_np()** routine locks the global mutex. If the global mutex is currently held by another thread when a thread calls this routine, the thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. (The locking thread must call

pthread_unlock_global_np() as many times as it called this routine to allow another thread to lock the global mutex.)

This routine is a new primitive.

Return Values

No value is returned.

Related Information

Functions: **pthread_mutex_lock(3thr)**, **pthread_mutex_unlock(3thr)**, **pthread_mutexattr_setkind_np(3thr)**, **pthread_unlock_global_np(3thr)**.

pthread_mutex_destroy

Purpose

Deletes a mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_destroy(
        pthread_mutex_t *mutex);
```

Parameters

mutex Mutex to be deleted.

Description

The **pthread_mutex_destroy()** routine deletes a mutex and must be called when a mutex object is no longer referenced. The effect of calling this routine is to reclaim storage for the mutex object.

It is illegal to delete a mutex that has a current owner (in other words, is locked).

The results of this routine are unpredictable if the mutex object specified in the *mutex* parameter does not currently exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EBUSY**]	An attempt was made to destroy a mutex that is locked.
−1	[**EINVAL**]	The value specified by *mutex* is invalid.

Related Information

Functions: **pthread_mutex_init(3thr)**, **pthread_mutex_lock(3thr)**, **pthread_mutex_trylock(3thr)**, **pthread_mutex_unlock(3thr)**.

pthread_mutex_init

Purpose

Creates a mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_init(
        pthread_mutex_t *mutex,
        pthread_mutexattr_t attr);
```

Parameters

mutex	Mutex that is created.
attr	Attributes object that defines the characteristics of the created mutex. If you specify **pthread_mutexattr_default**, default attributes are used.

Description

The **pthread_mutex_init()** routine creates a mutex and initializes it to the unlocked state. If the thread that called this routine terminates, the created mutex is not automatically deallocated, because it is considered shared among multiple threads.

Return Values

If an error condition occurs, this routine returns –1, the mutex is not initialized, the contents of *mutex* are undefined, and **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EAGAIN**]	The system lacks the necessary resources to initialize another mutex.
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**ENOMEM**]	Insufficient memory exists to initialize the mutex.

Related Information

Functions: **pthread_mutex_lock(3thr)**, **pthread_mutex_trylock(3thr)**, **pthread_mutex_unlock(3thr)**, **pthread_mutexattr_create(3thr)**, **pthread_mutexattr_getkind_np(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutex_lock

Purpose

Locks an unlocked mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_lock(
        pthread_mutex_t *mutex);
```

Parameters

mutex	Mutex that is locked.

Description

The **pthread_mutex_lock()** routine locks a mutex. If the mutex is locked when a thread calls this routine, the thread waits for the mutex to become available.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

If you specified a fast mutex in a call to **pthread_mutexattr_setkind_np()**, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. If you specified a recursive mutex in a call to **pthread_mutexattr_setkind_np()**, the current owner of a mutex can relock the same mutex without blocking. If you specify a nonrecursive mutex in a call to **pthread_mutexattr_setkind_np()**, an error is returned and the thread does not block if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time.

The preemption of a lower-priority thread that locks a mutex possibly results in the indefinite blocking of higher-priority threads waiting for the same mutex. The execution of the waiting higher-priority threads is blocked for as long as there is a

sufficient number of runnable threads of any priority between the lower-priority and higher-priority values. Priority inversion occurs when any resource is shared between threads with different priorities.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *mutex* is invalid.
−1	[**EDEADLK**]	A deadlock condition is detected.

Related Information

Functions: **pthread_mutex_destroy(3thr)**, **pthread_mutex_init(3thr)**, **pthread_mutex_trylock(3thr)**, **pthread_mutex_unlock(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutex_trylock

Purpose

Locks a mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_trylock(
        pthread_mutex_t *mutex);
```

Parameters

mutex	Mutex that is locked.

Description

The **pthread_mutex_trylock()** routine locks a mutex. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

When a thread calls this routine, an attempt is made to lock the mutex immediately. If the mutex is successfully locked, 1 is returned and the current thread is then the mutex's current owner.

If the mutex is locked by another thread when this routine is called, 0 (zero) is returned and the thread does not wait to acquire the lock. If a fast mutex is owned by the current thread, 0 is returned. If a recursive mutex is owned by the current thread, 1 is returned and the mutex is relocked. (To unlock a recursive mutex, each call to **pthread_mutex_trylock()** must be matched by a call to the **pthread_mutex_unlock()** routine.)

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
1		Successful completion.
0		The mutex is locked; therefore, it was not acquired.
–1	[**EINVAL**]	The value specified by *mutex* is invalid.

Related Information

Functions: **pthread_mutex_destroy(3thr)**, **pthread_mutex_init(3thr)**, **pthread_mutex_lock(3thr)**, **pthread_mutex_unlock(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutex_unlock

Purpose

Unlocks a mutex

Synopsis

```
#include <pthread.h>

int pthread_mutex_unlock(
        pthread_mutex_t *mutex);
```

Parameters

mutex	Mutex that is unlocked.

Description

The **pthread_mutex_unlock()** routine unlocks a mutex. If no threads are waiting for the mutex, the mutex unlocks with no current owner. If one or more threads are waiting to lock the specified mutex, this routine causes one thread to return from its call to **pthread_mutex_lock()**. The scheduling policy is used to determine which thread acquires the mutex. For the **SCHED_FIFO** and **SCHED_RR** policies, a blocked thread is chosen in priority order.

The results of calling this routine are unpredictable if the mutex specified in *mutex* is unlocked. The results of calling this routine are also unpredictable if the mutex specified in *mutex* is currently owned by a thread other than the calling thread.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *mutex* is invalid.

Related Information

Functions: **pthread_mutex_destroy(3thr)**, **pthread_mutex_init(3thr)**, **pthread_mutex_lock(3thr)**, **pthread_mutex_trylock(3thr)**, **pthread_unlock_global_np(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutexattr_create

Purpose

Creates a mutex attributes object

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_create(
        pthread_mutexattr_t *attr);
```

Parameters

attr	Mutex attributes object created.

Description

The **pthread_mutexattr_create()** routine creates a mutex attributes object used to specify the attributes of mutexes when they are created. The mutex attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a mutex attributes object is used to create a mutex, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional parameters to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

Return Values

The created mutex attributes object is returned to the *attr* parameter.

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**ENOMEM**]	Insufficient memory exists to create the mutex attributes object.

Related Information

Functions: **pthread_create(3thr)**, **pthread_mutex_init(3thr)**, **pthread_mutexattr_delete(3thr)**, **pthread_mutexattr_getkind_np(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutexattr_delete

Purpose

Deletes a mutex attributes object

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_delete(
        pthread_mutexattr_t *attr);
```

Parameters

attr	Mutex attributes object deleted.

Description

The **pthread_mutexattr_delete()** routine deletes a mutex attributes object. Call this routine when a mutex attributes object is no longer referenced by the **pthread_mutexattr_create()** routine.

This routine gives permission to reclaim storage for the mutex attributes object. Mutexes that were created using this attributes object are not affected by the deletion of the mutex attributes object.

The results of calling this routine are unpredictable if the attributes object specified in the *attr* parameter does not exist.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_mutexattr_create(3thr)**.

pthread_mutexattr_getkind_np

Purpose

Obtains the mutex type attribute used when a mutex is created

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_getkind_np(
        pthread_mutexattr_t attr);
```

Parameters

attr Mutex attributes object whose mutex type is obtained.

Description

The **pthread_mutexattr_getkind_np()** routine obtains the mutex type attribute that is used when a mutex is created. See the **pthread_mutexattr_setkind_np()** reference page for information about mutex type attributes.

This routine is a new primitive.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Mutex type attribute		Successful completion.
–1	[**EINVAL**]	The value specified by *attr* is invalid.

Related Information

Functions: **pthread_mutex_init(3thr)**, **pthread_mutexattr_create(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_mutexattr_setkind_np

Purpose

Specifies the mutex type attribute

Synopsis

```
#include <pthread.h>

int pthread_mutexattr_setkind_np(
        pthread_mutexattr_t *attr,
        int kind);
```

Parameters

attr	Mutex attributes object modified.
kind	New value for the mutex type attribute. The *kind* parameter specifies the type of mutex that is created. Valid values are **MUTEX_FAST_NP** (default), **MUTEX_RECURSIVE_NP**, and **MUTEX_NONRECURSIVE_NP**.

Description

The **pthread_mutexattr_setkind_np()** routine sets the mutex type attribute that is used when a mutex is created.

A fast mutex is locked and unlocked in the fastest manner possible. A fast mutex can only be locked (obtained) once. All subsequent calls to **pthread_mutex_lock()** cause the calling thread to block until the mutex is freed by the thread that owns it. If the thread that owns the mutex attempts to lock it again, the thread waits for itself to release the mutex (causing a deadlock).

A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. In other words, a single thread can make consecutive calls to **pthread_mutex_lock()** without blocking. The thread must

then call **pthread_mutex_unlock()** the same number of times as it called **pthread_mutex_lock()** before another thread can lock the mutex.

A nonrecursive mutex is locked only once by a thread, like a fast mutex. If the thread tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, nonrecursive mutexes are more informative than fast mutexes because fast mutexes block in such a case, leaving it up to you to determine why the thread no longer executes. Also, if someone other than the owner tries to unlock a nonrecursive mutex, an error is returned.

Never use a recursive mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait()** or **pthread_cond_timedwait()** might not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

This routine is a new primitive.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *attr* is invalid.
−1	[**EPERM**]	The caller does not have the appropriate privileges.
−1	[**ERANGE**]	One or more parameters supplied have an invalid value.

Related Information

Functions: **pthread_mutex_init(3thr)**, **pthread_mutexattr_create(3thr)**, **pthread_mutexattr_getkind_np(3thr)**.

pthread_once

Purpose

Calls an initialization routine executed by one thread, a single time

Synopsis

```
#include <pthread.h>

int pthread_once(
        pthread_once_t *once_block,
        pthread_initroutine_t init_routine);
```

Parameters

once_block Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique **pthread_once_t** data structure.

init_routine Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once_block* are passed to **pthread_once()**.

Description

The **pthread_once()** routine calls an initialization routine executed by one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads or multiple times in the same thread.

For example, a mutex or a thread-specfic data key must be created exactly once. Calling **pthread_once()** prevents the code that creates a mutex or thread-specific data from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code are delayed until the first thread is finished.

This routine initializes the control record if it has not been initialized and then determines if the client one-time initialization routine has executed once. If it has not executed, this routine calls the initialization routine specified in *init_routine*. If the client one-time initialization code has executed once, this routine returns.

The **pthread_once_t** data structure is a record that allows client initialization operations to guarantee mutual exclusion of access to the initialization routine, and that each initialization routine is executed exactly once.

The client code must declare a variable of type **pthread_once_t** to use the client initialization operations. This variable must be initialized using the **pthread_once_init** macro, as follows:

```
static pthread_once_t myOnceBlock = pthread_once_init;
```

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
−1	[**EINVAL**]	The value specified by a parameter is invalid.
0		Successful completion.

pthread_self

Purpose

Obtains the identifier of the current thread

Synopsis

```
#include <pthread.h>

pthread_t pthread_self( );
```

Description

The **pthread_self()** routine allows a thread to obtain its own identifier. For example, this identifier allows a thread to set its own priority.

This value becomes meaningless when the thread object is deleted; that is, when the thread terminates its execution and **pthread_detach()** is called.

Return Values

Returns the identifier of the calling thread to **pthread_t**.

Related Information

Functions: **pthread_create(3thr)**, **pthread_setprio(3thr)**, **pthread_setscheduler(3thr)**.

pthread_setasynccancel

Purpose

Enables or disables the current thread's asynchronous cancelability

Synopsis

```
#include <pthread.h>

int pthread_setasynccancel(
        int state);
```

Parameters

state State of asynchronous cancelability set for the calling thread. On return, receives the prior state of asynchronous cancelability. Valid values are as follows:

CANCEL_ON
Asynchronous cancelability is enabled.

CANCEL_OFF
Asynchronous cancelability is disabled.

Description

The **pthread_setasynccancel()** routine enables or disables the current thread's asynchronous cancelability and returns the previous asynchronous cancelability state.

When general cancelability is set to **CANCEL_OFF**, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled. When general cancelability is set to **CANCEL_ON**, cancelability depends on the state of the thread's asynchronous cancelability.

When general cancelability is set to **CANCEL_ON** and asynchronous cancelability is set to **CANCEL_OFF**, the thread can only receive a cancel at specific cancellation points (for example, condition waits, thread joins, and calls to the **pthread_testcancel()** routine). If both general cancelability and asynchronous

cancelability are set to **CANCEL_ON**, the thread can be canceled at any point in its execution.

When a thread is created, the default asynchronous cancelability state is **CANCEL_OFF**.

If you call this routine to enable asynchronous cancels, call it in a region of code where asynchronous delivery of cancels is disabled by a previous call to this routine. Do not call threads routines in regions of code where asynchronous delivery of cancels is enabled. The previous state of asynchronous delivery can be restored later by another call to this routine.

Return Values

On successful completion, the previous state of asynchronous cancelability is returned. If the function fails, -1 is returned. Following are the possible return values and the possible corresponding values (if any) for **errno**:

Return	Error	Description
CANCEL_ON		Asynchronous cancelability was on.
CANCEL_OFF		Asynchronous cancelability was off.
–1	[**EINVAL**]	The specified state is not **CANCEL_ON** or **CANCEL_OFF**.

Related Information

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

pthread_setcancel

Purpose

Enables or disables the current thread's general cancelability

Synopsis

```
#include <pthread.h>

int pthread_setcancel(
        int state);
```

Parameters

state State of general cancelability set for the calling thread. On return, receives the prior state of general cancelability. Valid values are as follows:

CANCEL_ON
General cancelability is enabled.

CANCEL_OFF
General cancelability is disabled.

Description

The **pthread_setcancel()** routine enables or disables the current thread's general cancelability and returns the previous general cancelability state.

When general cancelability is set to **CANCEL_OFF**, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled.

When a thread is created, the default general cancelability state is **CANCEL_ON**.

Possible Dangers of Disabling Cancelability

The most important use of cancels is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which may take days to respond (or may never respond), is normally made cancelable.

However, when cancelability is disabled, no routine is cancelable. Waits must be completed normally before a cancel can be delivered. As a result, the program stops working and the user is unable to cancel the operation.

When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancels around that particular region of code.

Return Values

On successful completion, the previous state of general cancelability is returned. If the function fails, −1 is returned. Following are the possible return values and the possible corresponding values (if any) for **errno**:

Return	Error	Description
CANCEL_ON		Asynchronous cancelability was on.
CANCEL_OFF		Asynchronous cancelability was off.
−1	[**EINVAL**]	The specified state is not **CANCEL_ON** or **CANCEL_OFF**.

Related Information

Functions: **pthread_cancel(3thr)**, **pthread_setasynccancel(3thr)**, **pthread_testcancel(3thr)**.

pthread_setprio

Purpose

Changes the current priority of a thread

Synopsis

```
#include <pthread.h>

int pthread_setprio(
        pthread_t thread,
        int priority);
```

Parameters

thread Thread whose priority is changed.

priority New priority value of the thread specified in *thread*. The priority value depends on scheduling policy. Valid values fall within one of the following ranges:

- **PRI_OTHER_MIN** <= *priority* <= **PRI_OTHER_MAX**
- **PRI_FIFO_MIN** <= *priority* <= **PRI_FIFO_MAX**
- **PRI_RR_MIN** <= *priority* <= **PRI_RR_MAX**
- **PRI_FG_MIN_NP** <= *priority* <= **PRI_FG_MAX_NP**
- **PRI_BG_MIN_NP** <= *priority* <= **PRI_BG_MAX_NP**

If you create a new thread without specifying a threads attributes object that contains a changed priority attribute, the default priority of the newly created thread is the midpoint between **PRI_OTHER_MIN** and **PRI_OTHER_MAX** (the midpoint between the minimum and the maximum for the **SCHED_OTHER** policy).

When you call this routine to specify a minimum or maximum priority, use the appropriate symbol; for example, **PRI_FIFO_MIN** or **PRI_FIFO_MAX**. To specify a value between the minimum and maximum, use an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and maximum

for the Round Robin scheduling policy, specify the following concept using your programming language's syntax:

```
pri_rr_mid = (PRI_RR_MIN + PRI_RR_MAX + 1)/2
```

If your expression results in a value outside the range of minimum to maximum, an error results when you use it.

Description

The **pthread_setprio()** routine changes the current priority of a thread. A thread can change its own priority using the identifier returned by **pthread_self()**.

Changing the priority of a thread can cause it to start executing or be preempted by another thread. The effect of setting different priority values depends on the scheduling priority assigned to the thread. The initial scheduling priority is set by calling the **pthread_attr_setprio()** routine.

Note that **pthread_attr_setprio()** sets the priority attribute that is used to establish the priority of a new thread when it is created. However, **pthread_setprio()** changes the priority of an existing thread.

Return Values

If successful, this routine returns the previous priority. If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Previous priority		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.
−1	[**ENOTSUP**]	An attempt is made to set the priority to an unsupported value.
−1	[**ESRCH**]	The value specified by *thread* does not refer to an existing thread.
−1	[**EPERM**]	The caller does not have the appropriate privileges to set the priority of the specified thread.

Related Information

Functions: **pthread_attr_setprio(3thr)**, **pthread_attr_setsched(3thr)**, **pthread_create(3thr)**, **pthread_self(3thr)**, **pthread_setscheduler(3thr)**.

pthread_setscheduler

Purpose

Changes the current scheduling policy and priority of a thread

Synopsis

```
#include <pthread.h>

int pthread_setscheduler(
        pthread_t thread,
        int scheduler,
        int priority);
```

Parameters

thread Thread whose scheduling policy is to be changed.

scheduler New scheduling policy value for the thread specified in *thread*. Valid values are as follows:

SCHED_FIFO
(First In, First Out) The highest-priority thread runs until it blocks. If there is more than one thread with the same priority, and that priority is the highest among other threads, the first thread to begin running continues until it blocks.

SCHED_RR
(Round Robin) The highest-priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. Timeslicing is a process in which threads alternate using available processors.

SCHED_OTHER
(Default) All threads are timesliced. **SCHED_OTHER** ensures that all threads, regardless of priority, receive some scheduling, and thus no thread is completely denied

execution time. (However, **SCHED_OTHER** threads can be denied execution time by **SCHED_FIFO** or **SCHED_RR** threads.)

SCHED_FG_NP
(Foreground) Same as **SCHED_OTHER**. Threads are timesliced and priorities can be modified dynamically by the scheduler to ensure fairness.

SCHED_BG_NP
(Background) Like **SCHED_OTHER**, ensures that all threads, regardless of priority, receive some scheduling. However, **SCHED_BG_NP** can be denied execution by any of the other scheduling policies.

priority New priority value of the thread specified in *thread*. The priority attribute depends on scheduling policy. Valid values fall within one of the following ranges:

- **PRI_OTHER_MIN <=** *priority* <= **PRI_OTHER_MAX**
- **PRI_FIFO_MIN <=** *priority* <= **PRI_FIFO_MAX**
- **PRI_RR_MIN <=** *priority* <= **PRI_RR_MAX**
- **PRI_FG_MIN_NP <=** *priority* <= **PRI_FG_MAX_NP**
- **PRI_BG_MIN_NP <=** *priority* <= **PRI_BG_MAX_NP**

If you create a new thread without specifying a threads attributes object that contains a changed priority attribute, the default priority of the newly created thread is the midpoint between **PRI_OTHER_MIN** and **PRI_OTHER_MAX** (the midpoint between the minimum and the maximum for the **SCHED_OTHER** policy).

When you call this routine to specify a minimum or maximum priority, use the appropriate symbol; for example, **PRI_FIFO_MIN** or **PRI_FIFO_MAX**. To specify a value between the minimum and maximum, use an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and maximum for the Round Robin scheduling policy, specify the following concept using your programming language's syntax:

```
pri_rr_mid = (PRI_RR_MIN + PRI_RR_MAX)/2
```

If your expression results in a value outside the range of minimum to maximum, an error results when you use it.

Description

The **pthread_setscheduler()** routine changes the current scheduling policy and priority of a thread. Call this routine to change both the priority and scheduling policy of a thread at the same time. To change only the priority, call the **pthread_setprio()** routine.

A thread changes its own scheduling policy and priority by using the identifier returned by **pthread_self()**. Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread.

This routine differs from **pthread_attr_setprio()** and **pthread_attr_setsched()** because those routines set the priority and scheduling policy attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the priority and scheduling policy of an existing thread.

Return Values

If successful, the previous scheduling policy value is returned. If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
Previous policy		Successful completion.
–1	[**EINVAL**]	The value specified by *thread* is invalid.
–1	[**ENOTSUP**]	An attempt is made to set the priority to an unsupported value.
–1	[**ESRCH**]	The value specified by *thread* does not refer to an existing thread.
–1	[**EPERM**]	The caller does not have the appropriate privileges to set the scheduling policy of the specified thread.

Related Information

Functions: **pthread_attr_setprio(3thr)**, **pthread_attr_setsched(3thr)**, **pthread_create(3thr)**, **pthread_self(3thr)**, **pthread_setprio(3thr)**.

pthread_setspecific

Purpose

Sets the thread-specific data associated with the specified key for the current thread

Synopsis

```
#include <pthread.h>

int pthread_setspecific(
        pthread_key_t key,
        pthread_addr_t value);
```

Parameters

key Context key value that uniquely identifies the context value specified in *value*. This key value must have been obtained from **pthread_keycreate()**.

value Address containing data to be associated with the specified key for the current thread; this is the thread-specific data.

Description

The **pthread_setspecific()** routine sets the thread-specific data associated with the specified key for the current thread. If a value has already been defined for the key in this thread, the new value is substituted for it.

Different threads can bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

Return Values

If the function fails, –1 is returned, and **errno** may be set to the following value:

Return	Error	Description
–1	[**EINVAL**]	The key value is invalid.

Related Information

Functions: **pthread_getspecific(3thr)**, **pthread_keycreate(3thr)**.

pthread_signal_to_cancel_np

Purpose

Cancels the specified thread

Synopsis

```
#include <pthread.h>

int pthread_signal_to_cancel_np(
        sigset_t *sigset,
        pthread_t *thread);
```

Parameters

sigset	Signal mask containing a list of signals that, when received by the process, cancels the specified thread.
thread	The thread canceled if a valid signal is received by the process.

Description

The **pthread_signal_to_cancel_np()** routine requests that the specified thread be canceled if one of the signals specified in the signal mask is received by the process. The set of legal signals is the same as that for the **sigwait()** service. The *sigset* parameter is not validated. If it is invalid, this routine returns successfully but neither the specified thread nor the previously specified thread is canceled if a signal occurs.

Note that the address of the specified thread is saved in a per-process global variable. Therefore, any subsequent call to this routine by your application or any library function will supercede the thread specified in the previous call, and that thread will not be canceled if one of the signals specified for it is delivered to the process. In other words, take care when you call this routine; if another thread calls it after you do, the expected result of this routine will not occur.

Return Values

If the function fails, **errno** may be set to one of the following values:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by *thread* is invalid.

Related Information

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

pthread_testcancel

Purpose

Requests delivery of a pending cancel to the current thread

Synopsis

```
#include <pthread.h>

void pthread_testcancel();
```

Description

The **pthread_testcancel()** routine requests delivery of a pending cancel to the current thread. The cancel is delivered only if a cancel is pending for the current thread and general cancel delivery is not currently disabled. (A thread disables delivery of cancels to itself by calling the **pthread_setcancel()** routine.)

This routine, when called within very long loops, ensures that a pending cancel is noticed within a reasonable amount of time.

Return Values

No value is returned.

Related Information

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

pthread_unlock_global_np

Purpose

Unlocks a global mutex

Synopsis

```
#include <pthread.h>

void pthread_unlock_global_np( );
```

Description

The **pthread_unlock_global_np()** routine unlocks the global mutex when each call to **pthread_lock_global_np()** is matched by a call to this routine. For example, if you called **pthread_lock_global_np()** three times, **pthread_unlock_global_np()** unlocks the global mutex when you call it the third time. If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, one thread returns from its call to **pthread_lock_global_np()**. The scheduling policy is used to determine which thread acquires the global mutex. For the policies **SCHED_FIFO** and **SCHED_RR**, a blocked thread is chosen in priority order.

The results of calling this routine are unpredictable if the global mutex is already unlocked. The results of calling this routine are also unpredictable if the global mutex is owned by a thread other than the calling thread.

This routine is a new primitive.

Return Values

No value is returned.

Related Information

Functions: **pthread_lock_global_np(3thr)**, **pthread_mutex_lock(3thr)**, **pthread_mutex_unlock(3thr)**, **pthread_mutexattr_setkind_np(3thr)**.

pthread_yield

Purpose

Notifies the scheduler that the current thread is willing to release its processor

Synopsis

```
#include <pthread.h>

void pthread_yield( );
```

Description

The **pthread_yield()** routine notifies the scheduler that the current thread is willing to release its processor to other threads of the same priority. (A thread releases its processor to a thread of a higher priority without calling this routine.)

If the current thread's scheduling policy (as specified in a call to the **pthread_attr_setsched()** or **pthread_setscheduler()** routine) is **SCHED_FIFO** or **SCHED_RR**, this routine yields the processor to other threads of the same or a higher priority. If no threads of the same priority are ready to execute, the thread continues.

This routine allows knowledge of the details of an application to be used to increase fairness. It increases fairness of access to the processor by removing the current thread from the processor. It also increases fairness of access to shared resources by removing the current thread from the processor as soon as it is finished with the resource.

Call this routine when a thread is executing code that denies access to other threads on a uniprocessor if the scheduling policy is **SCHED_FIFO**.

Use **pthread_yield()** carefully because misuse causes unnecessary context switching, which increases overhead without increasing fairness. For example, it is counterproductive for a thread to yield while it has a needed resource locked.

Return Values

No value is returned.

Related Information

Functions: **pthread_attr_setsched(3thr)**, **pthread_setscheduler(3thr)**.

sigaction

Purpose

Examines and changes synchronous signal actions (POSIX software signal facilities)

Synopsis

```
#include <signal.h>

struct sigaction {
        void  (*sa_handler);
        sigset_t  sa_mask;
        int  sa_flags;
};

int sigaction(sig, act, oact)
int  sig;
const struct sigaction  *act;
struct sigaction  *oact;
```

Parameters

sig	Synchronous signal to examine or change.
act	Points to a **sigaction** structure that describes the action to be taken upon receipt of the signal indicated by the value of the *act* parameter.
oact	Points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction()** function call is returned.

Description

The **sigaction** POSIX service allows for per-thread handlers to be installed for catching synchronous signals. It is called in a multithreaded process to establish thread specific actions for such signals. This call is the POSIX equivalent of the **sigaction()** system call with the following exceptions or modifications:

- The **sigaction()** routine only modifies behavior for individual threads.
- The **sigaction()** routine only works for synchronous signals. Attempting to set a signal action for an asynchronous signal is an error. This is true even in a single-threaded process.

  Any multithreaded application using DCE Threads will need to use the **sigwait()** function for dealing with asynchronous signals. The **sigwait()** function can be used to synchronously wait for delivery of asynchronously generated signals.
- The **SA_RESTART** flag is always set by the underlying system in POSIX mode so that interrupted system calls will fail with return value of −1 and the **EINTR** error in *errno* instead of getting restarted.

  The system's **SA_RESTART** flag has the opposite meaning of the **SA_RESTART** flag in the *sa_flags* field and is always set in the underlying system call resulting from **sigaction()** regardless of whether **SA_RESTART** was indicated in *sa_flags*.
- The signal mask is manipulated using the POSIX § 3.3.3 **sigsetops()** functions. They are **sigemptyset()**, **sigfillset()**, **sigaddset()**, **sigdelset()**, and **sigismember()**.

The **sigaction()** function can be used to inquire about the current handling of a given signal by specifying a null pointer for *act*, since the action is unchanged unless this parameter is not a null pointer. In order for the signal action in effect at the time of the **sigaction()** call to be returned, the *oact* parameter must not be a null pointer.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
−1	[**EFAULT**]	Either *act* or *oact* points to memory which is not a valid part of the process address space.

		A new signal handler is not installed.
–1	[**EINVAL**]	The value specified by *sig* is invalid. A new signal handler is not installed.
–1	[**EINVAL**]	An attempt is made to ignore or supply a handler for **SIGKILL** or **SIGSTOP**. A new signal handler is not installed.

Related Information

Functions: **setjmp(3)**, **siginterrupt(3)**, **sigpending(3thr)**, **sigprocmask(3thr)**, **sigsetops(3)**, **sigsuspend(3)**, **sigvec(2)**, **tty(4)**.

sigpending

Purpose

Examines pending signals (POSIX software signal facilities)

Synopsis

#include <signal.h>

int sigpending(sigset_t **set***;**

Parameters

set Points to a location in which the signals that are blocked from delivery and pending at the time of the **sigpending()** function call are returned.

Description

The **sigpending()** function stores the set of signals that are blocked from delivery and pending for the calling process in the space pointed to by the argument *set*.

The **sigpending()** function may be called by any thread in a multithreaded process to determine which signals are in the pending set for that thread. Since DCE Threads supports the **{_POSIX_THREADS_PER_PROCESS_SIGNALS_1}** option, signals pending upon the thread are those that are pending upon the process.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
−1	[**EFAULT**]	The *set* argument points to memory that is not a valid part of the process address space.

Related Information

Functions: **sigprocmask(3thr)**, **sigsetops(3)**.

sigprocmask

Purpose

Examines and changes blocked signals (POSIX software signal facilities)

Synopsis

```
#include <signal.h>

int sigprocmask(int how const sigset_t *set sigset_t *oset);
```

Parameters

how — The manner in which the values in *set* are changed as defined by one of the described argument values.

set — A set of signals that will be used to change the current thread's signal mask according to the value in the *how* parameter.

oset — Points to a location in which the signal mask in effect at the time of the **sigprocmask()** function call is returned.

Description

The **sigprocmask()** function is used to examine or change (or both) the signal mask of the calling process. If the value of the argument *set* is not NULL, it points to a set of signals to be used to change the currently blocked set according to the *how* parameter as follows:

SIG_BLOCK
The resulting signal set is the union of the current set and the signal set pointed to by the argument *set*.

SIG_UNBLOCK
The resulting signal set is the intersection of the current set and the and the complement of the signal set pointed to by the argument *set*.

SIG_SETMASK
The resulting signal set is the signal set pointed to by the argument *set*.

If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*.

The **sigprocmask()** function can be used to inquire about the currently blocked signals by specifying a null pointer for *set*, since the value of the argument *how* is not significant and the signal mask of the process is unchanged unless this parameter is not a null pointer. In order for the signal mask in effect at the time of the **sigprocmask()** call to be returned, the *oset* argument must not be a null pointer.

If there are any pending unblocked signals after the call to the **sigprocmask()** function, at least one of those signals shall be delivered before the **sigprocmask()** function returns. As a system restriction, the SIGKILL and SIGSTOP signals cannot be blocked.

If the **sigprocmask()** function fails, the signal mask of the process is not changed by this function call.

Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
−1	[**EINVAL**]	The value specified by the *how* parameter is not equal to one of the defined values. The signal mask of the process remains unchanged.

Related Information

Functions: **sigaction(3thr)**, **sigpending(3thr)**, **sigsetops(3)**, **sigsuspend(3)**.

sigwait

Purpose

Causes a thread to wait for an asynchronous signal

Synopsis

```
#include <pthread.h>

int sigwait(
        sigset_t *set);
```

Parameters

set	Set of pending signals upon which the calling thread will wait.

Description

This routine causes a thread to wait for an asynchronous signal. It atomically chooses a pending signal from *set*, atomically clears it from the system's set of pending signals and returns that signal number. If no signal in *set* is pending at the time of the call, the thread is blocked until one or more signals becomes pending. The signals defined by *set* may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

A thread must block the signals it waits for using **sigprocmask**() prior to calling this function.

If more than one thread is using this routine to wait for the same signal, only one of these threads will return from this routine with the signal number.

A call to **sigwait**() is a cancellation point.

Return Values

Possible return values are as follows:

Return	Error	Description
Signal number		Successful completion.
−1	[**EINVAL**]	One or more of the values specified by *set* is invalid.
−1	[**EINVAL**]	One or more of the values specified by *set* is not blocked.
−1	[**EINVAL**]	There are no values specified in *set*.

Related Information

Functions: **pause(3)**, **pthread_cancel(3thr)**, **pthread_setasynccancel(3thr)**, **sigpending(3)**, **sigprocmask(3)**, **sigsetops(3)**.

Chapter 3

DCE Remote Procedure Call

rpc_intro

Purpose

Introduction to the DCE RPC API runtime

Description

This introduction gives general information about the DCE RPC application programming interface (API) and an overview of the following parts of the DCE RPC API runtime:

- Runtime services
- Environment variables
- Data types and structures
- Permissions required
- Frequently used routine arguments

General Information

The following subsections contain topics, beyond those directly related to the RPC API, that application programmers need to know.

IDL-to-C Mappings

The Interface Definition Language (IDL) compiler converts an interface definition into output files. The **rpc_intro(1rpc)** reference page in the *DCE 1.2.2 Command Reference* contains a summary of the **idl** command, which invokes the IDL compiler.

Additional information about the IDL compiler appears in the following table, which shows the IDL base types and the IDL-to-C mappings.

The following table lists the IDL base data type specifiers. Where applicable, the table shows the size of the corresponding transmittable type and the type macro emitted by the IDL compiler for resulting declarations.

Base Data Type Specifiers—rpc_intro(3rpc)				
Specifier				**Type Macro**
(sign)	**(size)**	**(type)**	**Size**	**Emitted by idl**
	small	**int**	8 bits	**idl_small_int**
	short	**int**	16 bits	**idl_short_int**
	long	**int**	32 bits	**idl_long_int**
	hyper	**int**	64 bits	**idl_hyper_int**
unsigned	**small**	**int**	8 bits	**idl_usmall_int**
unsigned	**short**	**int**	16 bits	**idl_ushort_int**
unsigned	**long**	**int**	32 bits	**idl_ulong_int**
unsigned	**hyper**	**int**	64 bits	**idl_uhyper_int**
		float	32 bits	**idl_short_float**
		double	64 bits	**idl_long_float**
		char	8 bits	**idl_char**
		boolean	8 bits	**idl_boolean**
		byte	8 bits	**idl_byte**
		void	—	**idl_void_p_t**
		handle_t	—	**—**

Note that you can use the **idl_** macros in the code you write for an application to ensure that your type declarations are consistent with those in the stubs, even when the application is ported to another platform. The **idl_** macros are especially useful when passing constant values to RPC calls. For maximum portability, all constants passed to RPC calls declared in your network interfaces should be cast to the appropriate type because the size of integer constants (like the size of the **int** data type) is unspecified in the C language.

The **idl_** macros are defined in **dce/idlbase.h**, which is included by header files that the IDL compiler generates.

Management Commands for Programmers

In addition to the **idl** command for programmers, DCE RPC provides two management commands for the RPC control program and the DCE host daemon, as follows:

- The **rpccp** control program accesses the RPC control program (RPCCP). This program provides a set of commands for accessing the operations of the RPC Name Service Interface (NSI). RPCCP also supports showing the elements of the local endpoint map and removing elements from it.

  You can manage the name service with RPCCP commands or with DCE RPC runtime routines. For example, suppose you want to obtain the members of a group. You can give the **show group** command to RPCCP or you can write an application program that calls the following DCE RPC runtime routines:

  — **rpc_ns_group_mbr_inq_begin()**

  — **rpc_ns_group_mbr_inq_next()**

  — **rpc_ns_group_mbr_inq_done()**

- The **dced** command starts the DCE host daemon. The daemon maintains the local endpoint map for RPC servers and looks up endpoints for RPC clients.

See the *DCE 1.2.2 Command Reference* for more information about these two management commands.

Overview of DCE RPC Runtime Services

The RPC runtime services consist of RPC routines that perform a variety of operations.

Note that the RPC API is thread safe and synchronous cancel safe (in the context of POSIX threads). However, the RPC API is not asynchronous cancel safe. For more information about threads and their cancellation, see the *DCE 1.2.2 Application Development Guide—Core Components*.

The rest of this overview consists of the following items:

- An explanation of abbreviations in the names of the RPC runtime routines
- An alphabetical list of DCE RPC runtime routines. With each routine name is its description and the type of application program that most likely calls the routine.

An alphabetical list of abbreviations in the names of the DCE RPC routines follows. The list can help you remember the names more easily. For example, consider the routine name **rpc_mgmt_ep_elt_inq_begin()**. Use the next list to expand the name to ''RPC management endpoint element inquiry begin,'' which summarizes the description ''Creates an inquiry context for viewing the elements in a local or remote endpoint map. (Management).''

auth	authentication, authorization
com	communications
cs	character/code set interoperability
dce	distributed computing environment
dflt	default
elt	element
ep	endpoint
exp	expiration
fn	function
id	identifier
idl_es	IDL encoding services
if	interface
inq	inquiry
mbr	member
mgmt	management
ns	name service
protseq	protocol sequence
rgy	DCE character and code set registry
rpc	remote procedure call
stats	statistics

An alphabetical list of the RPC runtime routines follows. With each routine name is its description and the type of application program that most likely calls the routine.

cs_byte_from_netcs()
Converts international character data from a network code set to a local code set. (Client, server).

cs_byte_local_size()
Calculates the necessary buffer size for a code set conversion from a network code set to a local code set. (Client, server).

cs_byte_net_size()
Calculates the necessary buffer size for a code set conversion from a local code set to a network code set. (Client, server).

cs_byte_to_netcs()
Converts international character data from a local code set to a network code set. (Client, server).

dce_cs_loc_to_rgy()
Maps a local name for a code set to a code set value in the code set registry. (Client, server).

dce_cs_rgy_to_loc()
Maps a code set value in the code set registry to a the local name for a code set. (Client, server).

idl_es_decode_buffer()
Returns a buffer decoding handle. (Client, server).

idl_es_decode_incremental()
Returns an incremental decoding handle. (Client, server).

idl_es_encode_dyn_buffer()
Returns a dynamic buffer encoding handle. (Client, server).

idl_es_encode_fixed_buffer()
Returns a fixed buffer encoding handle. (Client, server).

idl_es_encode_incremental()
Returns an incremental encoding handle. (Client, server).

idl_es_handle_free()
Frees an IDL encoding services handle. (Client, server).

idl_es_inq_encoding_id()
Identifies an application encoding operation. (Client, server).

rpc_binding_copy()
Returns a copy of a binding handle. (Client or server).

rpc_binding_free()
Releases binding handle resources. (Client or server).

rpc_binding_from_string_binding()
Returns a binding handle from a string representation of a binding handle. (Client or management).

rpc_binding_inq_auth_client()
Returns authentication and authorization information from the binding handle for an authenticated client. (Server).

rpc_binding_inq_auth_info()
Returns authentication and authorization information from a server binding handle. (Client).

rpc_binding_inq_object()
Returns the object UUID from a binding handle. (Client or server).

rpc_binding_reset()
Resets a server binding handle so the host remains specified, but the server instance on that host is unspecified. (Client or management).

rpc_binding_server_from_client()
Converts a client binding handle to a server binding handle. (Server).

rpc_binding_set_auth_info()
Sets authentication and authorization information into a server binding handle. (Client).

rpc_binding_set_object()
Sets the object UUID value into a server binding handle. (Client).

rpc_binding_to_string_binding()
Returns a string representation of a binding handle. (Client, server, or management).

rpc_binding_vector_free()
Frees the memory used to store a vector and binding handles. (Client or server).

rpc_cs_binding_set_tags()
Places code set tags into a server binding handle. (Client).

rpc_cs_char_set_compat_check()
Evaluates character set compatibility between a client and a server. (Client).

rpc_cs_eval_with_universal()
Evaluates a server's supported character sets and code sets during the server binding selection process. (Client).

rpc_cs_eval_without_universal()
Evaluates a server's supported character sets and code sets during the server binding selection process. (Client).

rpc_cs_get_tags()
Retrieves code set tags from a binding handle. (Client, server).

rpc_ep_register()
Adds to, or replaces, server address information in the local endpoint map. (Server).

rpc_ep_register_no_replace()
Adds to server address information in the local endpoint map. (Server).

rpc_ep_resolve_binding()
Resolves a partially bound server binding handle into a fully bound server binding handle. (Client or management).

rpc_ep_unregister()
Removes server address information from the local endpoint map. (Server).

rpc_if_id_vector_free()
Frees a vector and the interface identifier structures it contains. (Client, server, or management).

rpc_if_inq_id()
Returns the interface identifier for an interface specification. (Client or server).

rpc_mgmt_ep_elt_inq_begin()
Creates an inquiry context for viewing the elements in a local or remote endpoint map. (Management).

rpc_mgmt_ep_elt_inq_done()
Deletes the inquiry context for viewing the elements in a local or remote endpoint map. (Management).

rpc_mgmt_ep_elt_inq_next()
Returns one element at a time from a local or remote endpoint map. (Management).

rpc_mgmt_ep_unregister()
Removes server address information from a local or remote endpoint map. (Management).

rpc_mgmt_inq_com_timeout()
Returns the communications timeout value in a binding handle. (Client).

rpc_mgmt_inq_dflt_protect_level()
Returns the default protection level for an authentication service. (Client or server).

rpc_mgmt_inq_if_ids()
Returns a vector of interface identifiers of interfaces a server offers. (Client, server, or management).

rpc_mgmt_inq_server_princ_name()
Returns a server's principal name. (Client, server, or management).

rpc_mgmt_inq_stats()
Returns RPC runtime statistics. (Client, server, or management).

rpc_mgmt_is_server_listening()
Tells whether a server is listening for remote procedure calls. (Client, server, or management).

rpc_mgmt_set_authorization_fn()
Establishes an authorization function for processing remote calls to a server's management routines. (Server).

rpc_mgmt_set_cancel_timeout()
Sets the lower bound on the time to wait before timing out after forwarding a cancel. (Client).

rpc_mgmt_set_com_timeout()
Sets the communications timeout value in a binding handle. (Client).

rpc_mgmt_set_server_stack_size()
Specifies the stack size for each server thread. (Server).

rpc_mgmt_stats_vector_free()
Frees a statistics vector. (Client, server, or management).

rpc_mgmt_stop_server_listening()
Tells a server to stop listening for remote procedure calls. (Client, server, or management).

rpc_network_inq_protseqs()
Returns all protocol sequences supported by both the RPC runtime and the operating system. (Client or server).

rpc_network_is_protseq_valid()
Tells whether the specified protocol sequence is supported by both the RPC runtime and the operating system. (Client or server).

rpc_ns_binding_export()
Establishes a name service database entry with binding handles or object UUIDs for a server. (Server).

rpc_ns_binding_import_begin()
Creates an import context for an interface and an object in the name service database. (Client).

rpc_ns_binding_import_done()
Deletes the import context for searching the name service database. (Client).

rpc_ns_binding_import_next()
Returns a binding handle of a compatible server (if found) from the name service database. (Client).

rpc_ns_binding_inq_entry_name()
Returns the name of an entry in the name service database from which the server binding handle came. (Client).

rpc_ns_binding_lookup_begin()
Creates a lookup context for an interface and an object in the name service database. (Client).

rpc_ns_binding_lookup_done()
Deletes the lookup context for searching the name service database. (Client).

rpc_ns_binding_lookup_next()
Returns a list of binding handles of one or more compatible servers (if found) from the name service database. (Client).

rpc_ns_binding_select()
Returns a binding handle from a list of compatible server binding handles. (Client).

rpc_ns_binding_unexport()
Removes the binding handles for an interface, or object UUIDs, from an entry in the name service database. (Server).

rpc_ns_entry_expand_name()
Expands the name of a name service entry. (Client, server, or management).

rpc_ns_entry_object_inq_begin()
Creates an inquiry context for viewing the objects of an entry in the name service database. (Client, server, or management).

rpc_ns_entry_object_inq_done()
Deletes the inquiry context for viewing the objects of an entry in the name service database. (Client, server, or management).

rpc_ns_entry_object_inq_next()
Returns one object at a time from an entry in the name service database. (Client, server, or management).

rpc_ns_group_delete()
Deletes a group attribute. (Client, server, or management).

rpc_ns_group_mbr_add()
Adds an entry name to a group; if necessary, creates the entry. (Client, server, or management).

rpc_ns_group_mbr_inq_begin()
Creates an inquiry context for viewing group members. (Client, server, or management).

rpc_ns_group_mbr_inq_done()
Deletes the inquiry context for a group. (Client, server, or management).

rpc_ns_group_mbr_inq_next()
Returns one member name at a time from a group. (Client, server, or management).

rpc_ns_group_mbr_remove()
Removes an entry name from a group. (Client, server, or management).

rpc_ns_import_ctx_add_eval()
Adds an evaluation routine to an import context. (Client).

rpc_ns_mgmt_binding_unexport()
Removes multiple binding handles, or object UUIDs, from an entry in the name service database. (Management).

rpc_ns_mgmt_entry_create()
Creates an entry in the name service database. (Management).

rpc_ns_mgmt_entry_delete()
Deletes an entry from the name service database. (Management).

rpc_ns_mgmt_entry_inq_if_ids()
Returns the list of interfaces exported to an entry in the name service database. (Client, server, or management).

rpc_ns_mgmt_free_codesets()
Frees a code sets array that has been allocated in memory. (Client).

rpc_ns_mgmt_handle_set_exp_age()
Sets a handle's expiration age for local copies of name service data. (Client, server, or management).

rpc_ns_mgmt_inq_exp_age()
Returns the application's global expiration age for local copies of name service data. (Client, server, or management).

rpc_ns_mgmt_read_codesets()
Reads the code sets attribute associated with an RPC server entry in the name service database. (Client).

rpc_ns_mgmt_remove_attribute()
Removes an attribute from an RPC server entry in the name service database. (Server, management).

rpc_ns_mgmt_set_attribute()
Adds an attribute to an RPC server entry in the name service database. (Server, management).

rpc_ns_mgmt_set_exp_age()
Modifies the application's global expiration age for local copies of name service data. (Client, server, or management).

rpc_ns_profile_delete()
Deletes a profile attribute. (Client, server, or management).

rpc_ns_profile_elt_add()
Adds an element to a profile. If necessary, creates the entry. (Client, server, or management).

rpc_ns_profile_elt_inq_begin()
Creates an inquiry context for viewing the elements in a profile. (Client, server, or management).

rpc_ns_profile_elt_inq_done()
Deletes the inquiry context for a profile. (Client, server, or management).

rpc_ns_profile_elt_inq_next()
Returns one element at a time from a profile. (Client, server, or management).

rpc_ns_profile_elt_remove()
Removes an element from a profile. (Client, server, or management).

rpc_object_inq_type()
Returns the type of an object. (Server).

rpc_object_set_inq_fn()
Registers an object inquiry function. (Server).

rpc_object_set_type()
Assigns the type of an object. (Server).

rpc_protseq_vector_free()
Frees the memory used by a vector and its protocol sequences. (Client or server).

rpc_rgy_get_codesets()
Gets supported code sets information from the local host. (Client, server).

rpc_rgy_get_max_bytes()
Gets the maximum number of bytes that a code set uses to encode one character. (Client, server).

rpc_server_inq_bindings()
Returns binding handles for communication with a server. (Server).

rpc_server_inq_if()
Returns the manager entry point vector registered for an interface. (Server).

rpc_server_listen()
Tells the RPC runtime to listen for remote procedure calls. (Server).

rpc_server_register_auth_info()
Registers authentication information with the RPC runtime. (Server).

rpc_server_register_if()
Registers an interface with the RPC runtime. (Server).

rpc_server_unregister_if()
Unregisters an interface from the RPC runtime. (Server).

rpc_server_use_all_protseqs()
Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls. (Server).

rpc_server_use_all_protseqs_if()
Tells the RPC runtime to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls. (Server).

rpc_server_use_protseq()
Tells the RPC runtime to use the specified protocol sequence for receiving remote procedure calls. (Server).

rpc_server_use_protseq_ep()
Tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls. (Server).

rpc_server_use_protseq_if()
Tells the RPC runtime to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls. (Server).

rpc_sm_allocate()
Allocates memory within the RPC stub memory management scheme. (Usually server, possibly client).

rpc_sm_client_free()
Frees memory allocated by the current memory allocation and freeing mechanism used by the client stubs. (Client).

rpc_sm_destroy_client_context()
Reclaims the client memory resources for a context handle, and sets the context handle to NULL. (Client).

rpc_sm_disable_allocate()
Releases resources and allocated memory within the RPC stub memory management scheme. (Client).

rpc_sm_enable_allocate()
Enables the stub memory management environment. (Client).

rpc_sm_free()
Frees memory allocated by the **rpc_sm_allocate()** routine. (Usually server, possibly client).

rpc_sm_get_thread_handle()
Gets a thread handle for the stub memory management environment. (Usually server, possibly client).

rpc_sm_set_client_alloc_free()
Sets the memory allocation and freeing mechanism used by the client stubs. (Client).

rpc_sm_set_thread_handle()
Sets a thread handle for the stub memory management environment. (Usually server, possibly client).

rpc_sm_swap_client_alloc_free()
Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client. (Client).

rpc_string_binding_compose()
Combines the components of a string binding into a string binding. (Client or server).

rpc_string_binding_parse()
Returns, as separate strings, the components of a string binding. (Client or server).

rpc_string_free()
Frees a character string allocated by the runtime. (Client, server, or management).

uuid_compare()
Compares two UUIDs and determines their order. (Client, server, or management).

uuid_create()
Creates a new UUID. (Client, server, or management).

uuid_create_nil()
Creates a nil UUID. (Client, server, or management).

uuid_equal()
Determines if two UUIDs are equal. (Client, server, or management).

uuid_from_string()
Converts a string UUID to its binary representation. (Client, server, or management).

uuid_hash()
Creates a hash value for a UUID. (Client, server, or management).

uuid_is_nil()
Determines if a UUID is nil. (Client, server, or management).

uuid_to_string()
Converts a UUID from a binary representation to a string representation. (Client, server, or management).

wchar_t_from_netcs()
Converts international character data from a network code set to a local code set. (Client, server).

wchar_t_local_size()
Calculates the necessary buffer size for a code set conversion from a network code set to a local code set. (Client, server).

wchar_t_net_size()
Calculates the necessary buffer size for a code set conversion from a local code set to a network code set. (Client, server).

wchar_t_to_netcs()
Converts international character data from a local code set to a network code set. (Client, server).

Environment Variables

The RPC NSI routines use the following environment variables:

- **RPC_DEFAULT_ENTRY**

  Designates the default entry in the name service database that the import and lookup routines use as the starting point to search for binding information for a compatible server. Normally, the starting entry is a profile.

  An application that uses a default entry name must define this environment variable. The RPC runtime does not provide a default.

  For example, suppose that a client application needs to search the name service database for a server binding handle. The application can use the **rpc_ns_binding_import_begin()** routine as part of the search. If so, the application must specify, to the routine's *entry_name* parameter, the name of the entry in the name service database at which to begin the search. If the search is to begin at the entry that the **RPC_DEFAULT_ENTRY** environment variable specifies, then the application must specify the value NULL to parameter *entry_name* in **rpc_ns_binding_import_begin()**.

- **RPC_DEFAULT_ENTRY_SYNTAX**

  Specifies the syntax of the name provided in the **RPC_DEFAULT_ENTRY** environment variable. In addition, provides the syntax for those RPC NSI routines that allow a default value for the name syntax argument.

  If the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable is not defined, the RPC runtime uses the **rpc_c_ns_syntax_dce** name syntax.

  (For the valid name syntaxes in this reference page and for the valid syntax values, see the table in the description of the frequently used routine argument *name_syntax*, which appears later in this reference page.)

Optionally, each application defines either or both of the first two environment variables. The application can change the value of either one, or both, at any time during runtime.

RPC Data Types and Structures

The following subsections contain the data types and structures used by client, server, and management application programs.

Much of the information in this section is derived from the *DCE 1.2.2 Application Development Guide*. You may want to refer to the appropriate volume of this book as you read this section. For example, this section contains a brief description of a binding handle. The *DCE 1.2.2 Application Development Guide—Core Components* explains binding handles in detail. It also explains concepts related to binding handles, such as binding information and string bindings.

Binding Handle

A binding handle is a pointer-size opaque variable containing information the RPC runtime uses to manage binding information. The RPC runtime uses binding information to establish a client/server relationship that allows the execution of remote procedure calls.

Based on the context where it is created, a binding handle is considered a server binding handle or a client binding handle.

A server binding handle is a reference to the binding information necessary for a client to establish a relationship with a specific server. Many RPC API runtime routines return a server binding handle that you can use to make a remote procedure call.

A server binding handle refers to several components of binding information. One is the network address of a server's host system. Each server instance has one or more transport addresses (endpoints). A well-known endpoint is a stable address on the host, while a dynamic endpoint is an address that the RPC runtime requests for the server. Some transport protocols provide fewer well-known endpoints than dynamic endpoints.

If binding information contains an endpoint, the corresponding binding handle is a fully bound binding handle. If the information lacks an endpoint, the binding handle is a partially bound binding handle.

The RPC runtime creates and provides a client binding handle to a called remote procedure as the **handle_t** parameter. The client binding handle contains information about the calling client. A client binding handle cannot be used to make a remote procedure call. A server uses the client binding handle. The **rpc_binding_server_from_client()** routine converts a client binding handle to a server binding handle. You can use the resulting server binding handle to make a remote procedure call.

For an explanation of making a remote procedure call with a partially bound binding handle, see the *DCE 1.2.2 Application Development Guide—Core Components*. For an explanation of failures associated with such a call, see the explanation of status code **rpc_s_wrong_boot_time** in the *DCE 1.2.2 Problem Determination Guide*.

Binding information can contain an object UUID. The default object UUID associated with a binding handle is a nil UUID. Clients can obtain

a nonnil UUID in various ways, such as from a string representation of binding information (a string binding), or by importing it.

The following table contains the RPC runtime routines that operate on binding handles. The table also specifies the type of binding handle, client or server, allowed.

Client and Server Binding Handles		
Routine	**Input Argument**	**Output Argument**
rpc_binding_copy()	Server	Server
rpc_binding_free()	Server	None
rpc_binding_from_string_binding()	None	Server
rpc_binding_inq_auth_client()	Client	None
rpc_binding_inq_auth_info()	Server	None
rpc_binding_inq_object()	Server or client	None
rpc_binding_reset()	Server	None
rpc_binding_server_from_client()	Client	Server
rpc_binding_set_auth_info()	Server	None
rpc_binding_set_object()	Server	None
rpc_binding_to_string_binding()	Server or client	None
rpc_binding_vector_free()	Server	None
rpc_ns_binding_export()	Server	None
rpc_ns_binding_import_next()	None	Server
rpc_ns_binding_inq_entry_name()	Server	None
rpc_ns_binding_lookup_next()	None	Server
rpc_ns_binding_select()	Server	Server
rpc_server_inq_bindings()	None	Server

If the input argument type is only a client or only a server, the routines return the status code **rpc_s_wrong_kind_of_binding** when an application provides the incorrect binding handle type.

An application can share a single binding handle across multiple threads of execution. The RPC runtime, instead of the application, manages binding handle concurrency control across concurrent remote procedure calls that use a single binding handle. However, the client application has responsibility for binding handle concurrency control for operations that read or modify a binding handle.

The related routines are as follows:

- **rpc_binding_free()**
- **rpc_binding_reset()**
- **rpc_binding_set_auth_info()**
- **rpc_binding_set_object()**
- **rpc_ep_resolve_binding()**
- **rpc_mgmt_set_com_timeout()**

For example, suppose an application shares a binding handle across two threads of execution and it resets the binding handle endpoint in one of the threads (by calling **rpc_binding_reset()**). The binding handle in the other thread is then also reset. Similarly, freeing the binding handle in one thread (by calling **rpc_binding_free()**) frees the binding handle in the other thread.

If you do not want this effect, your application can create a copy of a binding handle by calling **rpc_binding_copy()**. An operation on one binding handle then has no effect on the second binding handle.

Clients and servers can access and set object UUIDs by using **rpc_binding_inq_object()** and **rpc_binding_set_object()**.

Routines requiring a binding handle as an argument show a data type of **rpc_binding_handle_t**. Binding handle arguments are passed by value.

Binding Vector

The binding vector data structure contains a list of binding handles over which a server application can receive remote procedure calls.

The binding vector contains a count member (*count*), followed by an array of binding handle (*binding_h*) elements.

The C language representation of a binding vector is as follows:

```
typedef struct {
        unsigned32     count;
        rpc_binding_handle_t  binding_h[1];
        } rpc_binding_vector_t;
```

The RPC runtime creates binding handles when a server application registers protocol sequences. To obtain a binding vector, a server application calls the **rpc_server_inq_bindings()** routine.

A client application obtains a binding vector of compatible servers from the name service database by calling the routine **rpc_ns_binding_lookup_next()**.

In both routines, the RPC runtime allocates memory for the binding vector. An application calls the **rpc_binding_vector_free()** routine to free the binding vector.

An application, when it is finished with an individual binding handle in a binding vector, frees the binding handle by calling **rpc_binding_free()**. This routine also sets the corresponding pointer in the binding vector to NULL.

Note that you should not decrement the *count* field in a binding vector structure when you call the **rpc_binding_free()** routine to free an individual binding handle.

The following routines require a binding vector and show an argument data type of **rpc_binding_vector_t**:

- **rpc_binding_vector_free()**
- **rpc_ep_register()**
- **rpc_ep_register_no_replace()**
- **rpc_ep_unregister()**
- **rpc_ns_binding_export()**
- **rpc_ns_binding_lookup_next()**
- **rpc_ns_binding_select()**
- **rpc_server_inq_bindings()**

Boolean
: Routines that require a Boolean-valued argument or return a Boolean value show a data type of **boolean32**. DCE RPC provides the integer constants TRUE (1) and FALSE (0) for use as Boolean values.

Code Set
: A code set is a mapping of the members of a character set to specific numeric code values. Different code sets use different numeric code values to represent the same character. In general, operating systems use string names to refer to the code sets that the system supports. It is common for different operating systems to use different string names to refer to the same code set.

  Distributed applications that run in a network of heterogeneous operating systems need to be able to identify the character sets and code sets that client and server machines are using to avoid losing data during communications between each other.

  DCE RPC supports transparent automatic conversion for characters that are members of the DCE Portable Character Set (DCE PCS) and which are encoded in the ASCII and U.S. EBCDIC code sets. The RPC runtime automatically converts DCE PCS characters encoded in ASCII or U.S. EBCDIC, if necessary, when they are passed over the network between client and server.

  DCE RPC applications that need to transfer character data that is outside the DCE PCS character set and ASCII and U.S. EBCDIC encodings (international characters) can use special IDL constructs and a set of DCE RPC routines to set up their applications so that they can pass this international character data with minimal or no loss between client and server applications. An example of such an application would be one that used European, Chinese, or Japanese characters mapped to EUC, Big5, or SJIS encodings. Together, the IDL constructs and the DCE RPC routines provide a method of automatic code set conversion for applications that transfer international character data in heterogeneous code set environments.

  DCE provides a mechanism to uniquely identify a code set; this mechanism is the code set registry. The code set registry assigns a unique identifier to each character set and code set. Because the registry provides code set identifiers that are consistent across a network of heterogeneous operating systems, it provides a method for clients and servers in a heterogeneous environment to use to identify code sets without having to rely on operating system-specific string names.

The code set data structure contains a 32-bit hexadecimal value (*c_set*) that uniquely identifies the code set followed by a 16-bit decimal value (*c_max_bytes*) that indicates the maximum number of bytes this code set uses to encode one character in this code set.

The value for *c_set* is one of the registered values in the code set registry.

The following routines require a code set value:

- **cs_byte_from_netcs()**
- **cs_byte_local_size()**
- **cs_byte_net_size()**
- **cs_byte_to_netcs()**
- **dce_cs_loc_to_rgy()**
- **dce_cs_rgy_to_loc()**
- **rpc_cs_get_tags()**
- **rpc_cs_binding_set_tags()**
- **rpc_rgy_get_max_bytes()**
- **wchar_t_from_netcs()**
- **wchar_t_local_size()**
- **wchar_t_net_size()**
- **wchar_t_to_netcs()**

In these routines, the code set value shows a data type of **unsigned32**.

The RPC stub buffer sizing routines ***_net_size()** and ***_local_size** use the value of *c_max_bytes* to calculate the size of a buffer for code set conversion.

The C language representation of a code set structure is as follows:

```
typedef struct {
        long            c_set;
        short           c_max_bytes;
} rpc_cs_c_set_t;
```

The code set data structure is a member of the code sets array.

Code Sets Array

The code sets array contains the list of the code sets that a client or server supports. The structure consists of a version number member (*version*), followed by a count member (*count*), followed by an array of code set data structures (*rpc_cs_c_set_t*). This array is declared to be a conformant array so that its size will be determined at runtime. The *count* member indicates the number of code sets contained in the array.

The first element in the code sets array represents the client or server process's local code set.

The second element through the *n*th element represents one or more intermediate code sets that the process can use to transmit character data over the network. Client or server processes can convert into an intermediate code set when their host system does not provide a converter for the other's local code set but does provide a converter for the intermediate code set.

DCE RPC routines for character/code sets compatibility evaluation and code set conversion support one intermediate code set, which is the ISO 10646 Universal character/code set. Consequently, DCE requires host systems running applications that transfer international characters to provide converters for this code set.

System administrators for machines in internationalized DCE cells (that is, cells of machines that run applications that use the DCE character/code sets compatibility evaluation and conversion functionality) and who want to use other intermediate code sets can run the **csrc** utility and specify that their intermediate code set(s) be used in preference to ISO 10646.

The remaining elements in the array represent other code sets that the process's host supports (that is, code sets for which the system provides converters).

The C language representation of a code set structure is as follows:

```
typedef struct rpc_codeset_mgmt_t {
    unsigned32      version;
    long            count;
```

```
    [size_is(count)] rpc_cs_c_set_t codesets[];
} rpc_codeset_mgmt_t, *rpc_codeset_mgmt_p_t;
```

Client and server applications and DCE RPC routines for automatic code set conversion obtain a code sets array by calling the routine **rpc_rgy_get_codesets()**. Server applications user the code sets array as input to the **rpc_ns_mgmt_set_attribute()** routine, which registers their supported code sets in the name service database. Client applications look up a server's supported code sets in the name service database by calling the routine **rpc_ns_mgmt_read_codesets()** and then use their code sets array to evaluate their supported code sets against the code sets that the server supports.

The following DCE RPC routines require a code sets array and show an argument data type of **rpc_codeset_mgmt_t**:

- **rpc_ns_mgmt_read_codesets()**
- **rpc_rgy_get_codesets()**

Server applications that use **rpc_ns_mgmt_set_attribute()** to register their supported code sets in the name service database also specify the code sets array, but show an argument data type of **void**.

Conversion Type

The conversion type data structure is an enumerated type that RPC stub buffer sizing routines return to indicate whether character data conversion is necessary and whether or not existing storage is sufficient for the stub to store the results of the conversion. The conversion type can be one of the following values:

idl_cs_no_convert
: No code set conversion is required.

idl_cs_in_place_convert
: Code set conversion can be performed in a single storage area.

idl_cs_new_buffer_convert
: The converted data must be written to a new storage area.

The C language representation of a conversion type structure is as follows:

```
typedef enum {
        idl_cs_no_convert,
        idl_cs_in_place_convert,
        idl_cs_new_buffer_convert,
} idl_cs_convert_t;
```

Endpoint Map Inquiry Handle

An endpoint map inquiry handle is a pointer-size opaque variable containing information the RPC runtime uses to access the elements in a local or remote endpoint map. The description of the **rpc_ep_register()** routine lists the contents of an element.

The following routines require an endpoint map inquiry handle and show an argument data type of **rpc_ep_inq_handle_t**:

- **rpc_mgmt_ep_elt_inq_begin()**
- **rpc_mgmt_ep_elt_inq_done()**
- **rpc_mgmt_ep_elt_inq_next()**

Global Name

The NSI uses global names for the names of name service entries. A global name includes both a cell name and a cell-relative name composed of a directory pathname and a leaf name. For a description of global names, see the *DCE 1.2.2 Administration Guide—Introduction*. The cell name is assigned to a cell root at its creation. When you specify only a cell-relative name to an NSI operation, the NSI automatically expands the name into a global name by inserting the local cell name. Thus, the name of a member in a group or in a profile element is always stored as a global name. When returning the name of a name service entry or a member, NSI operations return global names.

For example, even when you specify a cell-relative name as the *member_name* parameter to routine **rpc_ns_group_mbr_add()**, when you read that group member (by calling **rpc_ns_group_mbr_inq_next()**), you will receive the corresponding global name.

IDL Encoding Service Handle

An IDL encoding service handle is a pointer-size opaque variable that points to functions that control how data encoding or decoding

is performed. The following routines return an IDL encoding service handle and show an argument data type of **idl_es_handle_t**:

- **idl_es_encode_incremental()**
- **idl_es_decode_buffer()**
- **idl_es_decode_incremental()**
- **idl_es_encode_dyn_buffer()**
- **idl_es_encode_fixed_buffer()**

The **idl_es_handle_free()** and **idl_es_inq_encoding_id()** routines require an IDL encoding service handle.

Note that in order to use the IDL encoding services, you must include a header file that has been generated for an application that has used the **encode** and **decode** ACF attributes on one or more of its operations.

Interface Handle and Specification

An interface handle is a pointer-size opaque variable containing information the RPC runtime uses to access the interface specification data structure.

The DCE IDL compiler automatically creates an interface specification data structure from each IDL file and creates a global variable of type **rpc_if_handle_t** for the interface specification.

The DCE IDL compiler places an interface handle declaration in the generated *interface-name***.h** file. The compiler generates this header file for each interface.

Routines requiring the interface handle as an argument show a data type of **rpc_if_handle_t**.

The form of each interface handle name is as follows:

- For the client:

  *if-name*_**v***major-version_minor-version*_**c_ifspec**

- For the server:

  *if-name*_**v***major-version_minor-version*_**s_ifspec**

where

- The *if-name* variable is the interface identifier specified in the IDL file.
- The *major-version* variable is the interface's major-version number specified in the IDL file.
- The *minor-version* variable is the interface's minor-version number specified in the IDL file.

An example is **notes_v1_2_c_ifspec**

The maximum combined length of the interface identifier and interface version number is 19 characters.

Since the major-version and minor-version numbers must each be at least 1 character, the interface name can be no more than 17 characters. This limits the interface handle name to 31 or fewer characters.

No concurrency control is required for interface handles.

The following routines require an interface handle and show an argument data type of **rpc_if_handle_t**:

- **rpc_ep_register()**
- **rpc_ep_register_no_replace()**
- **rpc_ep_resolve_binding()**
- **rpc_ep_unregister()**
- **rpc_if_inq_id()**
- **rpc_ns_binding_export()**
- **rpc_ns_binding_import_begin()**
- **rpc_ns_binding_lookup_begin()**
- **rpc_ns_binding_unexport()**
- **rpc_server_inq_if()**
- **rpc_server_register_if()**
- **rpc_server_unregister_if()**
- **rpc_server_use_all_protseqs_if()**
- **rpc_server_use_protseq_if()**

Interface Identifier

The interface identifier (id) data structure contains the interface UUID and major-version and minor-version numbers of an interface. The interface identifier is a subset of the data contained in the interface specification structure.

The C language representation of an interface identifier structure is as follows:

```
typedef struct {
        uuid_t      uuid;
        unsigned16  vers_major;
        unsigned16  vers_minor;
        } rpc_if_id_t;
```

Routines that require an interface identifier structure show a data type of **rpc_if_id_t**. In those routines, the application is responsible for providing memory for the structure.

The **rpc_if_inq_id()** routine returns the interface identifier from an interface specification. The following routines require an interface identifier:

- **rpc_mgmt_ep_elt_inq_begin()**
- **rpc_mgmt_ep_elt_inq_next()**
- **rpc_mgmt_ep_unregister()**
- **rpc_ns_mgmt_binding_unexport()**
- **rpc_ns_profile_elt_add()**
- **rpc_ns_profile_elt_inq_begin()**
- **rpc_ns_profile_elt_inq_next()**
- **rpc_ns_profile_elt_remove()**

Interface Identifier Vector

The interface identifier vector data structure contains a list of interfaces offered by a server. The interface identifier vector contains a count member (*count*), followed by an array of pointers to interface identifiers (**rpc_if_id_t**).

The C language representation of an interface identifier vector is as follows:

```
typedef struct {
        unsigned32      count;
        rpc_if_id_t     *if_id[1];
        } rpc_if_id_vector_t;
```

The interface identifier vector is a read-only vector. To obtain a vector of the interface identifiers registered by a server with the RPC runtime, an application calls the **rpc_mgmt_inq_if_ids()** routine. To obtain a vector of the interface identifiers exported by a server to a name service database, an application calls the **rpc_ns_mgmt_entry_inq_if_ids()** routine.

The RPC runtime allocates memory for the interface identifier vector. The application calls the **rpc_if_id_vector_free()** routine to free the interface identifier vector.

Manager Entry Point Vector

The manager entry point vector (EPV) is an array of pointers to remote procedures.

The DCE IDL compiler automatically generates a manager EPV data type, into the header file generated by the IDL compiler, for use in constructing manager EPVs. The data type is named as follows:

*if-name*_**v***major-version*_*minor-version*_**epv_t**

where

- The *if-name* variable is the interface identifier specified in the IDL file.
- The *major-version* variable is the interface's major-version number specified in the IDL file.
- The *minor-version* variable is the interface's minor-version number specified in the IDL file.

By default, the DCE IDL compiler automatically creates and initializes a manager EPV. DCE IDL creates this EPV assuming that a manager

routine of the same name exists for each procedure in the interface (as specified in the IDL file).

The DCE IDL compiler can define a client entry point vector with addresses of local routines. Client applications can call these routines. For more information about client entry point vectors, see the explanation of the **–cepv** argument in the **idl(1rpc)** reference page.

If the server offers multiple implementations of the same interface, the server must create additional manager EPVs, one for each implementation. Each EPV must contain exactly one entry point (address of a function) for each procedure defined in the IDL file. The server application declares and initializes one manager EPV variable of type *if-name*_**v***major-version_minor-version*_**epv_t** for each implementation of the interface.

The **rpc_server_register_if()** and **rpc_server_inq_if()** routines use the manager EPV data type and show the manager EPV argument as having an **rpc_mgr_epv_t** data type.

Name Service Handle

A name service handle is a pointer-size opaque variable containing information the RPC runtime uses to return the following RPC data from the name service database:

- Server binding handles
- UUIDs of resources offered by a server
- Profile members
- Group members

The following routines require a name service handle and show an argument data type of **rpc_ns_handle_t**:

- **rpc_ns_binding_import_begin()**
- **rpc_ns_binding_import_next()**
- **rpc_ns_binding_import_done()**
- **rpc_ns_binding_lookup_begin()**
- **rpc_ns_binding_lookup_next()**
- **rpc_ns_binding_lookup_done()**

- **rpc_ns_entry_object_inq_begin()**
- **rpc_ns_entry_object_inq_next()**
- **rpc_ns_entry_object_inq_done()**
- **rpc_ns_group_mbr_inq_begin()**
- **rpc_ns_group_mbr_inq_next()**
- **rpc_ns_group_mbr_inq_done()**
- **rpc_ns_profile_elt_inq_begin()**
- **rpc_ns_profile_elt_inq_next()**
- **rpc_ns_profile_elt_inq_done()**
- **rpc_ns_mgmt_handle_set_exp_age()**

The scope of a name service handle is from a ***_begin()** routine through the corresponding ***_done()** routine.

Applications have responsibility for concurrency control of name service handles across threads.

Protocol Sequence

A protocol sequence is a character string identifying the network protocols used to establish a relationship between a client and server. The protocol sequence contains a set of options that the RPC runtime must know about. The following options are in this set:

- The RPC protocol used for communications (choices are **ncacn** and **ncadg**).
- The format used in the network address supplied in the binding (choice is **ip**).
- The transport protocol used for communications (choices are **tcp** and **udp**).

Because only certain combinations of these options are valid (are useful for interoperation), RPC provides predefined strings that represent the valid combinations. RPC applications use only these strings.

The following table contains predefined strings representing valid protocol sequences. In the descriptions NCA is an abbreviation of Network Computing Architecture.

Valid Protocol Sequences	
Protocol Sequence	**Description**
ncacn_ip_tcp	NCA Connection over Internet Protocol: Transmission Control Protocol
ip or **ncadg_ip_udp**	NCA Datagram over Internet Protocol: User Datagram Protocol

A server application can use a particular protocol sequence only if the operating system software supports that protocol. A server chooses to accept remote procedure calls over some or all of the supported protocol sequences.

Client and server applications can determine if a protocol sequence is supported by both the RPC runtime and the operating system. The applications make this determination by calling the following routines:

- **rpc_network_inq_protseqs()**
- **rpc_network_is_protseq_valid()**

The following routines allow server applications to register protocol sequences with the runtime:

- **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()**

Those routines requiring a protocol sequence argument show a data type of **unsigned_char_t ***.

A client can use the protocol sequence strings to construct a string binding using the **rpc_string_binding_compose()** routine.

Protocol Sequence Vector

The protocol sequence vector data structure contains a list of protocol sequences over which the RPC runtime can send or receive remote procedure calls. The protocol sequence vector contains a count member

(*count*), followed by an array of pointers to protocol sequence strings (*protseq*).

The C language representation of a protocol sequence vector is as follows:

```
typedef struct {
        unsigned32      count;
        unsigned_char_t  *protseq[1];
        } rpc_protseq_vector_t;
```

The protocol sequence vector is a read-only vector. To obtain a protocol sequence vector, a server application calls the **rpc_network_inq_protseqs()** routine. The RPC runtime allocates memory for the protocol sequence vector. The server application calls the **rpc_protseq_vector_free()** routine to free the protocol sequence vector.

Statistics Vector

The statistics vector data structure contains statistics from the RPC runtime on a per address space basis. The statistics vector contains a count member (*count*), followed by an array of statistics. Each array element contains an **unsigned32** value. The following list describes the statistics indexed by the specified constant:

rpc_c_stats_calls_in
: The number of remote procedure calls received by the runtime.

rpc_c_stats_calls_out
: The number of remote procedure calls initiated by the runtime.

rpc_c_stats_pkts_in
: The number of network packets received by the runtime.

rpc_c_stats_pkts_out
: The number of network packets sent by the runtime.

The C language representation of a statistics vector is as follows:

```
typedef struct {
        unsigned32      count;
        unsigned32      stats[1];
        } rpc_stats_vector_t;
```

To obtain runtime statistics, an application calls the **rpc_mgmt_inq_stats()** routine. The RPC runtime allocates memory for the statistics vector. The application calls the **rpc_mgmt_stats_vector_free()** routine to free the statistics vector.

String Binding

A string binding contains the character representation of a binding handle.

String bindings are a convenient way of representing portions of a binding handle. However, you cannot use string bindings directly to make remote procedure calls. You must first call the routine **rpc_binding_from_string_binding()**, which converts a string binding to a binding handle.

A string binding does not contain all the information from a binding handle. For example, a call to **rpc_binding_to_string_binding()** does not translate the authentication information sometimes associated with a binding handle into the resulting string binding.

You can begin the development of a distributed application by having its servers communicate their binding information to clients by using string bindings. This communication allows a server to establish a client/server relationship without using the local endpoint map or the name service database.

In this case, the server calls none of the **rpc_ep_register()**, **rpc_ep_register_no_replace()**, and **rpc_ns_binding_export()** routines. Instead, the server calls only routine **rpc_server_inq_bindings()** to obtain a vector of binding handles. The server obtains binding handles one at a time from the vector and calls routine **rpc_binding_to_string_binding()** to convert each binding handle into a string binding. The resulting string binding is always fully bound and may contain a nonnil object UUID. The server then makes some or all of its string bindings available to clients. One way is placing the string bindings in a file to be read by clients or users or both. Another way

is delivering the string bindings to clients or users by means of a file, mail, or paper.

You can continue the distributed application's development by changing the application so that servers use the local endpoint map and the name service database to communicate their binding information.

To find the server, a client obtains a string binding containing a protocol sequence that the client runtime supports and, optionally, an object UUID that the client requires. The client then calls routine **rpc_binding_from_string_binding()** to convert the string binding into a server binding handle.

Other useful routines for working with string bindings are **rpc_string_binding_compose()**, which creates a string binding from its component parts, and **rpc_string_binding_parse()**, which separates a string binding into its component parts.

The two formats of a string binding follow. The four fields represent the object UUID, RPC protocol sequence, network address, and endpoint and network options of the binding. A delimiter character such as **@** (at sign) or **:** (colon) separates each field. A string binding does not contain any whitespace.

object-uuid **@** *rpc-prot-seq* **:** *nw-addr* **[***endpoint***,** *opt* **...]**

or

object-uuid **@** *rpc-prot-seq* **:** *nw-addr* **[***endpoint* **=** *endpoint***,** *opt* **...]**

object-uuid This field specifies the UUID of the object operated on by the remote procedure that is called with this string binding. The RPC runtime, at the server, maps the object's type to a manager entry point vector (EPV) to invoke the correct manager routine. The explanation of the routine **rpc_server_register_if()** discusses mapping object UUIDs to manager EPVs.

This field is optional. If you do not provide it the RPC runtime assumes a nil UUID.

@ This symbol is the delimiter character for the object UUID field. If you specify an object UUID you must follow it with this symbol.

rpc-protocol-sequence
This field specifies the protocol sequence used for making remote procedure calls. The valid protocol sequences are as follows:

ncacn_ip_tcp
ncacn_dnet_nsp
ncacn_osi_dna
ncadg_ip_udp
ncadg_dds

More information about these valid protocol sequences appears in the preceding table.

This field is required.

: This symbol is the delimiter character for the RPC protocol sequence field.

nw-addr This field specifies the address (*addr*) of a host on a network (*nw*) that receives remote procedure calls made with this string binding. The format and content of the network address depends on the value of *rpc-protocol-sequence* as follows:

ncacn_ip_tcp and **ncadg_ip_udp**

Specify an Internet address using the common Internet address notation or host name.

Two examples with common Internet address notation are **128.10.2.30** and **#126.15.1.28**. The second example shows the use of the optional **#** (number sign) character.

An example with a host name is **ko**.

If the specified host name is multihomed, the binding handle that is returned from the routine

rpc_binding_from_string_binding() contains a host address. It is the first host address returned from the system library call that translates a host name to a host address for the network address format in the protocol sequence. To control the host address used, specify the network address using the common Internet address notation instead of a host name.

The network address field is optional. If you do not supply this field, the string binding refers to your local host.

[This symbol is the delimiter character specifying that one endpoint and zero or more options follow. If the string binding contains at least one endpoint, this symbol is required.

endpoint This field specifies the endpoint, or address of a specific server instance on a host, to receive remote procedure calls made with this string binding. Optionally the keyword **endpoint=** can precede the endpoint specifier.

The format and content of the endpoint depends on the specified protocol sequence as follows:

ncacn_ip_tcp and **ncadg_ip_udp**

The endpoint field is optional. For more information about endpoints, see the information on binding handles in this reference page.

, This symbol is the delimiter character specifying that option data follows. If an option follows, this delimiter is required.

option This field specifies any options. Each option is specified as *option name=option value*.

The format and content of the option depends on the specified protocol sequence as follows:

ncacn_ip_tcp and **ncadg_ip_udp**

There are no Internet options.

The *option* field is optional.

] This symbol is the delimiter character specifying that one endpoint and zero or more options precede. If the string binding contains at least one endpoint, this symbol is required.

The \ (backslash) character is treated as an escape character for all string binding fields.

Examples of valid string bindings follow. In each example *obj-uuid* represents a UUID in string form. In other words, the symbol *obj-uuid* can represent the UUID 308fb580-1eb2-11ca-923b-08002b1075a7.

```
obj-uuid@ncacn_ip_tcp:16.20.16.27[2001]
obj-uuid@ncacn_ip_tcp:16.20.16.27[endpoint=2001]
```

String UUID

A string UUID contains the character representation of a UUID. A string UUID consists of multiple fields of hexadecimal characters. Each field has a fixed length, and dashes separate the fields. An example of a string UUID follows:

```
989c6e5c-2cc1-11ca-a044-08002b1bb4f5
```

When you supply a string UUID as an input argument to an RPC runtime routine, you can enter the alphabetic hexadecimal characters in either uppercase or lowercase letters. The RPC runtime routines that return a string UUID always return the hexadecimal characters in lowercase letters.

The following routines require a string UUID:

- **rpc_string_binding_compose()**
- **uuid_from_string()**

The following routines return a string UUID:

- **rpc_string_binding_parse()**
- **uuid_to_string()**

Unsigned Character String

DCE RPC treats all characters in strings as unsigned characters. Those routines with character string arguments show a data type of **unsigned_char_t ***.

UUID Vector

The UUID vector data structure contains a list of UUIDs. The UUID vector contains a count member (*count*), followed by an array of pointers to UUIDs.

The C language representation of a UUID vector is as follows:

```
typedef struct
{
    unsigned32     count;
    uuid_t         *uuid[1];
} uuid_vector_t;
```

An application constructs a UUID vector to contain object UUIDs to be exported or unexported from the name service database. The following routines require a UUID vector and show an argument data type of **uuid_vector_t**:

- **rpc_ep_register()**
- **rpc_ep_register_no_replace()**
- **rpc_ep_unregister()**
- **rpc_ns_binding_export()**
- **rpc_ns_binding_unexport()**
- **rpc_ns_mgmt_binding_unexport()**

Permissions Required

To use the NSI routines to access entries in a Cell Directory Service (CDS) database, you need access control list (ACL) permissions. Depending on the NSI operation, you need ACL permissions to the parent directory or the CDS object entry (the name service entry) or both.

The ACL permissions are as follows:

- To create an entry, you need insert permission to the parent directory.

- To read an entry, you need read permission to the CDS object entry.
- To write to an entry, you need write permission to the CDS object entry.
- To delete an entry, you need delete permission either to the CDS object entry or to the parent directory.
- To test an entry, you need either test permission or read permission to the CDS object entry.

Note that write permission does not imply read permission.

To find the ACL permissions for the NSI routines whose names begin with **rpc_ns**, see these routines' reference pages.

The non-NSI routines whose names do not begin with **rpc_ns** do not need ACL permissions, so their reference pages do not specify any.

Frequently Used Routine Parameters

A few parameters are common to many of the DCE RPC routines. These parameters are described fully here and again briefly on the specific routine reference pages.

binding Used as an input or output parameter.

Returns a binding handle for making remote procedure calls to a server.

A client obtains a binding handle by calling one of the following routines:

- **rpc_binding_copy()**
- **rpc_binding_from_string_binding()**
- **rpc_ns_binding_import_next()**
- **rpc_ns_binding_select()**

Creating a binding handle establishes a relationship between a client and a server. However, the relationship does not involve any communications between the client and server. The communications occur when a client makes a remote procedure call.

As an input parameter to a remote procedure call, *binding* specifies a binding handle that refers to binding information. The client's RPC runtime uses this binding information to make a remote procedure call to a server.

Server manager routines can extract client information from a client binding handle by using the following routines:

- **rpc_binding_inq_auth_client()**
- **rpc_binding_inq_object()**
- **rpc_binding_to_string_binding()**
- **rpc_string_binding_parse()**

name Used as an input/output parameter.

When used as an input parameter, the value of this parameter depends on the syntax selected in the *name_syntax* parameter. If it is allowed by the called routine, the value NULL specifies that the routine uses the name specified in the **RPC_DEFAULT_ENTRY** environment variable. Specifying NULL also has the called routine use the name syntax that the environment variable **RPC_DEFAULT_ENTRY_SYNTAX** specifies.

For a *name_syntax* value of **rpc_c_ns_syntax_dce**, use the DCE naming rules to specify parameter *name*.

As an output parameter, returns an entry in the name service database in the form of a character string that includes a terminating null character. The value of this parameter depends on the syntax selected in *name_syntax*.

For a *name_syntax* value of **rpc_c_ns_syntax_dce**, *name* is returned using the DCE naming syntax.

The DCE RPC runtime allocates memory for the returned string. The application is responsible for calling the **rpc_string_free()** routine to deallocate the string.

If an application does not want a returned name string, the application usually specifies NULL for this parameter. The one exception is routine **rpc_ns_entry_expand_name()**; it always returns a name string.

name_syntax Used as an input parameter, an integer value that specifies the syntax of an entry name. When allowed by the called routine, a value of **rpc_c_ns_syntax_default** specifies that the routine uses the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable. The following table lists the valid syntaxes that applications can use in DCE RPC for entries in the name service database.

Valid Name Syntaxes		
Constant	**Value**	**Description**
rpc_c_ns_syntax_default	0	Default syntax
rpc_c_ns_syntax_dce	3	DCE

The *name_syntax* parameter tells routines how to parse the entry name specified in an input *name* parameter or specifies the syntax to use when returning an entry name as an output *name* parameter.

If the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable is not defined, the RPC runtime uses the **rpc_c_ns_syntax_dce** name syntax.

string Used as an input or output parameter.

Returns a character string, which always includes the terminating null character **\0**. The DCE RPC runtime allocates memory for the returned string. The application calls the **rpc_string_free()** routine to deallocate the memory occupied by the string.

If there is no data for the requested string, the routine returns the string **\0**. For example, if the string binding passed to routine **rpc_string_binding_parse()** does not contain an object UUID, the routine returns **\0** as the value of the object UUID string. The application must call the **rpc_string_free()** routine to deallocate the memory occupied by this string.

If an application does not require a returned output string, the application specifies NULL for this parameter.

status Each routine in the RPC API returns a DCE status code indicating whether the routine completed successfully or, if not, why not. A return value of **rpc_s_ok** indicates success. All other return values signify routine failure. The status codes listed for each RPC runtime routine are the most likely, but not necessarily all, the status codes that the routine can return.

The status code argument has a data type of **unsigned32**.

To translate a DCE status code to a text message, call the routine **dce_error_inq_text()**.

Note that RPC exceptions are equivalent to RPC status codes. To identify the status code that corresponds to a given exception, replace

the **_x_** string of the exception with the string **_s_**; for example, the exception **rpc_x_already_listening** is equivalent to the status code **rpc_s_already_listening**.

For more information about the RPC status codes, see the *DCE 1.2.2 Problem Determination Guide*.

uuid Used as an input or output parameter.

When you need to specify a nil UUID to a *uuid* input parameter in any of the DCE RPC routines, you can supply the value NULL.

Related Information

Books: *DCE 1.2.2 Application Development—Introduction and Style Guide*, *DCE 1.2.2 Application Development Guide—Core Components*, *DCE 1.2.2 Application Development Guide—Directory Services*, *DCE 1.2.2 Command Reference*, *DCE 1.2.2 Problem Determination Guide*.

cs_byte_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 cs_byte_from_netcs(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        idl_byte *network_data,
        unsigned32 network_data_length,
        unsigned32 local_buffer_size,
        idl_byte *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 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.

local_buffer_size
A pointer to the buffer size to be allocated to contain the converted data, in units of **cs_byte**. The value specified in this parameter is the local buffer size returned from the **cs_byte_local_size()** routine.

Output

local_data A pointer to the converted data, in **cs_byte** format.

local_data_length
The length of the converted data, in units of **cs_byte**. 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 **cs_byte_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 **cs_byte_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 **cs_byte_*_netcs()** routines when the **cs_byte** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. (The **cs_byte** type is equivalent to the **idl_byte** type.)

Client and server stubs call the **cs_byte_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 **cs_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 **cs_byte**.

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 running the host converter, the following errors can occur:

- **rpc_s_ss_invalid_char_input**

- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, the routine calls the **dce_cs_loc_to_rgy()** routine and the host converter routines. If these routines return an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_to_netcs(3rpc)**.

cs_byte_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 cs_byte_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. The conversion type can be one of the following values:

idl_cs_no_convert
No code set conversion is required.

idl_cs_in_place_convert
Code set conversion can be performed in the current buffer.

idl_cs_new_buffer_convert
The converted data must be written to a new buffer.

local_buffer_size
A pointer to the buffer size that needs to be allocated to contain the converted data, in units of **cs_byte**. 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 **cs_byte_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 **cs_byte_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 **cs_byte_*_size** routines when the **cs_byte** type (which is internally equivalent to **idl_byte**) has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **cs_byte_local_size()** routine is used to evaluate buffer size requirements prior to unmarshalling data received over the network.

Applications do not call **cs_byte_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**, if a conformant or conformant varying array is to be unmarshalled, or they specify NULL if a fixed or varying array is to be marshalled.

When called from a client stub, the **cs_byte_local_size()** routine determines the value of *conversion_type* from the client and server's code set tag information stored in the binding handle by a code sets evaluation routine or a tag-setting routine. If the conversion type specified in the handle is **idl_cs_new_buffer_convert**, the routine sets the *conversion_type* parameter to this value and, if a conformant or conformant varying array is to be unmarshalled, calculates a new buffer size by multiplying the value of *network_buffer_size* by the maximum number of bytes required to represent the code set specified in *network_code_set_value*. The routine returns the new buffer size in the *local_buffer_size* parameter. The size is specified in units of **cs_byte**, which is the local representation used for international character data (and is equivalent to the **idl_byte** data type). For fixed and varying arrays, the routine assumes that *network_buffer_size* is sufficient to store the converted data.

If the handle information specifies **idl_cs_convert_in_place** or **idl_cs_no_convert**, the routine assumes that *network_buffer_size* can store the converted data (or that no conversion is necessary) and returns **idl_cs_convert_in_place** (or **idl_cs_no_convert**) in the *conversion_type* parameter. If a conformant or conformant varying array is to be unmarshalled. the routine also returns the value of *network_buffer_size* in *local_buffer_size*

In cases in which the binding handle does not contain the results of character and code sets evaluation, or in which the **cs_byte_local_size()** routine is being called from the server stub, it determines the value of *conversion_type* itself using the local code set value and the code set value passed in the *network_code_set_value* parameter and returns the appropriate *conversion_type* value. If a conformant or conformant varying array is to be unmarshalled, and the routine finds that a new buffer is required to hold the converted data, it also calculates the size of this new buffer (by multiplying the value of *network_buffer_size* by the maximum number of bytes required to represent the code set specified in *network_code_set_value*) and returns the results, in units of **cs_byte**, in *local_buffer_size*.

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 **cs_byte_local_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_net_size(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**. **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**.

cs_byte_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 cs_byte_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 **cs_byte**, 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 existing storage is sufficient for storing the results of the conversion. The conversion type can be one of the following values:

idl_cs_no_convert
No code set conversion is required.

idl_cs_in_place_convert
Code set conversion can be performed in the current buffer.

idl_cs_new_buffer_convert
The converted data must be written to a new buffer.

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 **cs_byte_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 **cs_byte_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 marshalled). 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 **cs_byte_*_size** routines when the **cs_byte** type (which is internally equivalent to **idl_byte**) has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **cs_byte_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 **cs_byte_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 **cs_byte**.

When called from a client stub, the **cs_byte_net_size()** routine determines the value of *conversion_type* from the client and server's code set tag information set up the binding handle by a code sets evaluation routine or a tag-setting routine. If the conversion type specified in the handle is **idl_cs_new_buffer_convert**, the routine sets the *conversion_type* parameter to this value and, if a conformant or conformant varying array is to be marshalled, calculates a new buffer size by multiplying the value of *local_buffer_size* by the maximum number of bytes required to represent the code set specified in *network_code_set_value* (the sending tag parameter).

The routine returns the new buffer size in the *network_buffer_size* parameter. The size is specified in units of **idl_byte**, which is the network representation used for international character data (and is internally equivalent to the **cs_byte** type). For fixed and varying arrays, the routine assumes that *local_buffer_size* is sufficient to store the converted data.

If the binding handle information specifies **idl_cs_convert_in_place** or **idl_cs_no_convert**, the routine assumes that *local_buffer_size* can store the converted data (or that no conversion is necessary) and returns **idl_cs_convert_in_place** (or **idl_cs_no_convert**) in the *conversion_type* parameter. If a conformant or conformant varying array is to be marshalled, the routine also returns the value of *local_buffer_size* in *network_buffer_size*.

In cases in which the binding handle does not contain the results of character and code sets evaluation, or in which the **cs_byte_net_size()** routine is being called from the server stub, it determines the value of *conversion_type* itself using the local code set value and the code set value passed in the *network_code_set_value* parameter and returns the appropriate *conversion_type* value. If a conformant or conformant varying array is to be marshalled, and the routine finds that a new buffer is required to hold the converted data, it also calculates the size of this new buffer (by multiplying the value of *local_buffer_size* by the maximum number of bytes required to represent the code set specified in *network_code_set_value*) and returns the results, in units of **idl_byte**, in *network_buffer_size*.

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 **cs_byte_net_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_local_size(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dcs_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**.

cs_byte_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 cs_byte_to_netcs(
        rpc_binding_handle_t binding,
        unsigned32 network_code_set_value,
        idl_byte *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 **cs_byte** 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 **cs_byte_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 **cs_byte_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 **cs_byte_*_netcs()** routines when the **cs_byte** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. (The **cs_byte** type is equivalent to the **idl_byte** type.)

Client and server stubs call the **cs_byte_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 to be converted, in units of **idl_byte**.

The routine compares the code set specified as the network 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**.

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 running the host converter, the following errors can occur:

- **rpc_s_ss_invalid_char_input**
- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, the routine calls the **dce_cs_loc_to_rgy()** routine and the host converter routines. If these routines return an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_to_netcs(3rpc)**.

dce_cs_loc_to_rgy

Purpose

Maps a local name for a code set to a code set value in the code set registry; used by client and server applications

Synopsis

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

void dce_cs_loc_to_rgy(
        idl_char *local_code_set_name,
        unsigned32 *rgy_code_set_value,
        unsigned16 *rgy_char_sets_number,
        unsigned16 **rgy_char_sets_value,
        error_status_t *status);
```

Parameters

Input

local_code_set_name
: A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 character data bytes plus a terminating NULL character.

Output

rgy_code_set_value
: The registered integer value that uniquely identifies the code set specified by *local_code_set_name*.

rgy_char_sets_number
: The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.

rgy_char_sets_value
: A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying

NULL prevents this routine from returning this parameter. The routine dynamically allocates 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 **dce_cs_loc_to_rgy()** routine is a DCE function that maps operating system-specific names for character/code set encodings to their unique identifiers in the code set registry.

The routine is currently used by the DCE RPC routines for character and code set interoperability, which permit DCE RPC clients and servers to transfer international character data in a heterogeneous character set and code sets environment.

The **dce_cs_loc_to_rgy()** routine takes as input a string that holds the host-specific local name of a code set and returns the corresponding integer value that uniquely identifies that code set, as registered in the host's code set registry. If the integer value does not exist in the registry, the routine returns the status **dce_cs_c_unknown**. The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the *rgy_char_sets_number* and *rgy_char_sets_value*[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a code set value from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the array after it is used, since the array is dynamically allocated.

The DCE RPC code sets compatibility evaluation routines **rpc_cs_eval_with_universal()**, **rpc_cs_eval_without_universal()**, and **rpc_cs_character_set_compat_check()** use this routine to obtain registered integer values for a client and server's supported character sets in order to ensure that the server supports a character set that is compatible with the client. The DCE RPC stub support routines for code set conversion can use this routine to obtain the registered code set value that corresponds to the code set they are currently using; that is, the local code set specified in their host's locale environment. The stub routines for code set conversion then compare their local code set value to the code set value specified in the sending tag for the call to determine whether code set conversion is necessary.

In general, programmers who are developing RPC applications that transfer international characters do not need to call this routine directly; the DCE RPC routines provided for code sets evaluation and the DCE RPC stub support routines for code set conversion call this routine on the client or server application's behalf.

However, programmers who are developing their own stub support routines for code set conversion may want to include this routine in their conversion code to map their current locale information to a code set value in order to perform local-to-sending tag code set comparison.

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_ok

dce_cs_c_cannot_allocate_memory

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

dce_cs_c_unknown

dce_cs_c_not_found

Related Information

Commands: **csrc(8dce)**.

Functions: **dce_cs_rgy_to_loc(3rpc)**, **rpc_cs_char_set_compat_check(3rpc)**, **rpc_cs_eval_with_universal(3rpc)**, **rpc_cs_eval_without_universal(3rpc)**, **rpc_rgy_get_code_sets(3rpc)**.

dce_cs_rgy_to_loc

Purpose

Maps a code set value in the code set registry to the local name for a code set; used by client and server applications

Synopsis

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

void dce_cs_rgy_to_loc(
        unsigned32 *rgy_code_set_value,
        idl_char **local_code_set_name,
        unsigned16 *rgy_char_sets_number,
        unsigned16 **rgy_char_sets_value,
        error_status_t *status);
```

Parameters

Input

rgy_code_set_value
: The registered hexadecimal value that uniquely identifies the code set.

Output

local_code_set_name
: A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 character data bytes and a terminating NULL character.

rgy_char_sets_number
: The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.

rgy_char_sets_value
: A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL

in this parameter prevents the routine from returning this value. The routine dynamically allocates 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 **dce_cs_rgy_to_loc()** routine is a DCE function that maps a unique identifier for a code set in the code set registry to the operating system-specific name for the code set, if it exists in the code set registry.

The routine is currently used by the DCE RPC routines for character and code set interoperability, which permit DCE applications to transfer international characters in a heterogeneous character and code sets environment.

The **dce_cs_rgy_to_loc()** routine takes as input a registered integer value of a code set and returns a string that holds the operating system-specific, or local name, of the code set.

If the local code set name does not exist in the registry, the routine returns the status **dce_cs_c_unknown** and returns an undefined string.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the *rgy_char_sets_number* and *rgy_char_sets_value*[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a local code set name from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the *rgy_char_sets_value* array after it is used.

In general, client and server applications that use the DCE RPC character and code set interoperablity features do not need to call this routine directly; the DCE RPC stub support routines provided for code set conversion call this routine on the client or server application's behalf to obtain the string name that matches the name of the host code set converter that they will call to perform the actual code set conversion.

However, application programmers who are developing their own RPC stub support routines for code set conversion may want to include this routine in their conversion code to map code set values sent in code set tags into the local names for the code sets so that they can locate the correct operating system code set converter.

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_ok

dce_cs_c_cannot_allocate_memory

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

dce_cs_c_unknown

dce_cs_c_not_found

Related Information

Commands: **csrc(8dce)**.

Functions: **dce_cs_loc_to_rgy(3rpc)**, **rpc_cs_char_set_compat_check(3rpc)**, **rpc_cs_eval_with_universal(3rpc)**, **rpc_cs_eval_without_universal(3rpc)**, **rpc_rgy_get_code_sets(3rpc)**.

idl_es_decode_buffer

Purpose

Returns a buffer decoding handle to the IDL encoding services

Synopsis

```
void idl_es_decode_buffer(
        idl_byte *encoded_data_buffer,
        idl_ulong_int buffer_size,
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input

encoded_data_buffer
: The address of the buffer that contains the data to be decoded.

buffer_size
: The number of bytes of data in the buffer to be decoded.

Output

es_handle
: Returns the address of an IDL encoding services handle for use by a client or server decoding operation.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network. Client and server applications can use the IDL encoding services to create persistent storage for their

data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_decode_buffer()** routine belongs to a set of routines that return handles to the IDL encoding services for use by client and server encoding and decoding operations. The information in the handle controls the way in which the IDL encoding services manage memory when encoding or decoding data.

The **idl_es_decode_buffer()** routine returns a buffer decoding handle, which directs the IDL encoding services to decode data from a single application-supplied buffer of encoded data.

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_ss_bad_buffer
Bad buffer operation.

rpc_s_no_memory
Insufficient memory available to complete operation.

Related Information

Function: **idl_es_decode_incremental(3rpc)**.

idl_es_decode_incremental

Purpose

Returns an incremental decoding handle to the IDL encoding services; used by client and server applications

Synopsis

```
void idl_es_decode_incremental(
        idl_void_p_t state,
        idl_es_read_fn_t read_fn,
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input/Output

state Specifies the address of an application-provided data structure that coordinates the actions of successive calls to the *read_fn* routine. The *state* data structure acts as a communications channel between the application and the *read_fn* routine.

Input

read_fn Specifies the address of a user-provided routine that generates a buffer of encoded data for decoding by the IDL encoding services. The IDL encoding services call the *read_fn* routine repeatedly until all of the data has been decoded.

The following C definition for **idl_es_read_fn_t** illustrates the prototype for the *read_fn* routine:

```
typedef void (*idl_es_read_fn_t)
(
  idl_void_p_t     state,          /* in/out */
  idl_byte         **buffer,       /* in */
  idl_ulong_int    *size,          /* in */
);
```

The **idl_es_decode_incremental()** routine passes the specified *state* parameter value as input to the *read_fn* routine. The *state* data structure is the communications path between the application and the *read_fn* routine. For example, the application can use the *state* parameter to pass in an open file pointer from which the *read_fn* routine is to read encoded data.

The *buffer* parameter specifies the address of the data to be decoded; this address must be 8-byte aligned. The *size* parameter specifies the size of the buffer to be decoded, and must be a multiple of 8 bytes unless it represents the size of the last buffer to be decoded.

The *read_fn* routine should return an exception on error.

Output

es_handle Returns the address of an IDL encoding services handle for use by a client or server decoding operation.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network. Client and server applications can use the IDL encoding services to create persistent storage for their data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_decode_incremental()** routine belongs to a set of routines that return handles to the IDL encoding services for use by client and server encoding and

decoding operations. The information in the handle controls the way in which the IDL encoding services manage memory when encoding or decoding data.

The **idl_es_decode_incremental()** routine returns an incremental decoding handle, which directs the IDL encoding services to decode data by calling the user-supplied *read_fn* routine, which generates a small buffer of encoded data for the IDL encoding services to decode. The routine passes the buffer address and size to the IDL encoding services, which then decode the buffer. The IDL encoding services call the *read_fn* routine repeatedly until there is no more data to decode.

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_no_memory
Insufficient memory available to complete operation.

Related Information

Functions: **idl_es_decode_buffer(3rpc)**, **idl_es_encode_incremental(3rpc)**.

idl_es_encode_dyn_buffer

Purpose

Returns a dynamic buffer encoding handle to the IDL encoding services; used by client and server applications

Synopsis

```
void idl_es_encode_dyn_buffer(
        idl_byte **encoded_data_buffer,
        idl_ulong_int *buffer_size,
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input

None.

Output

encoded_data_buffer
: The address to which the IDL encoding services will write the address of the buffer that contains the encoded data, when the encoding process is complete. When the application no longer needs the buffer, it should release the memory resource. See the *DCE 1.2.2 Application Development Guide—Core Components* for an explanation of how to manage memory when using the IDL encoding services.

buffer_size
: The address to which the IDL encoding services will write the size of the buffer that contains the encoded data, when the encoding process is complete.

es_handle
: Returns the address of an IDL encoding services handle for use by a client or server encoding operation.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network. Client and server applications can use the IDL encoding services to create persistent storage for their data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_encode_dyn_buffer()** routine belongs to a set of routines that return handles to the IDL encoding services for use by client and server encoding and decoding operations. The information in the handle controls the way in which the IDL encoding services manage memory when encoding or decoding data.

The **idl_es_encode_dyn_buffer()** routine returns a dynamic buffer encoding handle, which directs the IDL encoding services to store the encoded data in a chain of small buffers, build an additional single buffer that contains the encoded data, and pass that buffer's address to the application. Dynamic buffering is the most expensive style of IDL encoding services buffering, since two copies of the encoded data exist (one in the chain of buffers, and one in the single buffer).

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_ss_bad_buffer
Bad buffer operation.

rpc_s_no_memory
Insufficient memory available to complete operation.

Related Information

Functions: **idl_es_encode_fixed_buffer(3rpc)**, **idl_es_encode_incremental(3rpc)**.

idl_es_encode_fixed_buffer

Purpose

Returns a fixed buffer encoding handle to the IDL encoding services

Synopsis

```
void idl_es_encode_fixed_buffer(
        idl_byte *data_buffer,
        idl_ulong_int data_buffer_size,
        idl_ulong_int *encoded_buffer_size,
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input

data_buffer The address of the application-supplied buffer. This address must be 8-byte aligned.

data_buffer_size
The size of the application-supplied buffer. The size must be a multiple of 8 bytes.

Output

encoded_buffer_size
Returns the address to which the IDL encoding services write the size of the encoded buffer when they have completed encoding the data.

es_handle Returns the address of an IDL encoding services handle for use by a client or server encoding operation.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network.

Client and server applications can use the IDL encoding services to create persistent storage for their data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_encode_fixed_buffer()** routine belongs to a set of routines that return handles to the IDL encoding services for use by client and server encoding and decoding operations. The information in the handle controls the way in which the IDL encoding services manage memory when encoding or decoding data.

The **idl_es_encode_fixed_buffer()** routine returns a fixed buffer encoding handle, which directs the IDL encoding services to encode data into a single buffer that the application has provided. The fixed buffer encoding style is useful for applications that need only one buffer for their encoding and decoding process. The buffer that the application allocates must be large enough to hold all of the encoded data, and must also allocate 56 bytes for each encoding operation that the application has defined (this space is used to hold per-operation header information.)

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_bad_buffer
Bad buffer operation.

rpc_s_no_memory
Insufficient memory available to complete operation.

Related Information

Functions: **idl_es_encode_dyn_buffer(3rpc)**, **idl_es_encode_incremental(3rpc)**.

idl_es_encode_incremental

Purpose

Returns an incremental encoding handle to the IDL encoding services; used by client and server applications

Synopsis

```
void idl_es_encode_incremental(
        idl_void_p_t state,
        idl_es_allocate_fn_t allocate_fn,
        idl_es_write_fn_t write_fn,
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input/Output

state Specifies the address of an application-provided data structure that coordinates the actions of the *allocate_fn* and *write_fn* routines. The *state* data structure acts as a communications channel between the application and the *allocate_fn* and *write_fn* routines.

Input

allocate_fn Specifies the address of a user-provided routine that allocates an empty buffer. The encoding stub uses the allocated buffer to store encoded data.

The following C definition for **idl_es_allocate_fn_t** illustrates the prototype for the buffer allocation routine:

```
typedef void (*idl_es_allocate_fn_t)
 (
  idl_void_p_t    state,          /* in/out */
  idl_byte        **buffer,       /* out */
  idl_ulong_int   *size,          /* in/out */
 );
```

The **idl_es_encode_incremental()** routine passes the specified *state* parameter value as input to the *allocate_fn* buffer allocation routine. When the IDL encoding services call the *allocate_fn* routine, the value at the address indicated by *size* represents the buffer size that the IDL encoding services have requested the routine to allocate. When the *allocate_fn* buffer allocation routine allocates the buffer, it writes the actual size of the allocated buffer to this parameter; the value must be a multiple of eight bytes. The *buffer* parameter specifies the address of the allocated buffer; this address must be 8-byte aligned.

The *allocate_fn* routine should return an exception on error.

write_fn Specifies the address of a user-provided routine that writes the contents of a buffer that contains data that has been encoded by the IDL encoding services. The IDL encoding services will call this routine when the buffer allocated by *allocate_fn* is full, or when all of the application's encoding operation parameters have been encoded.

The following C definition for **idl_es_write_fn_t** illustrates the prototype for the *write_fn* routine:

```
typedef void (*idl_es_write_fn_t)
 (
  idl_void_p_t    state,          /* in/out */
  idl_byte        *buffer,        /* in */
  idl_ulong_int   size,           /* in */
 );
```

The **idl_es_encode_incremental()** routine passes the specified *state* parameter value as input to the *write_fn* routine. The *buffer* parameter value is the address of the data that the IDL encoding services have encoded. The *size* parameter value is the size, in bytes, of the encoded data.

The *write_fn* routine should return an exception on error.

Output

es_handle Returns the address of an IDL encoding services handle for use by a client or server encoding operation.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network. Client and server applications can use the IDL encoding services to create persistent storage for their data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_encode_incremental()** routine belongs to a set of routines that return handles to the IDL encoding services for use by client and server encoding and decoding operations. The information in the handle controls the way in which the IDL encoding services manage memory when encoding or decoding data.

The **idl_es_encode_incremental()** routine returns an incremental encoding handle, which directs the IDL encoding services to encode data into a chain of small buffers that the user-provided *allocate_fn* routine manages. The user-provided *write_fn* routine writes the encoded data in these buffers back for access by the application.

The *state* data structure is the communications path between the application and the *allocate_fn* and *write_fn* routines. For example, the application can build a cache of pre-allocated memory to store encoded data, and store pointers to that pre-allocated memory in the *state* data structure. When invoked by the IDL encoding services to allocate a buffer, the *allocate_fn* routine can search the *state* data structure for a free memory location to use.

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_no_memory
Insufficient memory available to complete operation.

Related Information

Functions: **idl_es_decode_incremental(3rpc)**, **idl_es_encode_dyn_buffer(3rpc)**, **idl_es_encode_fixed_buffer(3rpc)**.

idl_es_handle_free

Purpose

Frees an IDL encoding services handle

Synopsis

```
void idl_es_handle_free(
        idl_es_handle_t *es_handle,
        error_status_t *status);
```

Parameters

Input/Output

es_handle The address of the handle whose resources are to be freed. The handle is made NULL by this operation.

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 **idl_es_handle_free** routine frees an IDL encoding services handle that has been allocated by one of the IDL encoding services handle-returning routines.

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: **idl_es_decode_buffer(3rpc)**, **idl_es_decode_incremental(3rpc)**, **idl_es_encode_dyn_buffer(3rpc)**, **idl_es_encode_fixed_buffer(3rpc)**, **idl_es_encode_incremental(3rpc)**.

idl_es_inq_encoding_id

Purpose

Identifies an operation within an interface that has been called to encode data using the IDL encoding services

Synopsis

```
void idl_es_inq_encoding_id(
        idl_es_handle_t es_handle,
        rpc_if_id_t *if_id,
        idl_ulong_int *op_num,
        error_status_t *status);
```

Parameters

Input

es_handle A encoding services handle returned by one of the IDL encoding services handle-returning routines.

Output

if_id Returns the interface UUID and version number assigned to the interface that defines the operation that encoded the data. This information is stored in the IDL encoding services handle that is associated with the encoded data.

op_num Returns the operation number assigned to the operation that encoded the data. Operations are numbered in the order in which they appear in the interface definition, starting with zero (0). The operation number for the operation that encoded the data is stored in the IDL encoding services handle that is associated with the encoded data.

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

Description

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into a byte stream and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally, and is not transmitted over the network. Client and server applications can use the IDL encoding services to create persistent storage for their data. Encoding flattens complex data types into a byte stream for storage on disk, while decoding restores the flattened data to complex form.

The **idl_es_inq_encoding_id()** routine returns the identity of an operation within an application that has been invoked to encode data using the IDL encoding services. Applications can use this routine to determine the identity of an encoding operation, for example, before calling their decoding operations.

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_unknown_if
Interface identifier and operation number unavailable.

Related Information

Functions: **idl_es_decode_buffer(3rpc)**, **idl_es_decode_incremental(3rpc)**, **idl_es_encode_dyn_buffer(3rpc)**, **idl_es_encode_fixed_buffer(3rpc)**, **idl_es_encode_incremental(3rpc)**.

rpc_binding_copy

Purpose

Returns a copy of a binding handle; used by client or server applications

Synopsis

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

void rpc_binding_copy(
        rpc_binding_handle_t source_binding,
        rpc_binding_handle_t *destination_binding,
        unsigned32 *status);
```

Parameters

Input

source_binding
: Specifies the server binding handle whose referenced binding information is copied.

Output

destination_binding
: Returns the server binding handle that refers to the copied binding 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 **rpc_binding_copy()** routine copies the server binding information referenced by the binding handle specified in the *source_binding* parameter. This routine returns a new server binding handle for the copied binding information. The new server binding handle is returned in the *destination_binding* parameter.

Use the **rpc_binding_copy()** routine if you want a change (made to binding information by one thread) *not* to affect the binding information used by other threads. The explanation of binding handles in the **rpc_intro(3rpc)** reference page has more detail about this use of routine **rpc_binding_copy()**.

After calling this routine, operations performed on the *source_binding* binding handle do not affect the binding information referenced by the *destination_binding* binding handle. Similarly, operations performed on the *destination_binding* binding handle do not affect the binding information referenced by the *source_binding* binding handle.

If you want the changes made to binding information by one thread to affect the binding information used by other threads, your program must share a single binding handle across the threads. In this case the application controls binding handle concurrency.

When an application is finished using the binding handle specified by the *destination_binding* parameter, the application calls the **rpc_binding_free()** routine to release the memory used by the *destination_binding* binding handle and its referenced binding information.

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_binding_free(3rpc)**.

rpc_binding_free

Purpose

Releases binding handle resources; used by client or server applications

Synopsis

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

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

Parameters

Input/Output

binding Specifies the server binding handle to free.

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_binding_free()** routine frees the memory used by a server binding handle and its referenced binding information. Use this routine when your application is finished using a server binding handle that was dynamically created during program execution.

If the free-binding operation succeeds, the *binding* parameter returns the value NULL.

An application can dynamically create binding handles by calling any of the following routines:

- **rpc_binding_copy()**
- **rpc_binding_from_string_binding()**
- **rpc_ns_binding_import_next()**

- **rpc_ns_binding_select()**
- **rpc_server_inq_bindings()**

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_binding_copy(3rpc)**, **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_vector_free(3rpc)**, **rpc_ns_binding_import_next(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_ns_binding_select(3rpc)**, **rpc_server_inq_bindings(3rpc)**.

rpc_binding_from_string_binding

Purpose

Returns a binding handle from a string representation; used by client or management applications

Synopsis

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

void rpc_binding_from_string_binding(
        unsigned_char_t *string_binding,
        rpc_binding_handle_t *binding,
        unsigned32 *status);
```

Parameters

Input

string_binding
: Specifies a string representation of 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_binding_from_string_binding()** routine creates a server binding handle from a string representation of a binding handle.

The *string_binding* parameter does not need to contain an object UUID. In this case, the returned *binding* contains a nil UUID.

If the provided *string_binding* parameter does not contain an endpoint field, the returned *binding* parameter is a partially bound server binding handle.

If the provided *string_binding* parameter does contain an endpoint field, the returned *binding* parameter is a fully bound server binding handle with a well-known endpoint.

If the provided *string_binding* parameter does not contain a host address field, the returned *binding* parameter refers to the local host.

To create a string binding, call the **rpc_string_binding_compose()** routine or call the **rpc_binding_to_string_binding()** routine or provide a character string constant.

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 partially and fully bound 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_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_invalid_string_binding
Invalid string binding.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

uuid_s_bad_version
Bad UUID version.

uuid_s_invalid_string_uuid
Invalid format for a string UUID.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_string_binding_compose(3rpc)**.

rpc_binding_inq_auth_caller

Purpose

Returns authentication and authorization information from the binding handle for an authenticated client; used by server applications

Synopsis

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

void rpc_binding_inq_auth_caller(
        rpc_binding_handle_t binding_handle,
        rpc_authz_cred_handle_t *privs,
        unsigned_char_p_t *server_princ_name,
        unsigned32 *protect_level,
        unsigned32 *authn_svc,
        unsigned32 *authz_svc,
        unsigned32 *status);
```

Parameters

Input

binding_handle

Specifies the client binding handle from which to return the authentication and authorization information.

Output

privs

Returns an opaque handle to the authorization information for the client that made the remote procedure call on *binding_handle*.

The data referenced by this parameter are read-only and should not be modified by the server. If the server wants to preserve any of the returned data, it must copy the data into server-allocated memory.

server_princ_name

Returns a pointer to the server principal name specified by the client that made the remote procedure call on *binding_handle*. The content of the returned name and its syntax are defined by the authentication service in use.

Specifying NULL prevents the routine from returning this parameter. In this case, the caller does not have to call the **rpc_string_free()** routine.

protect_level Returns the protection level requested by the client that made the remote procedure call on *binding*. The protection level determines the degree to which authenticated communications between the client and the server are protected.

Specifying NULL prevents the routine from returning this parameter.

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 encrypt each remote procedure call argument value.

authn_svc Returns the authentication service requested by the client that made the remote procedure call on *binding*.

Specifying NULL prevents the routine from returning this parameter.

The possible 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.

authz_svc Returns the authorization service requested by the client that made the remote procedure call on *binding_handle*.

Specifying NULL prevents the routine from returning this parameter.

The possible authorization services are as follows:

rpc_c_authz_none
Server performs no authorization. This is valid only if the *authn_svc* parameter is **rpc_c_authn_none**.

rpc_c_authz_name
Server performs authorization based on the client principal name.

rpc_c_authz_dce
Server performs authorization by using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with *binding_handle*. Generally, access is checked against DCE access control lists (ACLs).

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 The routine completed successfully.

rpc_s_invalid_binding
The routine did not complete because of an invalid binding handle.

rpc_s_wrong_kind_of_binding
The routine did not complete because of the wrong kind of binding was specified for the operation.

rpc_s_binding_has_no_auth
The routine completed successfully, but the binding has no authentication information.

Description

The **rpc_binding_inq_auth_caller()** routine returns authentication and authorization information associated with the client identified by *binding_handle*. The calling server manager routine can use the returned data for authorization purposes.

If the client is part of a delegation chain, the call returns the authentication and authorization information for each member of the chain, the initiator and all subsequent delegates. You can use the **sec_cred_get_initiator()** or **sec_cred_get_delegate()** calls to obtain the authorization information for a specific member of the chain.

The RPC runtime allocates memory for the returned *server_princ_name* parameter. The server is responsible for calling the **rpc_string_free()** routine for the returned parameter string.

For applications in which the client side uses the Interface Definition Language (IDL) **auto_handle** or **implicit_handle** attributes, the server side needs to be built with the IDL **explicit_handle** attribute specified in the attribute configuration file (ACF). Using **explicit_handle** provides *binding_handle* as the first parameter to each server manager routine.

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_invalid_binding

rpc_s_wrong_kind_of_binding

rpc_s_binding_has_no_auth

sec_login_s_default_use

sec_login_s_context_invalid

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_info(3rpc)**, **rpc_binding_set_auth_info(3rpc)**, **rpc_string_free(3rpc)**, **sec_cred_get_initiator(3sec)**, **sec_cred_get_delegate(3sec)**.

rpc_binding_inq_auth_client

Purpose

Returns authentication and authorization information from the binding handle for an authenticated client; used by server applications

Synopsis

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

void rpc_binding_inq_auth_client(
        rpc_binding_handle_t binding,
        rpc_authz_handle_t *privs,
        unsigned_char_t **server_princ_name,
        unsigned32 *protect_level,
        unsigned32 *authn_svc,
        unsigned32 *authz_svc,
        unsigned32 *status);
```

Parameters

Input

binding Specifies the client binding handle from which to return the authentication and authorization information.

Output

privs Returns a handle to the authorization information for the client that made the remote procedure call on *binding*.

The server must cast this handle to the data type specified by *authz_svc*. The following table shows how to cast the return value:

Casts for Authorization Information		
For *authz_svc value***:**	*privs* **contains this data:**	**Use this cast:**
rpc_c_authz_none	A NULL value.	None
rpc_c_authz_name	The calling client's principal name.	**(unsigned_char_t *)**
rpc_c_authz_dce	The calling client's privilege attribute certificate.	**(sec_id_pac_t *)**

Note that **rpc_c_authz_none** is valid only if the *authn_svc* parameter is **rpc_c_authn_none**.

The data referenced by this parameter are read-only and should not be modified by the server. If the server wants to preserve any of the returned data, it must copy the data into server-allocated memory.

Specifying NULL prevents the routine from returning this parameter.

server_princ_name

Returns a pointer to the server principal name specified by the client that made the remote procedure call on *binding*. The content of the returned name and its syntax are defined by the authentication service in use.

Specifying NULL prevents the routine from returning this parameter. In this case, the caller does not have to call the **rpc_string_free()** routine.

protect_level Returns the protection level requested by the client that made the remote procedure call on *binding*. The protection level determines the degree to which authenticated communications between the client and the server are protected.

Specifying NULL prevents the routine from returning this parameter.

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 encrypt each remote procedure call argument value.

authn_svc Returns the authentication service requested by the client that made the remote procedure call on *binding*.

Specifying NULL prevents the routine from returning this parameter.

The possible 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.

authz_svc Returns the authorization service requested by the client that made the remote procedure call on *binding*.

Specifying NULL prevents the routine from returning this parameter.

The possible authorization services are as follows:

rpc_c_authz_none
Server performs no authorization. This is valid only if the *authn_svc* parameter is **rpc_c_authn_none**.

rpc_c_authz_name
Server performs authorization based on the client principal name.

rpc_c_authz_dce
Server performs authorization by using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with *binding*. Generally, access is checked against DCE access control lists (ACLs).

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 The routine completed successfully.

rpc_s_invalid_binding
The routine did not complete because of an invalid binding handle.

rpc_s_wrong_kind_of_binding
The routine did not complete because of the wrong kind of binding was specified for the operation.

rpc_s_binding_has_no_auth
The routine completed successfully, but the binding has no authentication information.

Description

The **rpc_binding_inq_auth_client()** routine returns authentication and authorization information associated with the client identified by *binding*. The calling server manager routine can use the returned data for authorization purposes.

Note: This call is provided only for compatibility with pre-DCE Version 1.1 applications. Applications based on DCE Version 1.1 and later releases of DCE should use the **rpc_binding_inq_auth_caller()** call.

The RPC runtime allocates memory for the returned *server_princ_name* parameter. The server is responsible for calling the **rpc_string_free()** routine for the returned parameter string.

For applications in which the client side uses the Interface Definition Language (IDL) **auto_handle** or **implicit_handle** attributes, the server side needs to be built with the IDL **explicit_handle** attribute specified in the attribute configuration file (ACF). Using **explicit_handle** provides *binding* as the first parameter to each server manager routine.

Return Values

No value is returned.

Related Information

Functions: **rpc_binding_inq_auth_info(3rpc)**, **rpc_binding_set_auth_info(3rpc)**, **rpc_string_free(3rpc)**.

rpc_binding_inq_auth_info

Purpose

Returns authentication and authorization information from a server binding handle; used by client applications

Synopsis

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

void rpc_binding_inq_auth_info(
        rpc_binding_handle_t binding,
        unsigned_char_t **server_princ_name,
        unsigned32 *protect_level,
        unsigned32 *authn_svc,
        rpc_auth_identity_handle_t *auth_identity,
        unsigned32 *authz_svc,
        unsigned32 *status);
```

Parameters

Input

binding Specifies the server binding handle from which to return the authentication and authorization information.

Output

server_princ_name
Returns a pointer to the expected principal name of the server referenced by *binding*. The content of the returned name and its syntax are defined by the authentication service in use.

Specifying NULL prevents the routine from returning this parameter. In this case, the caller does not have to call the **rpc_string_free()** routine.

protect_level Returns the protection level used for remote procedure calls made with *binding*. The protection level determines the degree to which

authenticated communications between the client and the server are protected.

Note that the returned level may be different from the level specified for *protect_level* on the call to **rpc_binding_set_auth_info()**. 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.

Specifying NULL prevents the routine from returning this parameter.

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 encrypt each remote procedure call parameter value.

authn_svc Returns the authentication service used for remote procedure calls made with *binding*.

Specifying NULL prevents the routine from returning this argument.

The possible 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.

auth_identity

Returns a handle for the data structure that contains the client's authentication and authorization credentials. This parameter must be cast as appropriate for the authentication and authorization services established via **rpc_binding_set_auth_info()**.

When using the **rpc_c_authn_dce_secret** authentication service and any authorization service, this value must be a **sec_login_handle_t** obtained from one of the following routines:

- **sec_login_setup_identity()**
- **sec_login_get_current_context()**
- **sec_login_newgroups()**

See the **sec_login_setup_identity(3sec)**, **sec_login_get_current_context(3sec)**, and **sec_login_newgroups(3sec)** reference pages for more information.

Specifying NULL prevents the routine from returning this parameter.

authz_svc Returns the authorization service used for remote procedure calls made with *binding*.

Specifying NULL prevents the routine from returning this parameter.

The possible authorization services are as follows:

rpc_c_authz_none
Server performs no authorization. This is valid only if the *authn_svc* parameter is **rpc_c_authn_none**.

rpc_c_authz_name
Server performs authorization based on the client principal name.

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 *binding*. Generally, access is checked against DCE access control lists (ACLs).

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 The routine completed successfully.

rpc_s_invalid_binding
The routine did not complete because of an invalid binding handle.

rpc_s_wrong_kind_of_binding
The routine did not complete because of the wrong kind of binding was specified for the operation.

rpc_s_binding_has_no_auth
The routine completed successfully, but the binding has no authentication information.

Description

The **rpc_binding_inq_auth_info()** routine returns authentication and authorization information associated with the specified server binding handle. The calling client associates the authentication and authorization data with the server binding handle by a prior call to the **rpc_binding_set_auth_info()** routine.

The RPC runtime allocates memory for the returned *server_princ_name* parameter. The caller is responsible for calling the **rpc_string_free()** routine for the returned parameter string.

Return Values

No value is returned.

Related Information

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

rpc_binding_inq_object

Purpose

Returns the object UUID from a binding handle; used by client or server applications

Synopsis

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

void rpc_binding_inq_object(
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a client or server binding handle.

Output

object_uuid Returns the object UUID found in the *binding* parameter. The object UUID is a unique identifier for an object for which a remote procedure call can be made.

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_binding_inq_object()** routine obtains the object UUID associated with a client or server binding handle. If no object UUID has been associated with the binding handle, this routine returns 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.

rpc_s_ok Success.

rpc_s_invalid_binding
Invalid binding handle.

Related Information

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

rpc_binding_reset

Purpose

Resets a server binding handle; used by client or management applications

Synopsis

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

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

Parameters

Input

binding Specifies the server binding handle to reset.

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_binding_reset()** routine disassociates a server instance from the server binding handle specified in the *binding* parameter. This routine removes the endpoint portion of the server address in the binding handle as well as any other server instance information in the binding handle. The host portion of the server address remains unchanged. The result is a partially bound server binding handle. This binding handle can rebind to another server instance on the previous host when it is later used to make a remote procedure call. The **rpc_intro(3rpc)** reference page contains an explanation of partially and fully bound binding handles.

This routine does not affect any authentication information for the *binding* parameter.

Suppose that a client can be serviced by any compatible server instance on the host specified in the binding handle. Then, the client can call the **rpc_binding_reset()** routine before making a remote procedure call using the binding handle specified in *binding*.

When the client makes the next remote procedure call using the reset server binding handle in *binding*, the client's RPC runtime uses a well-known endpoint from the client's interface specification, if any. Otherwise, the client's RPC runtime automatically communicates with the DCE host daemon (**dced**) on the specified remote host, to obtain the endpoint of a compatible server from the local endpoint map. If a compatible server is located, the RPC runtime updates *binding* with a new endpoint.

However, if a compatible server is not located, the client's remote procedure call fails. If the failed call uses a connection protocol (**ncacn**), it receives the **rpc_s_endpoint_not_found** status code. If the failed call uses a datagram protocol (**ncadg**), it receives the **rpc_s_comm_failure** status code.

If a server application wants to be available to clients making a remote procedure call on a reset binding handle, it registers all binding handles by calling **rpc_ep_register()** or **rpc_ep_register_no_replace()**. If, however, the IDL-generated file contains endpoint address information, then the application does not have to call either of these two routines.

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_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**.

rpc_binding_server_from_client

Purpose

Converts a client binding handle to a server binding handle; used by server applications

Synopsis

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

void rpc_binding_server_from_client(
        rpc_binding_handle_t client_binding,
        rpc_binding_handle_t *server_binding,
        unsigned32 *status);
```

Parameters

Input

client_binding
: Specifies the client binding handle to convert to a server binding handle.

Output

server_binding
: Returns a 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

When a remote procedure call arrives at a server, the RPC runtime creates a client binding handle to refer to information about the calling client (client binding information). The RPC runtime passes the client binding handle to the called remote procedure as the first input argument (which uses the **handle_t** type).

The **rpc_binding_server_from_client()** routine converts client binding information into server binding information corresponding to the client's system. When calling

this routine, the called remote procedure specifies the client binding handle, and the routine returns a partially bound server binding handle (that is, the newly constructed server binding information contains a network address for the client's system, but lacks an endpoint).

The server binding information also lacks authentication information, but the called procedure can add it by calling **rpc_binding_set_auth_info()**. The object UUID from the client binding information remains.

The **rpc_binding_server_from_client()** routine is relevant when a called remote procedure (the first remote procedure) needs to make its own remote procedure call (a nested procedure call) to a second remote procedure offered by a server on the system of the client that called the first remote procedure (that is, the original client). The partially bound server binding handle returned by the **rpc_binding_server_from_client()** routine ensures that a nested call requests the second remote procedure on the original client's system.

In a multithreaded RPC application, the second remote procedure can belong to a server that shares the original client's address space (that is, the server and client can operate jointly as a server/client instance). If the original client belongs to a server/client instance and the application requires the nested call to execute in that instance, the application must guarantee that the nested remote procedure call uses one of the instances' endpoints.

An application can provide this guarantee by meeting any of the following conditions:

- The interface possesses its own well-known endpoints, and the server elects to use these interface-specific endpoints (by calling the routine **rpc_server_use_protseq_if()** or **rpc_server_use_all_protseqs_if()**).
- The server uses server-specific endpoints, and the interface is offered by only one server/client instance per system.

  To use server-specific endpoints, a server either requests dynamic endpoints (by calling **rpc_server_use_protseq()** or **rpc_server_use_all_protseqs()**) or specifies its own well-known endpoints (by calling the routine **rpc_server_use_protseq_ep()**). The server must also register its server-specific endpoints in the local endpoint map (by calling **rpc_ep_register()**).
- The original client sets an object UUID into the server binding information of the first call (by calling **rpc_binding_set_object()**); the object UUID identifies the server/client instance.

The client can obtain the object UUID from the list of object UUIDs used to register the endpoints of the server/client instance. The client must select an object UUID that belongs exclusively to its instance.

Server binding information containing an object UUID impacts the selection of a manager for a remote procedure call; see the *DCE 1.2.2 Application Development Guide—Core Components* for a description of manager selection. The object UUID can either identify a particular resource offered by the companion server or, used as an instance UUID, the object UUID can identify the original client's server/client instance.

The object UUID is passed in the first remote procedure call as part of the client binding information and is retained in the server binding information. This server binding information is newly constructed by the **rpc_binding_server_from_client()** routine. When the second remote procedure call arrives at the original client's system, the DCE host daemon uses the object UUID to look for associated endpoints in the local endpoint map. To ensure that the object UUID is associated with the endpoints of the original server/client instance, the server must complete the following steps:

1. Obtain the UUID (for example, by calling **uuid_create()**).
2. Specify the UUID as part of registering endpoints for the interface of the second remote procedure (by calling **rpc_ep_register()** or **rpc_ep_register_no_replace()**).

   If the second remote procedure call will be routed to a manager of a nonnil type, then the server must also do the following:

   - Specify the type for the manager that implements that interface (by calling **rpc_server_register_if()**).
   - Set the object UUID to the same type as the manager (by calling **rpc_object_set_type()**).

- The first remote procedure call contains a distinct call argument used by the original client to pass server information that identifies its server/client instance.

  The first remote procedure call uses this information to route the second remote procedure call to the original server/client instance. For example, server information can be as follows:

  — A fully bound string binding that identifies the client's server/client instance.

If the first remote procedure receives this string binding, calling the **rpc_binding_server_from_client** routine is unnecessary. Instead, the first remote procedure requests a server binding handle for the string binding (by calling **rpc_binding_from_string_binding()**).

— An object UUID that is associated in the endpoint map with one or more endpoints of the original server/client instance.

The client can obtain the object UUID from the list of object UUIDs used to register the endpoints of the server/client instance. The client must select an object UUID that belongs exclusively to its instance, and pass that UUID as a call argument.

After calling the **rpc_binding_server_from_client()** routine, add the object UUID from the call argument to the newly constructed server binding information (by calling **rpc_binding_set_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_cant_getpeername
Cannot get peer name.

rpc_s_connection_closed
Connection closed.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding.

Related Information

Functions: **rpc_binding_free(3rpc)**, **rpc_binding_set_object(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_binding_set_auth_info

Purpose

Sets authentication and authorization information for a server binding handle; used by client applications

Synopsis

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

void rpc_binding_set_auth_info(
        rpc_binding_handle_t binding,
        unsigned_char_t *server_princ_name,
        unsigned32 protect_level,
        unsigned32 authn_svc,
        rpc_auth_identity_handle_t auth_identity,
        unsigned32 authz_svc,
        unsigned32 *status);
```

Parameters

Input

binding Specifies the server binding handle for which to set the authentication and authorization information.

server_princ_name
Specifies the principal name of the server referenced by *binding*. The content of the name and its syntax is defined by the authentication service in use.

A client that does not know the server principal name can call the **rpc_mgmt_inq_server_princ_name()** routine to obtain the principal name of a server that is registered for the required authentication service. Using a principal name obtained in this way means that the client is interested in one-way authentication. In other words, it means that the client does not care which server principal received the remote procedure

call request. The server, though, still verifies that the client is who the client claims to be.

protect_level Specifies the protection level for remote procedure calls made using *binding*. 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:

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

rpc_c_protect_level_pkt_integ is the default protection level for the DCE shared-secret key authentication service.

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
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.

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 **rpc_c_protect_level_pkt** instead.

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.

This is the highest protection level that is guaranteed to be present in the RPC runtime.

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

This is the highest protection level, but it may not be available 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 the *protect_level* parameter. The supported authentication services are as follows:

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 *binding*.

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_default
DCE default authentication service.

Note: The current default authentication service is DCE shared-secret key. Specifying **rpc_c_authn_default** is therefore equivalent to specifying **rpc_c_authn_dce_secret**.

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

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** obtained from one of the following routines:

- **sec_login_setup_identity()**
- **sec_login_get_current_context()**
- **sec_login_newgroups()**

Specify NULL to use the default security login context for the current address space.

authz_svc Specifies the authorization service implemented by the server for the interface of interest. 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:

rpc_c_authz_none
Server performs no authorization. This is valid only if the *authn_svc* parameter is **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 *binding*. Generally, access is checked against DCE access control lists (ACLs). This value cannot be used if *authn_svc* is **rpc_c_authn_none**.

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_binding_set_auth_info()** routine sets up the specified server binding handle so that it can be used to make authenticated remote procedure calls that include authorization information.

Unless a client calls **rpc_binding_set_auth_info()** with the parameters to set establish authentication and authorization methods, all remote procedure calls made on the *binding* binding handle are unauthenticated. Some authentication services (*authn_svc*) may need to communicate with the security service to perform this operation. Otherwise, they may receive the **rpc_s_comm_failure** status.

The *authn_svc* parameter specifies the authentication service to use. Since currently, there is only one available authentication service (DCE shared-secret key), the parameter currently functions to specify whether or not rpc calls will be authenticated and client PACs certified. If authentication is chosen, the *protect_level* parameter can specify a variety of protection levels, ranging from no authentication to the highest level of authentication and encryption. If the *protect_level* parameter is set to **rpc_c_protect_level_none**, no authentication is performed, regardless of the authentication service choosen.

The *authz_svc* parameter specifies the authorization service to use. If no authentication has been chosen (*authn_svc* of **rpc_c_authn_none**), then no authorization (*authz_svc* of **rpc_c_authz_none**) must be chosen as well. If authentication will be performed, you have two choices for authorization: name-based authorization and DCE authorization. The use of name based_authorization, which provides a server with a client's principal name, is not recommended. DCE authorization uses PACs, a trusted mechanism for conveying client authorization data to authenticated servers. PACs are designed to be used with the DCE ACL facility.

Whether the call actually wakes up in the server manager code or is rejected by the runtime depends on following conditions:

- If the client specified no authentication, then none is attempted by the RPC runtime. The call wakes up in the manager code whether the server specified authentication or not. This permits both authenticated and unauthenticated clients to call authenticated servers. When the manager receives an unauthenticated call, it needs to make a decision about how to proceed.
- If the client specified DCE secret key authentication and the server specified no authentication, then the runtime will fail the call, and it will never reach the manager routine.

- If both client and server specified DCE secret key authentication, then authentication will be carried out by the RPC runtime transparently. Whether the call reaches the server manager code or is rejected by the runtime depends on whether the authentication succeeded.

Although the RPC runtime is responsible any authentication that is carried out, the fact that the runtime will always permit unauthenticated clients to reach the manager code means that a manager access function typically does need to make an authentication check. When the manager access routine calls **rpc_binding_inq_auth_client()** it needs to check for a *status* of **rpc_s_binding_has_no_auth**. In this case, the client has specified no authentication and the manager access function needs to make an access decision based on this fact. Note that in such a case, no meaningful authentication or authorization information is returned from **rpc_binding_inq_auth_client()**.

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.

rpc_s_unknown_authn_service
Unknown authentication service.

rpc_s_authn_authz_mismatch
Requested authorization service is not supported by the requested authentication service.

rpc_s_unsupported_protect_level
Requested protection level is not supported.

Related Information

Functions: **rpc_binding_inq_auth_client(3rpc)**, **rpc_binding_inq_auth_info(3rpc)**, **rpc_mgmt_inq_dflt_protect_level(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**, **sec_login_get_current_context(3sec)**, **sec_login_newgroups(3sec)**, **sec_login_setup_identity(3sec)**.

rpc_binding_set_object

Purpose

Sets the object UUID value into a server binding handle; used by client applications

Synopsis

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

void rpc_binding_set_object(
        rpc_binding_handle_t binding,
        uuid_t *object_uuid,
        unsigned32 *status);
```

Parameters

Input

binding Specifies the server binding into which parameter *object_uuid* is set. Supply NULL to specify a nil UUID for this parameter.

object_uuid Specifies the UUID of the object serviced by the server specified in the *binding* parameter. The object UUID is a unique identifier for an object for which a remote procedure call can be made.

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_binding_set_object()** routine associates an object UUID with a server binding handle. This operation replaces the previously associated object UUID with the UUID in the *object_uuid* parameter.

To set the object UUID to the nil UUID, specify NULL or the nil UUID for the *object_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.

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_binding_from_string_binding(3rpc)**, **rpc_binding_inq_object(3rpc)**.

rpc_binding_to_string_binding

Purpose

Returns a string representation of a binding handle; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h

void rpc_binding_to_string_binding(
        rpc_binding_handle_t binding,
        unsigned_char_t **string_binding,
        unsigned32 *status);
```

Parameters

Input

binding Specifies a client or server binding handle to convert to a string representation of a binding handle.

Output

string_binding
Returns a pointer to the string representation of the binding handle specified in 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_binding_to_string_binding()** routine converts a client or server binding handle to its string representation.

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

If the binding handle in the *binding* parameter contains a nil object UUID, the object UUID field is not included in the returned string.

To parse the returned *string_binding* parameter, call **rpc_string_binding_parse()**.

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_getpeername
Cannot get peer name.

rpc_s_connection_closed
Connection closed.

rpc_s_invalid_binding
Invalid binding handle.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_string_binding_parse(3rpc)**, **rpc_string_free(3rpc)**.

rpc_binding_vector_free

Purpose

Frees the memory used to store a vector and binding handles; used by client or server applications

Synopsis

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

void rpc_binding_vector_free(
        rpc_binding_vector_t **binding_vector,
        unsigned32 *status);
```

Parameters

Input/Output

binding_vector
: Specifies the address of a pointer to a vector of server binding handles. 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_binding_vector_free()** routine frees the memory used to store a vector of server binding handles. The freed memory includes both the binding handles and the vector itself.

A server obtains a vector of binding handles by calling **rpc_server_inq_bindings()**. A client obtains a vector of binding handles by calling **rpc_ns_binding_lookup_next()**. Call **rpc_binding_vector_free()** if you have used either of these routines.

The **rpc_binding_free()** routine frees individual elements of the vector. If an element is freed with this routine, the NULL element entry replaces it; **rpc_binding_vector_free()** ignores such an 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_invalid_arg
Invalid argument.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_binding_free(3rpc)**, **rpc_ns_binding_lookup_next(3rpc)**, **rpc_server_inq_bindings(3rpc)**.

rpc_cs_binding_set_tags

Purpose

Places code set tags into a server binding handle; used by client applications

Synopsis

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

void rpc_cs_binding_set_tags(
        rpc_binding_handle_t *binding,
        unsigned32 sending_tag,
        unsigned32 desired_receiving_tag,
        unsigned16 sending_tag_max_bytes,
        error_status_t *status);
```

Parameters

Input/Output

binding On input, specifies the server binding handle to modify with tag information. This handle is the binding handle returned by the **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()** routine. On output, returns the server binding handle modified with code set tag information. The server stub retrieves the tag information from the binding handle and uses it to invoke the appropriate buffer sizing and code set conversion routines.

Input

sending_tag Specifies the code set value for the code set in which client data to be sent to the server is to be encoded. If the client is not sending any data, set this value to the client's current code set. This step prevents the code set conversion routine from being invoked.

desired_receiving_tag
Specifies the code set value for the code set in which the client prefers data to be encoded when sent back from the server. If the client is not planning to receive any data from the server, set this value to the server's

current code set. This step prevents the code set conversion routine from being invoked.

sending_tag_max_bytes
Specifies the maximum number of bytes that a code set requires to encode one character. The value is the *c_max_bytes* value associated with the code set value (*c_set*) used as the *sending_tag* value.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The routine can also return status codes generated by the **rpc_rgy_get_codesets()** routine.

Description

The **rpc_cs_binding_set_tags()** 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. These routines are used to enable automatic code set conversion between client and server for character representations that are not part of the DCE portable character set.

Client applications use the **rpc_cs_binding_set_tags()** routine to add code sets tag information to the binding handle of a compatible server. The tag information specified in the routine is usually obtained from a character and code sets evaluation routine (which is typically a user-written routine).

The *sending_tag* value identifies the code set encoding that the client is using to send international character data to the server. The *desired_receiving_tag* value indicates to the server the code set that the client prefers the server to use when sending return international character data. The *sending_tag_max_bytes* value is the number of bytes the sending code set uses to encode one character.

Client applications that use the **rpc_cs_eval_with_universal()** or **rpc_cs_eval_without_universal()** routines do not need to call this routine because these routines set tag information in the server binding handle as part of their operation. Application developers who are writing their own character and code sets evaluation routines need to include code that sets tags in a server binding handle.

The **rpc_cs_binding_set_tags()** routine provides this function and can be used in user-written evaluation routines, or alone if the application does not need to perform

evaluation. In this case, the routine provides a short cut for application programmers whose applications do not need to evaluate for character and code set compatibility.

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

rpc_s_no_memory

Related Information

Functions: **rpc_cs_eval_with_universal(3rpc)**, **rpc_cs_eval_without_universal(3rpc)**, **rpc_cs_get_tags(3rpc)**.

rpc_cs_char_set_compat_check

Purpose

Evaluates character set compatibility between a client and a server; used by client applications

Synopsis

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

void rpc_cs_char_set_compat_check(
        unsigned32 client_rgy_code_set_value,
        unsigned32 server_rgy_code_set_value,
        error_status_t *status);
```

Parameters

Input

client_rgy_code_set_value
: The registered hexadecimal value that uniquely identifies the code set that the client is using as its local code set.

server_rgy_code_set_value
: The registered hexadecimal value that uniquely identifies the code set that the server is using as its local code set.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The routine can also return status codes from the **dce_cs_rgy_to_loc()** routine.

Description

The **rpc_cs_char_set_compat_check()** 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_cs_char_set_compat_check()** routine provides a method for determining character set compatibility between a client and a server; if the server's character set is incompatible with that of the client, then connecting to that server is most likely not acceptable, since massive data loss would result from such a connection.

The RPC routines that perform character and code sets evaluation use the **rpc_cs_char_set_compat_check()** routine in their character sets and code sets compatibility checking procedure. The routine takes the registered integer values that represent the code sets that the client and server are currently using and calls the code set registry to obtain the registered values that represent the character set(s) that the specified code sets support. If both client and server support just one character set, the routine compares client and server registered character set values to determine whether or not the sets are compatible. If they are not, the routine returns the status message **rpc_s_ss_no_compat_charsets**.

If the client and server support multiple character sets, the routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the caller.

Client and server applications that use the DCE RPC code sets evaluation routines **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()** do not need to call this routine explicitly because these DCE RPC routines call it on their behalf.

Client applications that do not use the DCE RPC code sets evaluation routines can use the **rpc_cs_char_set_compat_check()** routine in their code sets evaluation code as part of their procedure for determining character and code set compatibility with 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

rpc_s_ss_no_compat_charsets

Related Information

Functions: **rpc_cs_eval_with_universal(3rpc)**, **rpc_cs_eval_without_universal(3rpc)**, **rpc_cs_get_tags(3rpc)**, **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_rgy_get_codesets(3rpc)**.

rpc_cs_eval_with_universal

Purpose

Evaluates a server's supported character sets and code sets during the server binding selection process; used indirectly by client applications

Synopsis

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

void rpc_cs_eval_with_universal(
        rpc_ns_handle_t binding_handle,
        idl_void_p_t eval_args,
        idl_void_p_t *context);
```

Parameters

Input

binding_handle
: The server binding handle.

eval_args
: An opaque data type that contains matching criteria that the routine uses to perform character and code sets compatibility evaluation.

Input/Output

context
: An opaque data type that contains search context to perform character and code sets compatibility evaluation. The routine returns the result of the evaluation in a field within *context*.

Description

The **rpc_cs_eval_with_universal()** routine is a DCE RPC character and code sets evaluation routine that can be added to an import context. The routine provides a mechanism for a client application that is passing character data in a heterogeneous character set and code sets environment to evaluate a server's character and code sets compatibility before establishing a connection with it.

Client applications do not call **rpc_cs_eval_with_universal()** directly. Instead, they add it to the import context created by the **rpc_ns_binding_import_begin()** routine by calling the routine **rpc_ns_import_ctx_add_eval()** and specifying the routine name and the RPC server entry name to be evaluated. When the client application calls the **rpc_ns_binding_import_next()** routine to import compatible binding handles for servers, this routine calls **rpc_cs_eval_with_universal()**, which applies client-server code sets compatibility checking as another criteria for compatible binding selection.

The **rpc_cs_eval_with_universal()** routine directs the routine **rpc_ns_binding_import_next()** to reject servers with incompatible character sets. If client and server character sets are compatible, but their supported code sets are not, the routine establishes tags that direct the client and/or server stubs to convert character data to the user-defined (if any) or default intermediate code set, which is the ISO10646 (or *universal*) code set.

Note: Application programmers need not pay attention to the arguments of this routine. Programmers only need to use the routine **rpc_ns_import_ctx_add_eval()** to set the routine, for example:

```
rpc_ns_import_ctx_add_eval(
&import_context,
rpc_c_eval_type_codesets,
(void *) nsi_entry_name,
rpc_cs_eval_with_universal,
NULL,
&status);
```

Permissions Required

No permissions are required.

Return Values

No value is returned.

Related Information

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

rpc_cs_eval_without_universal

Purpose

Evaluates a server's supported character sets and code sets during the server binding selection process; used indirectly by client applications

Synopsis

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

void rpc_cs_eval_without_universal(
        rpc_ns_handle_t binding_handle,
        idl_void_p_t eval_args,
        idl_void_p_t *context);
```

Parameters

Input

binding_handle
: The server binding handle.

eval_args
: An opaque data type that contains matching criteria that the routine uses to perform code sets compatibility evaluation.

Input/Output

context
: An opaque data type that contains search context to perform character and code sets compatibility evaluation. The routine returns the result of the evaluation in a field within *context*.

Description

The **rpc_cs_eval_without_universal()** routine is a DCE RPC character and code sets evaluation routine that can be added to an import context. The routine provides a mechanism for a client application that is passing character data in a heterogeneous character set and code sets environment to evaluate a server's character and code sets compatibility before establishing a connection with it.

Client applications do not call **rpc_cs_eval_without_universal()** directly. Instead, they add it to the import context created by the **rpc_ns_binding_import_begin()** routine by calling the routine **rpc_ns_import_ctx_add_eval()** and specifying the routine name and the RPC server entry name to be evaluated. When the client application calls the **rpc_ns_binding_import_next()** routine to import compatible binding handles for servers, this routine calls **rpc_cs_eval_without_universal()**, which applies client-server code sets compatibility checking as another criteria for compatible binding selection.

The **rpc_cs_eval_without_universal()** routine directs the routine **rpc_ns_binding_import_next()** to reject servers with incompatible character sets. The routine also directs the **rpc_ns_binding_import_next()** routine to reject servers whose supported code sets are incompatible with the client's supported code sets; that is, it does not resort to using an intermediate code set as a last resort.

Note: Application programmers need not pay attention to the arguments of this routine. Programmers only need to use the routine **rpc_ns_import_ctx_add_eval()** to set the routine, for example:

```
rpc_ns_import_ctx_add_eval(
&import_context,
rpc_c_eval_type_codesets,
(void *) nsi_entry_name,
rpc_cs_eval_without_universal,
NULL,
&status);
```

Permissions Required

No permissions are required.

Return Values

No value is returned.

Related Information

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

rpc_cs_get_tags

Purpose

Retrieves code set tags from a binding handle; used by client and server applications

Synopsis

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

void rpc_cs_get_tags(
        rpc_binding_handle_t binding,
        boolean32 server_side,
        unsigned32 *sending_tag,
        unsigned32 *desired_receiving_tag,
        unsigned32 *receiving_tag,
        error_status_t *status);
```

Parameters

Input

binding Specifies the target binding handle from which to obtain the code set tag 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()** routines.

server_side Indicates whether a client stub or a server stub is calling the routine.

desired_receiving_tag
(Server stub only) Specifies the code set value for the code set in which the client prefers data to be encoded when sent back from the server. The client stub passes this value in the RPC call. If the routine is retrieving code set tags for an operation that does not specify a desired receiving tag parameter (the **cs_drtag** ACF parameter attribute has not been applied to one of the operation's parameters), this value is NULL.

Output

sending_tag (Client stub only) Specifies the code set value for the code set in which client data to be sent to the server is to be encoded. If the routine is retrieving code set tags for an operation that does not specify a sending tag parameter (the **cs_stag** ACF parameter attribute has not been applied to one of the operation's parameters), this value is NULL.

desired_receving_tag
(Client stub only) Specifies the code set value for the code set in which the client prefers to receive data sent back to it from the server. If the routine is retrieving code set tags for an operation that does not specify a desired receiving tag parameter (the **cs_drtag** ACF parameter attribute has not been applied to one of the operation's parameters), this value is NULL.

receiving_tag
(Server stub only) Specifies the code set value for the code set in which the server is to encode data to be sent back to the client. If the routine is retrieving code set tags for an operation that does not specify a receiving tag parameter (the **cs_rtag** ACF parameter attribute has not been applied to one of the operation's parameters), this value is NULL.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. If code set compatibility evaluation is performed, error values can also be returned from the following routines:

- **rpc_rgy_get_codesets()**
- **rpc_ns_binding_inq_entry_name()**
- **rpc_ns_mgmt_read_codesets()**.

Description

The **rpc_cs_get_tags()** 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_cs_get_tags()** routine is a DCE RPC routine that RPC stubs can use to retrieve the code set values to be used to tag international character data to be sent over the network. In general, the code set values to be used as tags are determined by a

character and code sets evaluation routine, which is invoked from the client application code. However, application programmers can use other methods to establish values for code set tags.

RPC stubs call the **rpc_cs_get_tags()** routine before they call the buffer sizing routines ***_net_size()** and the code set conversion routines ***_netcs()**. The **rpc_cs_get_tags()** routine provides the stubs with code set values to use as input to the buffer sizing routines (to determine whether or not buffer storage needs to be allocated for conversion) and as input to the code set conversion routines (to determine whether conversion is necessary, and if so, which host code set converter to invoke).

Client and server stubs call the **rpc_cs_get_tags()** routine before they marshall any data. When called from the client stub, the boolean value *server_side* is set to FALSE to indicate that the client stub has invoked the routine. The binding handle is the handle to a compatible server that is returned by the routines **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()**. If the client has added a code sets evaluation routine to the binding import procedure (by calling the routine **rpc_ns_import_ctx_add_eval()**), the binding handle will contain the conversion method and the code set values to set for the client's sending tag and desired receiving tag. If the binding handle does not contain the results of an evaluation, the **rpc_cs_get_tags()** routine will perform the character/code sets evaluation within the client stub and set the client code set tag values itself.

On the client side, the output of the routine is the code set value that represents the client's sending tag and the code set value that represents the client's desired receiving tag. If the conversion method is ''client makes it right'' (CMIR), the sending tag and desired receiving tags will be set to the code set value of the server's local code set. If the conversion method is ''server makes it right'' (SMIR), the sending tag and desired receiving tag will be set to the client's local code set value. If the conversion method is ''receiver makes it right'' (RMIR), the sending tag is the client's code set, and the desired receiving tag is the server's code set.

When called from the server stub, the boolean value *server_side* is set to TRUE to indicate that the server stub has invoked the routine.

The server stub specifies the code set value given in the client's desired receiving tag as input to the routine. The **rpc_cs_get_tags()** routine sets the code set value in *desired_receiving_tag* to *receiving_tag* and returns this value as output to the server stub. The server stub will then use the code set value in *receiving_tag* as the code set to use for data it sends back to the client.

Application programmers who want their applications to use the **rpc_cs_get_tags()** routine to retrieve code set tag information as part of the automatic code set conversion

process specify the routine name as the argument to the ACF attribute **cs_tag_rtn** when developing their internationalized RPC application.

Application programmers can also write their own code set tags retrieval routine that RPC stubs can call; in this case, they specify the name of this routine as the argument to the ACF attribute **cs_tag_rtn** instead of specifying the DCE RPC routine **rpc_cs_get_tags()**. Application programmers can also use the automatic code conversion mechanism, but design their applications so that the code set tags are set explicitly in the application instead of in the stubs.

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_invalid_codeset_tag
The result of the client-side evaluation used an invalid code set tag.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **cs_byte_to_netcs(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ep_register

Purpose

Adds to, or replaces, server address information in the local endpoint map; used by server applications

Synopsis

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

void rpc_ep_register(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

Parameters

Input

if_handle Specifies an interface specification to register with the local endpoint map.

binding_vec Specifies a vector of binding handles over which the server can receive remote procedure calls.

object_uuid_vec
Specifies a vector of object UUIDs that the server offers. The server application constructs this vector.

Supply the value NULL to indicate there are no object UUIDs to register.

annotation Defines a character string comment applied to each cross product element added to the local endpoint map. The string can be up to 64 characters long, including the NULL terminating character. Specify NULL or the string **\0** if there is no annotation string.

The string is used by applications for informational purposes only. The RPC runtime does not use this string to determine which server instance a client communicates with, or for enumerating endpoint map 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_ep_register()** routine adds elements to, or replaces elements in, the local host's endpoint map.

Each element in the local endpoint map logically contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information
- Object UUID (optional)
- Annotation (optional)

A server uses this routine, instead of **rpc_ep_register_no_replace()**, when only a single instance of the server runs on the server's host. Use this routine if, at any time, no more than one server instance offers the same interface UUID, object UUID, and protocol sequence.

When local endpoint map elements are not replaced, obsolete elements accumulate each time a server instance stops running without calling **rpc_ep_unregister()**. Periodically the DCE host daemon identifies these obsolete elements and removes them. However, during the time between these removals the obsolete elements increase the chance that a client will receive endpoints to nonexistent servers. The client then wastes time trying to communicate with these servers before obtaining another endpoint.

Using this routine to replace any existing local endpoint map elements reduces the chance that a client will receive the endpoint of a nonexistent server instance.

Suppose an existing element in the local endpoint map matches the interface UUID, binding information exclusive of the endpoint, and object UUID of an element this routine provides. The routine changes the endpoint map according to the elements' interface major and minor version numbers, as shown in the following table:

Existing Element	Relationship	Provided Element	Routine's Action
Major version number	Not equal to	Major version number	Ignores minor version number relationship and adds a new endpoint map element. The existing element remains unchanged.
Major version number	Equal to	Major version number	Acts according to the minor version number relationship.
Minor version number	Equal to	Minor version number	Replaces the endpoint of the existing element based on the provided information.
Minor version number	Less than	Minor version number	Replaces the existing element based on the provided information.
Minor version number	Greater than	Minor version number	Ignores the provided information. The existing element remains unchanged.

For example, suppose under these circumstances that the existing interface version number is 1.3 (major.minor) and the provided version number is 2.0. The routine adds a new endpoint map element with interface version number 2.0 and does not change the element with version number 1.3. However, if the existing interface version number is 1.5 and the provided version number is 1.4, the routine does not change the endpoint map.

A server program calls this routine to register endpoints that have been specified by calling any of the following routines:

- **rpc_server_use_all_protseqs()**

- **rpc_server_use_protseq()**
- **rpc_server_use_protseq_ep()**

A server that calls only the **rpc_server_use_all_protseqs_if()** or **rpc_server_use_protseq_if()** routines does not need to call this routine. In such cases, the client's runtime uses an endpoint from the client's interface specification to fill in a partially bound binding handle. However, it is recommended that you also register well-known endpoints that the server specifies (registering endpoints from interface definitions is unnecessary).

If the server also exports to the name service database, the server calls this routine with the same *if_handle*, *binding_vec* and *object_uuid_vec* parameters as the server uses when calling the **rpc_ns_binding_export()** routine.

The **rpc_ep_register()** routine communicates with the DCE host daemon (**dced**), which in turn communicates with the local endpoint map. The routine communicates using one of the protocol sequences specified in one of the binding handles in *binding_vec*. Attempting to register a binding that specifies a protocol sequence that the DCE host daemon is not listening on results in the failure of **rpc_ep_register()**. The routine indicates this failure by placing the value **rpc_s_comm_failure** into *status*.

For information about how the endpoint map service selects an element for an interface ID and an object UUID, see the RPC information in the *DCE 1.2.2 Application Development Guide—Core Components*. This guide explains how the endpoint map service searches for the endpoint of a server that is compatible with a client. If the client supplies a nonnil object UUID that is not in the endpoint map, or the client supplies a nil object UUID, the search can succeed, but only if the server has registered a nil object UUID using the **rpc_ep_register()** or **rpc_ep_register_no_replace()** routines. The *object_uuid_vec* parameter can contain both nil and nonnil object UUIDs for the routine to place into endpoint map elements.

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

This routine creates a cross product from the *if_handle*, *binding_vec* and *object_uuid_vec* parameters, and adds each element in the cross product as a separate registration in the local endpoint map. If you supply NULL to *object_uuid_vec*, the corresponding elements in the cross product contain a nil object UUID.

For example, suppose that *if_handle* has the value **ifhand**, *binding_vec* has the values **b1**, **b2**, **b3**, and *object_uuid_vec* has the values **u1**, **u2**, **u3**, **u4**. The resulting 12 elements in the cross product are as follows:

(ifhand,b1,u1) (ifhand,b1,u2) (ifhand,b1,u3) (ifhand,b1,u4)
(ifhand,b2,u1) (ifhand,b2,u2) (ifhand,b2,u3) (ifhand,b2,u4)
(ifhand,b3,u1) (ifhand,b3,u2) (ifhand,b3,u3) (ifhand,b3,u4)

(An annotation string is part of each of these 12 elements.)

Return Values

No value is returned.

Errors

rpc_s_ok Success.

ept_s_cant_access
Error reading endpoint database.

ept_s_cant_create
Error creating endpoint database.

ept_s_cant_perform_op
Cannot perform requested operation.

ept_s_database_invalid
Endpoint map database invalid.

ept_s_invalid_entry
Invalid database entry.

ept_s_update_failed
Update failed.

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_no_bindings
No bindings.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_register_no_replace(3rpc)**, **rpc_ep_resolve_binding(3rpc)**, **rpc_ep_unregister(3rpc)**, **rpc_mgmt_ep_unregister(3rpc)**, **rpc_ns_binding_export(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)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ep_register_no_replace

Purpose

Adds to server address information in the local endpoint map; used by server applications

Synopsis

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

void rpc_ep_register_no_replace(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned_char_t *annotation,
        unsigned32 *status);
```

Parameters

Input

if_handle Specifies an interface specification to register with the local endpoint map.

binding_vec Specifies a vector of binding handles over which the server can receive remote procedure calls.

object_uuid_vec
Specifies a vector of object UUIDs that the server offers. The server application constructs this vector.

Supply the value NULL to indicate there are no object UUIDs to register.

annotation Defines a character string comment applied to each cross-product element added to the local endpoint map. The string can be up to 64 characters long, including the NULL terminating character. Specify NULL or the string **\0** if there is no annotation string.

The string is used by applications for informational purposes only. The RPC runtime does not use this string to determine which server instance a client communicates with, or for enumerating endpoint map 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_ep_register_no_replace()** routine adds elements to the local host's endpoint map. The routine does not replace existing elements. Otherwise, this routine is identical to **rpc_ep_register()**.

Each element in the local endpoint map logically contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information
- Object UUID (optional)
- Annotation (optional)

A server uses this routine, instead of **rpc_ep_register()**, when multiple instances of the server run on the same host. Use this routine if, at any time, more than one server instance offers the same interface UUID, object UUID, and protocol sequence.

Since this routine does not replace elements, calling servers must unregister (that is, remove) themselves before they stop running. Otherwise, when local endpoint map elements are not replaced, obsolete elements accumulate each time a server instance stops running without calling **rpc_ep_unregister()**. Periodically the DCE host daemon identifies obsolete elements and removes them from the local endpoint map. However, during the time between these removals, the obsolete elements increase the chance that a client will receive endpoints to nonexistent servers. The client then wastes time trying to communicate with these servers before obtaining another endpoint.

A server program calls this routine to register endpoints that were specified by calling any of the following routines:

- **rpc_server_use_all_protseqs()**
- **rpc_server_use_protseq()**
- **rpc_server_use_protseq_ep()**

A server that calls only the **rpc_server_use_all_protseqs_if()** or **rpc_server_use_protseq_if()** routine does not need to call this routine. In such cases, the client's runtime uses an endpoint from the client's interface specification to fill in a partially bound binding handle. However, it is recommended that you also register well-known endpoints that the server specifies (registering endpoints from interface definitions is unnecessary).

If the server also exports to the name service database, the server calls this routine with the same *if_handle*, *binding_vec* and *object_uuid_vec* parameters as the server uses when calling the **rpc_ns_binding_export()** routine.

The **rpc_ep_register_no_replace()** routine communicates with the DCE host daemon (**dced**), which in turn communicates with the local endpoint map. The routine communicates using one of the protocol sequences specified in one of the binding handles in *binding_vec*. Attempting to register a binding that specifies a protocol sequence that the DCE host daemon is not listening on results in the failure of **rpc_ep_register_no_replace()**. The routine indicates this failure by placing the value **rpc_s_comm_failure** into *status*.

For information about how the endpoint map service selects an element for an interface ID and an object UUID, see the RPC information in the *DCE 1.2.2 Application Development Guide—Core Components*. This guide explains how the endpoint map service searches for the endpoint of a server that is compatible with a client. If the client supplies a nonnil object UUID that is not in the endpoint map, or the client supplies a nil object UUID, the search can succeed, but only if the server has registered a nil object UUID using the **rpc_ep_register_no_replace()** or **rpc_ep_register()** routine. The *object_uuid_vec* parameter can contain both nil and nonnil object UUIDs for the routine to place into endpoint map elements.

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

This routine creates a cross-product from the *if_handle*, *binding_vec* and *object_uuid_vec* parameters, and adds each element in the cross-product as a separate registration in the local endpoint map. If you supply NULL to *object_uuid_vec*, the corresponding elements in the cross-product contain a nil object UUID. The **rpc_ep_register()** routine's reference page summarizes the contents of an element in the local endpoint map.

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 endpoint database.

ept_s_cant_create
Error creating endpoint database.

ept_s_cant_perform_op
Cannot perform requested operation.

ept_s_database_invalid
Endpoint map database invalid.

ept_s_invalid_entry
Invalid database entry.

ept_s_update_failed
Update failed.

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_no_bindings
No bindings.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

Functions: **rpc_ep_register(3rpc)**, **rpc_ep_resolve_binding(3rpc)**, **rpc_ep_unregister(3rpc)**, **rpc_mgmt_ep_unregister(3rpc)**, **rpc_ns_binding_export(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)**.

Books: *DCE 1.2.2 Application Development Guide—Core Components*.

rpc_ep_resolve_binding

Purpose

Resolves a partially bound server binding handle into a fully bound server binding handle; used by client and management applications

Synopsis

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

void rpc_ep_resolve_binding(
        rpc_binding_handle_t binding,
        rpc_if_handle_t if_handle,
        unsigned32 *status);
```

Parameters

Input/Output

binding Specifies a partially bound server binding handle to resolve into a fully bound server binding handle.

if_handle Contains a stub-generated data structure that specifies the interface of interest.

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 the **rpc_ep_resolve_binding()** routine to resolve a partially bound server binding handle into a fully bound server binding handle.

Resolving binding handles requires an interface UUID and an object UUID. The object UUID can be a nil UUID. The RPC runtime requests the DCE host daemon's endpoint mapper service, on the host that the *binding* parameter specifies, to look

up an endpoint for a compatible server instance. The endpoint mapper service finds the endpoint by looking in the local endpoint map for the interface UUID from the *if_handle* parameter and for the object UUID in the *binding* parameter.

The **rpc_ep_resolve_binding()** routine depends on whether the specified binding handle is partially bound or fully bound. When the application specifies a partially bound handle, the routine produces the following results:

- If no compatible server instances are registered in the local endpoint map, the routine returns the **ept_s_not_registered** status code.
- If one compatible server instance is registered in the local endpoint map, the routine returns a fully bound binding handle in *binding* and the **rpc_s_ok** status code.
- If more than one compatible server instance is registered in the local endpoint map, the routine randomly selects one. It then returns the corresponding fully bound binding handle in *binding* and the **rpc_s_ok** status code.

When the application specifies a fully bound binding handle, the routine returns the specified binding handle in *binding* and the **rpc_s_ok** status code. The routine makes no request of the DCE host daemon.

In neither the partially bound case nor the fully bound case does the routine contact a compatible server instance.

Using This Routine

For each server instance, the RPC runtime automatically provides routines (the **rpc_mgmt_*** routines) that form an RPC management interface. If a server instance registers any application-provided interfaces, the RPC runtime automatically registers the RPC-provided management interface with the local endpoint map for that server instance.

An application can call **rpc_ep_resolve_binding()** at any time with either a partially bound or a fully bound handle. However, applications typically call this routine to avoid calling a routine in the management interface with a partially bound handle.

An application can have a partially bound binding handle at the following times:

- After importing a binding handle.
- After resetting a binding handle.
- After converting a string binding without an endpoint to a binding handle.

If an application calls an application-provided remote procedure using a partially bound handle, the RPC runtime automatically asks the DCE host daemon to resolve the binding handle into a fully bound handle. This fully bound binding handle corresponds to the RPC interface of the called remote procedure and the requested object, if any. The application can then use this fully bound handle to make remote management calls, so calling the **rpc_ep_resolve_binding()** routine is unnecessary.

When a high proportion of all servers in an environment offers the same interface, the interface is known as a pervasive one. The RPC management interface is a pervasive interface in all environments that use DCE RPC.

Using this routine to unambiguously locate compatible server instances applies to application-pervasive interfaces as well as to the RPC management interface.

Partially Bound Handles with a Nonnil Object UUID

If the application has a partially bound handle with a nonnil object UUID, the application can decide not to call the **rpc_ep_resolve_binding()** routine before calling a procedure in the management interface. In this case the remote management call is sent to a server instance, registered on the remote host, that offers that object UUID.

After completing the remote management call, the application has a fully bound handle to that server instance. The server instance that the handle specifies probably offers the nonmanagement interfaces of interest to the calling application. However, if you want to be certain of obtaining a fully bound handle to a server instance that offers the interfaces needed for later remote procedure calls, call the **rpc_ep_resolve_binding()** routine.

Partially Bound Handles with a Nil Object UUID

When an application makes a remote procedure or management call using a partially bound handle with a nil object UUID, the DCE host daemon searches for a compatible server instance. The search is based on the nil object UUID and the UUID of the interface to which the call belongs.

All server instances that register any RPC interface automatically offer the RPC management interface. When an application makes a remote management call using a partially bound handle with a nil object UUID, the DCE host daemon on the remote host cannot distinguish among server instances registered in the local endpoint map.

When the DCE host daemon cannot distinguish among these instances it selects *any* server instance. After completing the remote management call, the calling application has a fully bound handle. However, the server instance that the handle represents probably does not offer the nonmanagement interfaces that interest the application.

The remote RPC management routines avoid this ambiguity. They do this by returning the status **rpc_s_binding_incomplete** if the provided binding handle is a partially bound one with a nil object UUID.

An application wanting to contact servers that have exported and registered interfaces with a nil object UUID calls routine **rpc_ep_resolve_binding()**. The application obtains a fully bound binding handle for calling remote management procedures in a server instance that also offers the remote procedures in the application-specific interface.

Note that an application that wants to manage all the server instances on a host does not call **rpc_ep_resolve_binding()**. Instead, the application obtains fully bound binding handles for each server instance by calling the routines **rpc_mgmt_ep_elt_inq_*()**.

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_not_registered
No entries found.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

rpc_s_rpcd_comm_failure
Communications failure while trying to reach the endpoint map.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_reset(3rpc)**, **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_ep_unregister

Purpose

Removes server address information from the local endpoint map; used by server applications

Synopsis

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

void rpc_ep_unregister(
        rpc_if_handle_t if_handle,
        rpc_binding_vector_t *binding_vec,
        uuid_vector_t *object_uuid_vec,
        unsigned32 *status);
```

Parameters

Input

if_handle Specifies an interface specification to remove (that is, unregister) from the local endpoint map.

binding_vec Specifies a vector of binding handles to remove.

object_uuid_vec
Specifies a vector of object UUIDs to remove. The server application constructs this vector. This routine removes all local endpoint map elements that match the specified *if_handle* parameter, *binding_vec* parameter, and object UUIDs.

This is an optional parameter. The value NULL indicates there are no object UUIDs to remove.

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_ep_unregister()** routine removes elements from the local host's endpoint map. A server application calls this routine only if the server has registered endpoints previously and the server wishes to remove that address information from the local endpoint map.

A server program is able to remove its own local endpoint map elements (server address information) based on either of the following:

- The interface specification.
- The interface specification and the object UUIDs of resources offered.

The server calls the **rpc_server_inq_bindings()** routine to obtain the required *binding_vec* parameter. To remove selected endpoints, the server can remove individual elements from *binding_vec* before calling this routine. (See the explanation of a binding vector in the **rpc_intro(3rpc)** reference page for more information about removing a single element from a vector of binding handles.)

This routine creates a cross product from the *if_handle*, *binding_vec* and *object_uuid_vec* parameters and removes each element in the cross product from the local endpoint map. The **rpc_ep_register()** routine's reference page summarizes the contents of a cross product in the local endpoint map.

Servers must always call the **rpc_ep_unregister()** routine to remove their endpoints from the local endpoint map before they exit. Otherwise, stale information will be in the local endpoint map. However, if a server prematurely removes endpoints (the server is not in the process of exiting), clients that do not already have fully bound binding handles to the server will not be able to send remote procedure calls to the server.

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 endpoint database.

ept_s_cant_create
Error creating endpoint database.

ept_s_cant_perform_op
Cannot perform requested operation.

ept_s_database_invalid
Endpoint map database invalid.

ept_s_invalid_entry
Invalid database entry.

ept_s_update_failed
Update failed.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_no_bindings
No bindings.

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_unregister(3rpc)**, **rpc_ns_binding_unexport(3rpc)**, **rpc_server_inq_bindings(3rpc)**.

rpc_if_id_vector_free

Purpose

Frees a vector and the interface identifier structures it contains; used by client, server, or management applications

Synopsis

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

void rpc_if_id_vector_free(
        rpc_if_id_vector_t **if_id_vector,
        unsigned32 *status);
```

Parameters

Input/Output

if_id_vector Specifies the address of a pointer to a vector of interface information. 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_if_id_vector_free()** routine frees the memory used to store a vector of interface identifiers. This includes memory used by the interface identifiers and the vector itself. On return this routine sets the *if_id_vector* parameter to NULL.

To obtain a vector of interface identifiers, call **rpc_ns_mgmt_entry_inq_if_ids()** or **rpc_mgmt_inq_if_ids()**. Call **rpc_if_id_vector_free()** if you have used either of these routines.

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.

Related Information

Functions: **rpc_if_inq_id(3rpc)**, **rpc_mgmt_inq_if_ids(3rpc)**, **rpc_ns_mgmt_entry_inq_if_ids(3rpc)**.

rpc_if_inq_id

Purpose

Returns the interface identifier for an interface specification; used by client or server applications

Synopsis

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

void rpc_if_inq_id(
        rpc_if_handle_t if_handle,
        rpc_if_id_t *if_id,
        unsigned32 *status);
```

Parameters

Input

if_handle Represents a stub-generated data structure that specifies the interface specification to inquire about.

Output

if_id Returns the interface identifier. The application provides memory for the returned data.

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 **rpc_if_inq_id()** routine to obtain a copy of the interface identifier from the provided interface specification.

The returned interface identifier consists of the interface UUID and interface version numbers (major and minor) specified in the DCE IDL file's interface specification.

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_if_id_vector_free(3rpc)**, **rpc_mgmt_inq_if_ids(3rpc)**, **rpc_ns_mgmt_entry_inq_if_ids(3rpc)**.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y