OSF® DCE Application Development Reference Release 1.2.2

December 8, 1998

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

OSF® DCE Application Development Reference

Release 1.2.2

Contents

Preface

The *OSF DCE 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 is organized into six sections.

- For DCE Routines, “Chapter 1. DCE Routines” on page 1.
- For DCE Threads, “Chapter 2. DCE Threads” on page 263.
- For DCE Remote Procedure Call,“Chapter 3. DCE Remote Procedure Call” on page 351.
- For DCE Directory Service,“Chapter 4. DCE Directory Service” on page 719.
- For DCE Distributed Time Service,“Chapter 5. DCE Distributed Time Service” on page 841.
- For DCE Security Service,“Chapter 6. DCE Security Service” on page 907.

Related Documents

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

- *Introduction to OSF DCE*
- *OSF DCE Administration Commands Reference*
- *OSF DCE Administration Guide—Introduction*
- *OSF DCE Administration Guide—Core Components*
- *OSF DCE DFS Administration Guide and Reference*
- *OSF DCE GDS Administration Guide and Reference*

- *OSF DCE Application Development Guide—Introduction and Style Guide*
- *OSF DCE Application Development Guide—Core Components*
- *OSF DCE Application Development Guide—Directory Services*
- *OSF DCE/File-Access Administration Guide and Reference*
- *OSF DCE/File-Access User's Guide*
- *OSF DCE Problem Determination Guide*
- *OSF DCE Testing Guide*
- *OSF DCE/File-Access FVT User's Guide*
- *Application Environment Specification/Distributed Computing*
- *OSF DCE Release Notes*

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 you 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 *OSF DCE Administration Guide—Introduction* and *OSF DCE Testing Guide*.

Problem Reporting

If you have any problems with the software or documentation, please contact your software vendor's customer service department.

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: *OSF DCE Application Development Guide—Core Components*, *OSF DCE Administration Commands 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 *OSF DCE 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: *OSF DCE 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: *OSF DCE 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: *OSF DCE 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 levels 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: *OSF DCE 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: *OSF DCE Application Development Guide*

DCE_SVC_INTRO

Purpose

Introduction to the DCE serviceability interface

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: *OSF DCE Application Development Guide*

dce_assert

Purpose

Inserts program diagnostics

Synopsis

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

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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_cursor_release(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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide—Core Components*, *OSF DCE Administration Commands 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>

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide—Core Components*, *OSF DCE Administration Commands 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide—Core Components*, *OSF DCE Administration Commands Reference*.

dce_cf_get_cell_name

Purpose

Returns the primary name for the local cell

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Administration Guide*.

dce_cf_prin_name_from_host

Purpose

Returns the host's principal name

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE Administration Guide*.

dce_cf_profile_entry_from_host

Purpose

Returns the host profile entry

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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.

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 *OSF DCE 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: *OSF DCE Application Development Guide—Core Components*, *OSF DCE Administration Commands Reference*.

dce_db_close

Purpose

Closes an open backing store

Synopsis

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

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 "dceappmsg.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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 "dceappmsg.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 *OSF DCE 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 *OSF DCE 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 "dceappmsg.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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dce_server_enable_service

Purpose

Enables an individual service for a server

Synopsis

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

void  dce_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 *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **dce_server_disable_service(3dce)**, **dce_server_register(3dce)**.

Books: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dce_server_inq_server

Purpose

Obtains the server configuration data dced used to start the server

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dce_server_register

Purpose

Registers a server with DCE

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE 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: *OSF DCE 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>

void  dce_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 *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **dce_server_sec_begin(3dce)**, **rpc_server_listen(3rpc)**.

Books: *OSF DCE Application Development Guide*.

dce_server_unregister

Purpose

Unregisters a DCE server

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dce_server_use_protseq

Purpose

Tells DCE to use the specified protocol sequence for receiving RPCs

Synopsis

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

void  dce_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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 levels 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.

Files

See **svcroute(5dce)**.

Errors

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

See **dce_svc_register(3dce)**.

Related Information

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

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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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>

void  dce_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 attributes

  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 *OSF DCE 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 tables 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dce_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 *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_binding_free

Purpose

Releases the resources associated with a dced binding handle

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_binding_set_auth_info

Purpose

Sets authentication and authorization information for a dced binding handle

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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_binding_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: *OSF DCE 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>

void  dced_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 following 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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_hostdata_read

Purpose

Reads a hostdata item maintained by dced on a specific host

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_hostdata_write

Purpose

Replaces an existing hostdata item maintained by dced on a specific host

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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_version_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: *OSF DCE 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>

void  dced_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. Management 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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_keytab_delete

Purpose

Deletes a key table file from a specific host

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_list_release

Purpose

Releases the resources for a list of entries of a dced service

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_object_read

Purpose

Reads a data item of a dced service on a specific host

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_objects_release

Purpose

Releases the resources allocated for data read from a dced service

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_secval_start

Purpose

Starts the security validation service of a specific host's dced

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_secval_stop

Purpose

Stops the security validation service of a specific host's dced

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_secval_validate

Purpose

Validates that the secd used by a specific host is legitimate

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE 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>

void  dced_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 *OSF DCE 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: *OSF DCE Application Development Guide*.

dced_server_modify_attributes

Purpose

Modifies attributes for a DCE server's configuration data

Synopsis

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

void  dced_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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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)**.

dced_server_start(3dce)

Functions: **dced_binding_create(3dce)**, **dced_binding_from_rpc_binding(3dce)**, **dced_server_create(3dce)**, **dced_server_stop(3dce)**.

Books: *OSF DCE 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 *OSF DCE 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)**, **dced_binding_from_rpc_binding(3dce)**, **dced_server_create(3dce) dced_server_start(3dce)**, **rpc_mgmt_stop_server_listening(3rpc)**.

Books: *OSF DCE 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 messages. 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

Table 1. 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.

Table 2. 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

Table 3. 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

Table 4. 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

Table 5. 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

Table 6. 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

Table 7. 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

Table 8. 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

Table 9. Cleanup Routines

Routine	Description
pthread_cleanup_pop()	Removes a cleanup handler from the stack
pthread_cleanup_push()	Establishes a cleanup handler

Table 10. The atfork() Routine

Routine	Description
atfork()	Arranges for fork cleanup handling

Table 11. 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 objects.

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
0		Successful completion.
−1	[**EINVAL**]	The value specified by a parameter is invalid.

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 *OSF DCE Administration Commands 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.

Table 12. Base Data Type Specifiers—rpc_intro(3rpc)

Specifier			Size	Type Macro Emitted by idl
(sign)	**(size)**	**(type)**		
	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**

Table 12. Base Data Type Specifiers—rpc_intro(3rpc) (continued)

Specifier			Size	Type Macro Emitted by idl
(sign)	(size)	(type)		
		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 *OSF DCE Administration Commands 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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.

Table 13. 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 sets 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 *OSF DCE 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 **ncad**).
- 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.

Table 14. 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.

Table 15. 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 *OSF DCE 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: *OSF DCE Application Development Guide—Introduction and Style Guide*, *OSF DCE Application Development Guide—Core Components*, *OSF DCE Application Development Guide—Directory Services*, *OSF DCE Administration Commands Reference*, *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 sets 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 *OSF DCE 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 sets 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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:

Table 16. 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 sets 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 *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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 *OSF DCE 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: *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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 *OSF DCE 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)**.

rpc_mgmt_ep_elt_inq_begin

Purpose

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

Synopsis

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

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

Parameters

Input

ep_binding

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

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

inquiry_type

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

Table 17. Valid Inquiries on Local Endpoint Maps

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

Table 17. Valid Inquiries on Local Endpoint Maps (continued)

Value	Description
rpc_c_ep_match_by_both	Searches the local endpoint map for those elements that contain the interface identifier and object UUID specified by the *if_id*, *vers_option*, and *object_uuid* parameters.

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

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

vers_option

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

The following table presents the valid values for this parameter:

Table 18. Valid values of vers_option

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

object_uuid

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

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

Output

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

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

Description

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

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_inquiry_context
Invalid inquiry context.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_ep_elt_inq_next(3rpc), **rpc_mgmt_ep_unregister(3rpc)**.

rpc_mgmt_ep_elt_inq_done

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

  Returns the value NULL.

Output

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

Description

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
: Success.

rpc_s_invalid_inquiry_context
: Invalid inquiry context.

Related Information

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

rpc_mgmt_ep_elt_inq_next

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

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

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

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

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

Specify NULL to prevent the routine from returning this parameter.

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

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

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

Description

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

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

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_comm_failure
Communications failure.

ept_s_database_invalid
Endpoint map database invalid.

rpc_s_fault_context_mismatch
Fault context mismatch.

ept_s_invalid_context
Invalid inquiry type for this context.

ept_s_invalid_entry
Invalid database entry.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_inquiry_context
Invalid inquiry context.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_no_more_elements
No more elements.

Related Information

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

rpc_mgmt_ep_unregister

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

binding
Specifies the binding handle to remove.

object_uuid
Specifies an optional object UUID to remove.

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

Output

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

Description

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

ept_s_cant_access
Error reading the endpoint database.

ept_s_cant_perform_op
Cannot perform the requested operation.

rpc_s_comm_failure
Communications failure.

ept_s_database_invalid
Endpoint map database is invalid.

ept_s_invalid_entry
Invalid database entry.

ept_s_not_registered
No entries found.

ept_s_update_failed
Update failed.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_no_interfaces
No interfaces registered.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_inq_com_timeout

Purpose

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

Synopsis

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

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

Parameters

Input

binding
Specifies a server binding handle.

Output

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

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

Description

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_inq_dflt_protect_level

Purpose

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

Synopsis

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

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

Parameters

Input

authn_svc

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

The supported authentication services are as follows:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

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

rpc_c_authn_default
DCE default authentication service.

Output

protect_level

Returns the default protection level for the specified authentication service. The protection level determines the degree to which authenticated communications between the client and the server are protected.

The possible protection levels are as follows:

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

rpc_c_protect_level_none
Performs no protection.

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

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

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

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

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

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

Description

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_unknown_authn_service
Unknown authentication service.

Related Information

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

rpc_mgmt_inq_if_ids

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

Output

if_id_vector
Returns the address of an interface identifier vector.

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

Description

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_no_interfaces
No interfaces registered.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_inq_server_princ_name

Purpose

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

Synopsis

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

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

Parameters

Input

binding

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

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

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

authn_svc

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

Output

server_princ_name

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

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

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

Description

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

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

rpc_s_comm_failure
Communications failure.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_unknown_authn_service
Unknown authentication service.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_inq_stats

Purpose

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

Synopsis

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

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

Parameters

Input

binding

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

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

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

Output

statistics

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

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

Description

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_is_server_listening

Purpose

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

Synopsis

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

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

Parameters

Input

binding

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

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

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

Output

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

Description

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

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

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

Return Values

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

Table 19. Values Returned by rpc_mgmt_is_server_listening()

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

Errors

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

rpc_s_ok
Success.

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

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_set_authorization_fn

Purpose

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

Synopsis

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

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

Parameters

Input

authorization_fn

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

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

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

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

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

Table 20. Operation Values Passed to Authorization Function

Called Remote Routine	*requested_mgmt_operation*Value
rpc_mgmt_inq_if_ids()	rpc_c_mgmt_inq_if_ids
rpc_mgmt_inq_server_princ_name()	rpc_c_mgmt_inq_princ_name
rpc_mgmt_inq_stats()	rpc_c_mgmt_inq_stats
rpc_mgmt_is_server_listening()	rpc_c_mgmt_is_server_listen
rpc_mgmt_stop_server_listening()	rpc_c_mgmt_stop_server_listen

Output

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

Description

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

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

Table 21. Default Controls for Remote Management Routines

Remote Routine	Default Authorization
rpc_mgmt_inq_if_ids()	Enabled
rpc_mgmt_inq_server_princ_name()	Enabled
rpc_mgmt_inq_stats()	Enabled
rpc_mgmt_is_server_listening()	Enabled
rpc_mgmt_stop_server_listening()	Disabled

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

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

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

Related Information

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

rpc_mgmt_set_cancel_timeout

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

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

Description

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

Related Information

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

rpc_mgmt_set_com_timeout

Purpose

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

Synopsis

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

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

Parameters

Input

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

timeout
Specifies a communications timeout value.

Output

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

Description

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

After the initial relationship is established, subsequent communications for the binding handle cannot revert to less than the default timeouts for the protocol service. This means that after setting a short initial timeout and establishing a connection, calls in progress are not timed out any sooner than the default.

Note:

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

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

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

Table 22. Predefined Time-Out Values

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_timeout
Invalid timeout value.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_mgmt_set_server_stack_size

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

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

Description

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

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

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

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

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_arg
Invalid argument.

rpc_s_not_supported
Not supported.

Return Values

No value is returned.

Related Information

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

rpc_mgmt_stats_vector_free

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Output

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

Description

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
: Success.

Related Information

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

rpc_mgmt_stop_server_listening

Purpose

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

Synopsis

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

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

Parameters

Input

binding

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

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

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

Output

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

Description

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

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

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

rpc_s_comm_failure
Communications failure.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_mgmt_op_disallowed
Management operation disallowed.

rpc_s_unknown_if
Unknown interface.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_network_inq_protseqs

Purpose

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

Synopsis

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

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

Parameters

Input

None.

Output

protseq_vector
Returns the address of a protocol sequence vector.

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

Description

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

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

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

rpc_network_is_protseq_valid

Purpose

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

Synopsis

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

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

Parameters

Input

protseq

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

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

Output

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

Description

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

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

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

Return Values

This routine can return the following values:

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

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

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

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

rpc_ns_binding_export

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

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

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

Output

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

Description

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

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

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

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

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

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

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

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

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_nothing_to_export
Nothing to export.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_ns_binding_import_begin

Purpose

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

Synopsis

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

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

Parameters

Input

entry_name_syntax

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

entry_name

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

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

if_handle

A stub-generated data structure specifying the interface to import. If the interface specification has not been exported or is of no concern to the caller, specify NULL for this parameter. In this case the bindings returned are only guaranteed to be of a compatible and supported protocol sequence and, depending on the value of parameter *obj_uuid*, contain the specified object Universal Unique Identifier (UUID). The desired interface may not be supported by the contacted server.

obj_uuid

Specifies an optional object UUID.

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

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

Output

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

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

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

The possible status codes and their meanings are as follows:

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_object
Invalid object.

rpc_s_no_env_setup
Environment variable not set up.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Description

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

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

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

Return Values

No value is returned.

Related Information

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

rpc_ns_binding_import_done

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Returns the value NULL.

Output

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

Description

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

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

rpc_ns_binding_import_next

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

binding
Returns a compatible server binding handle.

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

Description

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

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

Note:

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

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

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

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

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

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

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

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

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

rpc_ns_binding_inq_entry_name

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

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

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

Description

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

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

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_binding
Invalid binding handle.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_no_entry_name
No entry name for binding.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

Related Information

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

rpc_ns_binding_lookup_begin

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

object_uuid
: Specifies an optional object UUID.

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

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

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

Output

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

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

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

Description

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_object
Invalid object.

rpc_s_no_env_setup
Environment variable not set up.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_binding_lookup_done

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Returns the value NULL.

Output

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

Description

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

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

rpc_ns_binding_lookup_next

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

binding_vec
Returns a vector of compatible server binding handles.

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

Description

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

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

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

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

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

- The search operation returns members of a profile with the same priority in random order.

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

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

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

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

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

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

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

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

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

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_bindings
No more bindings.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

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

rpc_ns_binding_select

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Output

binding
: Returns a selected server binding handle.

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

Description

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

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

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_no_more_bindings
No more bindings.

Related Information

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

rpc_ns_binding_unexport

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

Output

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

Description

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

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

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

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

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

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

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_interface_not_found
Interface not found.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_nothing_to_unexport
Nothing to unexport.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_entry_expand_name

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

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

Description

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

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

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

```
/.:/subsys/PrintQ/server1
```

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

Related Information

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

Books: *OSF DCE Administration Guide—Introduction*.

rpc_ns_entry_inq_resolution

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Input/Output

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

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

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

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

Output

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

Description

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

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

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

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

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

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

Return Values

None.

Errors

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

rpc_s_ok
: Success.

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

rpc_s_invalid_name_syntax
: The requested name syntax is invalid.

rpc_s_unsupported_name_syntax
: The requested name syntax is not supported.

Related Information

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

rpc_ns_entry_object_inq_begin

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

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

Description

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_entry_object_inq_done

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Returns the value NULL.

Output

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

Description

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

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

rpc_ns_entry_object_inq_next

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

obj_uuid
Returns an exported object UUID.

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

Description

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

An application can view all of the exported object UUIDs by repeatedly calling the **rpc_ns_entry_object_inq_next()** routine. When all the object UUIDs are viewed, this routine returns an **rpc_s_no_more_members** status. The returned object UUIDs are unordered.

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

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

rpc_ns_group_delete

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

Description

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_group_mbr_add

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

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

Output

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

Description

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_group_mbr_inq_begin

Purpose

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

Synopsis

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

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

Parameters

Input

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

group_name
Specifies the name of the RPC group to view.

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

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

Output

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

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

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

Description

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_group_mbr_inq_done

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Returns the value NULL.

Output

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

Description

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

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

rpc_ns_group_mbr_inq_next

Purpose

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

Synopsis

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

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

Parameters

Input

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

Output

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

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

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

Description

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_members
No more members.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

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

rpc_ns_group_mbr_remove

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

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

Output

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

Description

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_group_member_not_found
Group member not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_import_ctx_add_eval

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

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

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

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

Output

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

- **rpc_ns_binding_import_next()**

- **rpc_ns_binding_import_done()**

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

Description

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

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

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

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

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

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

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

Restrictions

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_no_memory
The RPC runtime could not allocate heap storage.

rpc_s_invalid_ns_handle
The *import_context* parameter was not valid.

Related Information

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

rpc_ns_mgmt_binding_unexport

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

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

The following table presents the accepted values for this parameter:

Table 23. Uses of vers_major and vers_minor fields of if_id

Value	Description
rpc_c_vers_all	Unexports (removes) all bindings for the interface UUID in *if_id*, regardless of the version numbers. For this value, specify 0 (zero) for both the major and minor versions in *if_id*.
rpc_c_vers_compatible	Removes those bindings for the interface UUID in *if_id* with the same major version as in *if_id*, and with a minor version greater than or equal to the minor version in *if_id*.
rpc_c_vers_exact	Removes those bindings for the interface UUID in *if_id* with the same major and minor versions as in *if_id*.

Table 23. Uses of vers_major and vers_minor fields of if_id (continued)

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

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

Output

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

Description

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

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

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

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

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

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

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_interface_not_found
Interface not found.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_all_objs_unexported
Not all objects unexported.

rpc_s_nothing_to_unexport
Nothing to unexport.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_mgmt_entry_create

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

Description

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_already_exists
Name service entry already exists.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_mgmt_entry_delete

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

Description

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

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

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

- Performance problems

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

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

  In this situation, if one application deletes the entry and another application updates the entry before the replicas are synchronized, the delete operation

takes precedence over the update operation. When the replicas are synchronized, the update is lost because the entry is deleted from all replicas.

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_mgmt_entry_inq_if_ids

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

if_id_vec
: Returns the address of the interface identifier vector.

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

Description

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

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

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_interfaces_exported
No interfaces were exported to entry.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

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

rpc_ns_mgmt_free_codesets

Purpose

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

Synopsis

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

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

Parameters

Input/Output

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

Output

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

Description

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

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

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

Permissions Required

None.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

Related Information

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

rpc_ns_mgmt_handle_set_exp_age

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

Output

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

Description

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

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

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

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

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

Permissions Required

No permissions are required.

Cautions

Use this routine with extreme caution.

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

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

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

  For example, during management operations to update a profile, you may need to always see the profile's current contents. In this case, before beginning to inquire about a profile, your application must call **rpc_ns_mgmt_handle_set_exp_age()** and specify 0 (zero) for the *expiration_age* parameter.
- When a request using the default expiration age fails, and your application needs to retry the operation.

  For example, a client application using import must first try to obtain bindings using the application's default expiration age. However, sometimes the import-next operation returns either no binding handles or an insufficient number of them. In this case, the client can retry the import operation and, after **rpc_ns_binding_import_begin()** terminates, include a

rpc_ns_mgmt_handle_set_exp_age() routine that specifies 0 (zero) for the *expiration_age* parameter. When the client calls the import-next routine again, the small expiration age for the name service handle causes the import-next operation to update the local attribute data.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

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

rpc_ns_mgmt_inq_exp_age

Purpose

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

Synopsis

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

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

Parameters

Input

None.

Output

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

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

Description

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

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

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

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

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

Related Information

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

rpc_ns_mgmt_read_codesets

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

Output

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

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

Description

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

Client applications use the **rpc_ns_mgmt_read_codesets()** routine to retrieve a server's supported code sets in order to evaluate them against the code sets that the client supports. Client applications that use the evaluation routines **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()** do not need to call this routine explicitly, because these code sets evaluation routines call it on

the client's behalf. Application developers who are writing their own character and code set evaluation routines may need to include **rpc_ns_mgmt_read_codesets()** in their user-written evaluation routines.

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok

rpc_s_invalid_name_syntax

rpc_s_mgmt_bad_type

rpc_s_name_service_unavailable

rpc_s_no_permission

rpc_s_incomplete_name

rpc_s_no_memory

Related Information

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

rpc_ns_mgmt_remove_attribute

Purpose

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

Synopsis

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

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

Parameters

Input

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

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

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

Output

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

Description

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

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

For DCE 1.2, you use **rpc_ns_mgmt_remove_attribute()** in your application server initialization routine or signal handling routine to remove a code sets attribute from

the server's entry in the Cell Directory Service database as part of the server cleanup procedure carried out prior to the server's termination.

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

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

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

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

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

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

Related Information

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

rpc_ns_mgmt_set_attribute

Purpose

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

Synopsis

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

void rpc_ns_mgmt_set_attribute(
   unsigned32 entry_name_syntax
   unsigned_char_t *entry_name
   uuid_t *attr_type
   void *attr_value
   error_status_t *status);
```

Parameters

Input

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

entry_name
Specifies the name of the RPC server entry in the name service database with which the attribute will be associated. The name can be either the global or cell-relative name. If you are using this routine to add a code sets attribute to an RPC server entry in the name service database, then this parameter specifies the name of the server entry with which the code sets attribute will be associated.

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

attr_val
An opaque data structure that specifies the attribute value to be stored in the name service database. If you are using this routine to add a code sets attribute to an RPC server entry, you must cast the representation of the code set data from the data type **rpc_codeset_mgmt_p_t** to the data type **void***.

Output

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

Description

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

The **rpc_ns_mgmt_set_attribute()** routine is designed to be a generic routine for adding an attribute to an RPC server entry in the name service database. The routine takes an attribute type and a pointer to the value, and stores the attribute value in the name service database.

For DCE 1.2, you use **rpc_ns_mgmt_set_attribute()** in your application server initialization routine to add a code sets attribute to the server's entry in the Cell Directory Service database (which the initialization routine has created with the **rpc_ns_binding_export()** routine). Because CDS stores integer values in little-endian format, the **rpc_ns_mgmt_set_attribute()** routine also encodes the code sets attribute value into an endian-safe format before storing it in the name service database.

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

Permissions Required

You need both read permission and write permission to the target RPC server entry (which is a CDS object). You also need insert permission to the parent directory.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

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

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

rpc_s_no_memory
The routine was unable to allocate memory to encode the value.

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

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

Related Information

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

rpc_ns_mgmt_set_exp_age

Purpose

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

Synopsis

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

void rpc_ns_mgmt_set_exp_age(
   unsigned32 expiration_age
   unsigned32 *status);
```

Parameters

Input

expiration_age
An integer value that specifies the default expiration age, in seconds, for local name service data. This expiration age applies to all RPC name service interface (NSI) read operations (all the next operations).

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

To reset the expiration age to an RPC-assigned random value between 8 and 12 hours, specify a value of **rpc_c_ns_default_exp_age**.

Output

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

Description

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

Normally, you should avoid using this routine; instead, rely on the default expiration age.

The RPC NSI next operations, which read data from name service attributes, use an expiration age. A next operation normally starts by looking for a local copy of the attribute data that an application requests. In the absence of a local copy, the next operation creates one with fresh attribute data from the name service database. If a local copy already exists, the operation compares its actual age to the expiration age being used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data from

the name service database. If updating is impossible, the old local data remains in place and the next operation fails, returning the **rpc_s_name_service_unavailable** status code.

Permissions Required

No permissions are required.

Cautions

Use this routine with extreme caution.

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

Related Information

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

rpc_ns_profile_delete

Purpose

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

Synopsis

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

void rpc_ns_profile_delete(
   unsigned32 profile_name_syntax
   unsigned_char_t *profile_name
   unsigned32 *status);
```

Parameters

Input

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

profile_name
Specifies the name of the profile to delete. This can be either the global or cell-relative name.

Output

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

Description

The **rpc_ns_profile_delete()** routine deletes the profile attribute from the specified entry in the name service database (the *profile_name* parameter).

Neither the specified entry nor the entry names included as members in each profile element are deleted.

Use this routine cautiously; deleting a profile may break a hierarchy of profiles.

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_profile_elt_add(3rpc)**, **rpc_ns_profile_elt_remove(3rpc)**.

rpc_ns_profile_elt_add

Purpose

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

Synopsis

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

void  rpc_ns_profile_elt_add(
   unsigned32 profile_name_syntax
   unsigned_char_t *profile_name
   rpc_if_id_t *if_id
   unsigned32 member_name_syntax
   unsigned_char_t *member_name
   unsigned32 priority
   unsigned_char_t *annotation
   unsigned32 *status);
```

Parameters

Input

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

profile_name
Specifies the RPC profile that receives a new element. This can be either the global or cell-relative name.

if_id Specifies the interface identifier of the new profile element. To add or replace the default profile element, specify NULL.

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

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

member_name
Specifies the entry in the name service database to include in the new profile element. This can be either the global or cell-relative name.

priority An integer value (0 to 7) that specifies the relative priority for using the new profile element during the import and lookup operations. A value of 0 (zero) is the highest priority. A value of 7 is the lowest priority. Two or more elements can have the same priority.

When adding the default profile member, use a value of 0 (zero).

annotation
Specifies an annotation string that is stored as part of the new profile element. The string can be up to 17 characters long. Specify NULL or the string **\0** if there is no annotation string.

The string is used by applications for informational purposes only. For example, an application can use this string to store the interface name string (specified in the IDL file).

DCE RPC does not use this string during lookup or import operations, or for enumerating profile elements.

Output

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

Description

The **rpc_ns_profile_elt_add()** routine adds an element to the profile attribute of the entry in the name service database specified by the *profile_name* parameter.

If the *profile_name* entry does not exist, this routine creates the entry with a profile attribute and adds the profile element specified by the *if_id*, *member_name*, *priority*, and *annotation* parameters. In this case, the application must have permission to create the entry. Otherwise, a management application with the necessary permissions creates the entry by calling **rpc_ns_mgmt_entry_create()** before the application is run.

If an element with the specified member name and interface identifier are already in the profile, this routine updates the element's priority and annotation string using the values provided in the *priority* and *annotation* parameters.

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

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_priority
Invalid profile element priority.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_if_inq_id(3rpc)**, **rpc_ns_mgmt_entry_create(3rpc)**, **rpc_ns_profile_elt_remove(3rpc)**.

rpc_ns_profile_elt_inq_begin

Purpose

Creates an inquiry context for viewing the elements in a profile; used by client, server, or management applications

Synopsis

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

void rpc_ns_profile_elt_inq_begin(
   unsigned32 profile_name_syntax
   unsigned_char_t *profile_name
   unsigned32 inquiry_type
   rpc_if_id_t *if_id
   unsigned32 vers_option
   unsigned32 member_name_syntax
   unsigned_char_t *member_name
   rpc_ns_handle_t *inquiry_context
   unsigned32 *status);
```

Parameters

Input

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

profile_name
Specifies the name of the profile to view. This can be either the global or cell-relative name.

inquiry_type
An integer value that specifies the type of inquiry to perform on the profile. The following table describes the valid inquiry types:

Table 24. Valid Values of inquiry_ type

Value	Description
rpc_c_profile_default_elt	Searches the profile for the default profile element, if any. The *if_id*, *vers_option*, and *member_name* parameters are ignored.
rpc_c_profile_all_elts	Returns every element from the profile. The *if_id*, *vers_option*, and *member_name* parameters are ignored.
rpc_c_profile_match_by_if	Searches the profile for those elements that contain the interface identifier specified by the *if_id* and *vers_option* values. The *member_name* parameter is ignored.

Table 24. Valid Values of inquiry_ type (continued)

Value	Description
rpc_c_profile_match_by_mbr	Searches the profile for those elements that contain the member name specified by the *member_name* parameter. The *if_id* and *vers_option* parameters are ignored.
rpc_c_profile_match_by_both	Searches the profile for those elements that contain the interface identifier and member name specified by the *if_id*, *vers_option*, and *member_name* parameters.

if_id Specifies the interface identifier of the profile elements to be returned by **rpc_ns_profile_elt_inq_next()**.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_if** or **rpc_c_profile_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you can specify the value NULL.

vers_option
Specifies how **rpc_ns_profile_elt_inq_next()** uses the *if_id* parameter.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_if** or **rpc_c_profile_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you can specify the value 0 (zero).

The following table describes the valid values for this parameter:

Table 25. Valid Values of vers_option

Value	Description
rpc_c_vers_all	Returns profile elements that offer the specified interface UUID, regardless of the version numbers. For this value, specify 0 (zero) for both the major and minor versions in *if_id*.
rpc_c_vers_compatible	Returns profile elements that offer the same major version of the specified interface UUID and a minor version greater than or equal to the minor version of the specified interface UUID.
rpc_c_vers_exact	Returns profile elements that offer the specified version of the specified interface UUID.
rpc_c_vers_major_only	Returns profile elements that offer the same major version of the specified interface UUID (ignores the minor version). For this value, specify 0 (zero) for the minor version in *if_id*.

Table 25. Valid Values of vers_option (continued)

Value	Description
rpc_c_vers_upto	Returns profile elements that offer a version of the specified interface UUID less than or equal to the specified major and minor version. (For example, if *if_id* contains V2.0 and the profile contains elements with the versions V1.3, V2.0, and V2.1, **rpc_ns_profile_elt_inq_next()** returns the elements with V1.3 and V2.0.)

member_name_syntax
An integer value that specifies the syntax of the *member_name* parameter in this routine and the syntax of the *member_name* parametr in **rpc_ns_profile_elt_inq_next()**. To use the syntax that is specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Specifies the member name that **rpc_ns_profile_elt_inq_next()** looks for in profile elements. This can be either the global or cell-relative name.

This parameter is used only when specifying a value of either **rpc_c_profile_match_by_mbr** or **rpc_c_profile_match_by_both** for the *inquiry_type* parameter. Otherwise, this parameter is ignored and you specify the value NULL.

Output

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

- **rpc_ns_profile_elt_inq_next()**
- **rpc_ns_profile_elt_inq_done()**

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

Description

The **rpc_ns_profile_elt_inq_begin()** routine creates an inquiry context for viewing the elements in a profile.

Using the *inquiry_type* and *vers_option* parameters, an application specifies which of the following profile elements will be returned from calls to **rpc_ns_profile_elt_inq_next()**:

- The default element.
- All elements.
- Those elements with the specified interface identifier.
- Those elements with the specified member name.
- Those elements with both the specified interface identifier and member name.

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

When finished viewing the profile elements, the application calls the **rpc_ns_profile_elt_inq_done()** routine to delete the inquiry context.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_inquiry_type
Invalid inquiry type.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_invalid_vers_option
Invalid version option.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_if_inq_id(3rpc)**, **rpc_ns_mgmt_handle_set_exp_age(3rpc)**, **rpc_ns_profile_elt_inq_done(3rpc)**, **rpc_ns_profile_elt_inq_next(3rpc)**.

rpc_ns_profile_elt_inq_done

Purpose

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

Synopsis

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

void rpc_ns_profile_elt_inq_done(
   rpc_ns_handle_t *inquiry_context
   unsigned32 *status);
```

Parameters

Input/Output

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

Returns the value NULL.

Output

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

Description

The **rpc_ns_profile_elt_inq_done()** routine deletes an inquiry context created by calling **rpc_ns_profile_elt_inq_begin()**.

An application calls this routine after viewing profile elements using the **rpc_ns_profile_elt_inq_next()** routine.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_invalid_ns_handle
Invalid name service handle.

Related Information

Functions: **rpc_ns_profile_elt_inq_begin(3rpc)**, **rpc_ns_profile_elt_inq_next(3rpc)**.

rpc_ns_profile_elt_inq_next

Purpose

Returns one element at a time from a profile; used by client, server, or management applications

Synopsis

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

void rpc_ns_profile_elt_inq_next(
   rpc_ns_handle_t inquiry_context
   rpc_if_id_t *if_id
   unsigned_char_t **member_name
   unsigned32 *priority
   unsigned_char_t **annotation
   unsigned32 *status);
```

Parameters

Input

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

Output

if_id Returns the interface identifier of the profile element.

member_name
Returns a pointer to the profile element's member name. The name is a global name.

The syntax of the returned name is specified by the **rpc_ns_profile_elt_inq_begin()** *member_name_syntax* parameter.

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

priority Returns the profile element priority.

annotation
Returns the annotation string for the profile element. If there is no annotation string in the profile element, the string **\0** is returned.

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

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

Description

The **rpc_ns_profile_elt_inq_next()** routine returns one element from the profile specified by the *profile_name* parameter in the **rpc_ns_profile_elt_inq_begin()** routine.

The selection criteria for the element returned are based on the *inquiry_type* parameter in the **rpc_ns_profile_elt_inq_begin()** routine. The **rpc_ns_profile_elt_inq_next()** routine returns all the components (interface identifier, member name, priority, annotation string) of a profile element.

An application can view all the selected profile entries by repeatedly calling the **rpc_ns_profile_elt_inq_next()** routine. When all the elements have been viewed, this routine returns an **rpc_s_no_more_elements** status code. The returned elements are unordered.

On each call to this routine that returns a profile element, the DCE RPC runtime allocates memory for the returned *member_name* (which points to a global name) and *annotation* strings. The application is responsible for calling the **rpc_string_free()** routine for each returned *member_name* and *annotation* string.

After viewing the profile's elements, the application must call the **rpc_ns_profile_elt_inq_done()** routine to delete the inquiry context.

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_class_version_mismatch
RPC class version mismatch.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_ns_handle
Invalid name service handle.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_more_elements
No more elements.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_not_rpc_entry
Not an RPC entry.

Related Information

Functions: **rpc_ns_profile_elt_begin(3rpc)**, **rpc_ns_profile_elt_done(3rpc)**, **rpc_string_free(3rpc)**.

rpc_ns_profile_elt_remove

Purpose

Removes an element from a profile; used by client, server, or management applications

Synopsis

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

void rpc_ns_profile_elt_remove(
   unsigned32 profile_name_syntax
   unsigned_char_t *profile_name
   rpc_if_id_t *if_id
   unsigned32 member_name_syntax
   unsigned_char_t *member_name
   unsigned32 *status);
```

Parameters

Input

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

profile_name
Specifies the profile from which to remove an element. This can be either the global or cell-relative name.

if_id Specifies the interface identifier of the profile element to be removed. Specify NULL to remove the default profile member.

member_name_syntax
An integer value that specifies the syntax of *member_name*. To use the syntax specified in the **RPC_DEFAULT_ENTRY_SYNTAX** environment variable, provide **rpc_c_ns_syntax_default**.

member_name
Specifies the name service entry name in the profile element to remove. This can be either the global or cell-relative name. When *if_id* is NULL, this argument is ignored.

Output

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

Description

The **rpc_ns_profile_elt_remove()** routine removes a profile element from the profile specified by *profile_name*. Unless *if_id* is NULL, the *member_name* parameter and the *if_id* parameter must match the corresponding profile element attributes exactly for an element to be removed. When *if_id* is NULL, the default profile element is removed, and the *member_name* argument is ignored.

The routine removes the reference to the entry specified by *member_name* from the profile; it does not delete the entry itself.

Use this routine cautiously; removing elements from a profile may break a hierarchy of profiles.

Permissions Required

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

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_entry_not_found
Name service entry not found.

rpc_s_incomplete_name
Incomplete name.

rpc_s_invalid_name_syntax
Invalid name syntax.

rpc_s_name_service_unavailable
Name service unavailable.

rpc_s_no_ns_permission
No permission for name service operation.

rpc_s_profile_element_not_found
Profile element not found.

rpc_s_unsupported_name_syntax
Unsupported name syntax.

Related Information

Functions: **rpc_ns_profile_delete(3rpc)**, **rpc_ns_profile_elt_add(3rpc)**.

rpc_object_inq_type

Purpose

Returns the type of an object; used by server applications

Synopsis

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

void rpc_object_inq_type(
   uuid_t *obj_uuid
   uuid_t *type_uuid
   unsigned32 *status);
```

Parameters

Input

obj_uuid
: Specifies the object UUID whose associated type UUID is returned. Supply NULL to specify a nil UUID for this parameter.

Output

type_uuid
: Returns the type UUID corresponding to the object UUID supplied in the *obj_uuid* parameter.

  Specifying NULL here prevents the return of a type UUID. An application, by specifying NULL here, can determine from the value returned in *status* whether *obj_uuid* is registered. This determination occurs without the application specifying an output type UUID variable.

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

Description

A server application calls the **rpc_object_inq_type()** routine to obtain the type UUID of an object.

If the object is registered with the RPC runtime using the **rpc_object_set_type()** routine, the registered type is returned.

Optionally, an application can maintain an object/type registration privately. In this case, if the application provides an object inquiry function (see the **rpc_object_set_inq_fn(3rpc)** reference page), the RPC runtime uses that function to determine an object's type.

The table below shows how **rpc_object_inq_type()** obtains the returned type UUID.

Table 26. Rules for Returning an Object's Type

Was object UUID registered (using rpc_object_set_type)?	Was an object inquiry runction registered (using rpc_object_set_inq_fn)?	Return Value
Yes	Ignored	Returns the object's registered type UUID.
No	Yes	Returns the type UUID returned from calling the inquiry function.
No	No	Returns the nil UUID.

Return Values

No value is returned.

Errors

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

rpc_s_ok
Success.

rpc_s_object_not_found
Object not found.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_object_set_inq_fn(3rpc)**, **rpc_object_set_type(3rpc)**.

rpc_object_set_inq_fn

Purpose

Registers an object inquiry function; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_object_set_inq_fn(
   rpc_object_inq_fn_t inquiry_fn
   unsigned32 *status);
```

Parameters

Input

inquiry_fn

Specifies a pointer to an object type inquiry function. When an application calls the **rpc_object_inq_type()** routine and the RPC runtime finds that the specified object is not registered, the runtime automatically calls the **rpc_object_inq_type()** routine to determine the object's type. Specify NULL to remove a previously set inquiry function.

The following C language definition for **rpc_object_inq_fn_t** illustrates the prototype for this function:

```
typedef void (*rpc_object_inq_fn_t)
(
  uuid_t    *object_uuid, /* in */
  uuid_t    *type_uuid,  /* out */
  unsigned32  *status    /* out */
);
```

The returned *type_uuid* and *status* values are returned as the output arguments from the **rpc_object_inq_type()** routine.

If you specify NULL, the **rpc_object_set_inq_fn()** routine unregisters (that is, removes) a previously registered object type inquiry function.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

A server application calls **rpc_object_set_inq_fn()** to specify a function to determine an object's type. If an application privately maintains object/type registrations, the specified inquiry function returns the type UUID of an object from that registration.

The RPC runtime automatically calls the inquiry function when the application calls **rpc_object_inq_type()** and the object was not previously registered by **rpc_object_set_type()**. The RPC runtime also automatically calls the inquiry function for every remote procedure call it receives if the object was not previously registered.

Cautions

Use this routine with caution. When the RPC runtime automatically calls this routine in response to a received remote procedure call, the inquiry function can be called from the context of runtime internal threads with runtime internal locks held. The inquiry function should not block or at least not block for long (for example, the inquiry function should not perform a remote procedure call). Also, the inquiry function must not unwind because of an exception. In general, the inquiry function should not call back into the RPC runtime. It is legal to call **rpc_object_set_type()** or any of the **uuid**_ * routines. Failure to comply with these restrictions will result in undefined behavior.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_object_inq_type(3rpc)**, **rpc_object_set_type(3rpc)**.

rpc_object_set_type

Purpose

Registers the type of an object with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_object_set_type(
   uuid_t *obj_uuid
   uuid_t *type_uuid
   unsigned32 *status);
```

Parameters

Input

obj_uuid
: Specifies an object UUID to associate with the type UUID in the *type_uuid* parameter. Do not specify NULL or a nil UUID.

type_uuid
: Specifies the type UUID of the *obj_uuid* parameter.

  Specify an argument value of NULL or a nil UUID to reset the object type to the default association of object UUID/nil type UUID.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_object_set_type()** routine assigns a type UUID to an object UUID.

By default, the RPC runtime assumes that the type of all objects is nil. A server program that contains one implementation of an interface (one manager entry point vector) does not need to call this routine, provided that the server registered the interface with the nil type UUID (see the **rpc_server_register_if(3rpc)** reference page).

A server program that contains multiple implementations of an interface (multiple manager entry point vectors; that is, multiple type UUIDs) calls this routine once for each object UUID the server offers. Associating each object with a type UUID tells the RPC runtime which manager entry point vector (interface implementation) to use when the server receives a remote procedure call for a nonnil object UUID.

The RPC runtime allows an application to set the type for an unlimited number of objects.

To remove the association between an object UUID and its type UUID (established by calling this routine), a server calls this routine again and specifies the value NULL or a nil UUID for the *type_uuid* parameter. This resets the association between an object UUID and type UUID to the default.

A server cannot register a nil object UUID. The RPC runtime automatically registers the nil object UUID with a nil type UUID. Attempting to set the type of a nil object UUID will result in the routine's returning the status code **rpc_s_invalid_object**.

Servers that want to maintain their own object UUID to type UUID mapping can use **rpc_object_set_inq_fn()** in place of, or in addition to, **rpc_object_set_type()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_already_registered
Object already registered.

rpc_s_invalid_object
Invalid object.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_object_set_inq_fn(3rpc)**, **rpc_server_register_if(3rpc)**.

rpc_protseq_vector_free

Purpose

Frees the memory used by a vector and its protocol sequences; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_protseq_vector_free(
   rpc_protseq_vector_t **protseq_vector
   unsigned32 *status);
```

Parameters

Input/Output

protseq_vector
: Specifies the address of a pointer to a vector of protocol sequences. On return the pointer is set to NULL.

Output

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_protseq_vector_free()** routine frees the memory used to store a vector of protocol sequences. The freed memory includes both the protocol sequences and the vector itself.

Call **rpc_network_inq_protseqs()** to obtain a vector of protocol sequences. Follow a call to **rpc_network_inq_protseqs()** with a call to **rpc_protseq_vector_free()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
: Success.

Related Information

Functions: **rpc_network_inq_protseqs(3rpc)**.

rpc_rgy_get_codesets

Purpose

Gets supported code sets information from the local host; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_rgy_get_codesets(
   rpc_codeset_mgmt_p_t *code_sets_array
   error_status_t *status);
```

Parameters

Input

No input is required.

Output

code_sets_array
: An integer array that specifies the code sets that the client's or server's host environment supports. Each array element is an integer value that uniquely identifies one code set.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_rgy_get_codesets()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_rgy_get_codesets()** routine examines the locale environment of the host on which the client or server process is running to determine the local code set currently in use by the client or server process and the set of supported code set conversion routines that exist on the host into which the client or server process can convert if necessary. It then reads the code sets registry on the local host to retrieve the unique identifiers associated with these supported code sets.

The routine returns a code sets array. The set of values returned in this structure correspond to the process's local code set and the code sets into which processes that run on this host can convert. The array also contains, for each code set, the maximum number of bytes that code set uses to encode one character (*c_max_bytes*).

Server applications use the **rpc_rgy_get_codesets()** routine in their initialization code to get their host's supported character and code sets values in order to export them into the name service database with **rpc_ns_mgmt_set_attribute()**.

Client applications use the **rpc_rgy_get_codesets()** routine during the server binding selection process to retrieve the supported character and code sets at their

host in order to evaluate them against the character and code sets that a server supports. Client applications that use the evaluation routines **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()** do not need to call this routine explicitly, because these code sets evaluation routines call it on the client's behalf. Application developers who are writing their own character and code set evaluation routines may need to include **rpc_rgy_get_codesets()** in their user-written evaluation routines.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

rpc_s_ok

rpc_s_no_memory

Related Information

Commands: **csrc(8dce)**.

Functions: **rpc_ns_mgmt_read_codesets(3rpc)**, **rpc_ns_mgmt_remove_attribute(3rpc)**, **rpc_ns_mgmt_set_attribute(3rpc)**.

rpc_rgy_get_max_bytes

Purpose

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host; used by client and server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_rgy_get_max_bytes(
   unsigned32 rgy_code_set_value
   unsigned16 *rgy_max_bytes
   error_status_t *status);
```

Parameters

Input

rgy_code_set_value
The registered hexadecimal value that uniquely identifies the code set.

Output

rgy_max_bytes
The registered decimal value that indicates the number of bytes this code set uses to encode one character.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_rgy_get_max_bytes()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **rpc_rgy_get_max_bytes()** routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

The DCE RPC stub support routines for buffer sizing use the **rpc_rgy_get_max_bytes()** routine as part of their procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets. The DCE RPC stub support routines call the **rpc_rgy_get_max_bytes()** routine once to get the *rgy_max_bytes* value for the code set to be used to transfer the data over the network (the network code set) then call the routine again to get the *rgy_max_bytes* value of their local code set. The stubs then compare the two values to determine whether or not additional buffers are necessary or whether the conversion can be done in place.

Client and server applications that use the following DCE RPC buffer sizing routines do not need to call this routine explicitly because these DCE RPC stub support routines call it on their behalf:

- **byte_net_size()**

- **byte_local_size()**
- **wchar_t_net_size()**
- **wchar_t_local_size()**

Application programmers who are developing their own stub support routines for buffer sizing can use the **rpc_rgy_get_max_bytes()** routine in their code to get code set *max_byte* information for their user-written buffer sizing routines.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

dce_cs_c_cannot_open_file

dce_cs_c_cannot_read_file

dce_cs_c_notfound

dce_cs_c_unknown

rpc_s_ok

Related Information

Commands: **csrc(8dce)**.

Functions: **dce_cs_loc_to_rgy(3rpc)**, **dce_cs_rgy_to_loc(3rpc)**, **rpc_ns_mgmt_read_code_sets(3rpc)**, **rpc_rgy_get_code_sets(3rpc)**.

rpc_server_inq_bindings

Purpose

Returns binding handles for communications with a server; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_inq_bindings(
   rpc_binding_vector_t **binding_vector
   unsigned32 *status);
```

Parameters

Input

None.

Output

binding_vector
Returns the address of a vector of server binding handles.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_inq_bindings()** routine obtains a vector of server binding handles. Binding handles are created by the RPC runtime when a server application calls any of the following routines to register protocol sequences:

- **rpc_server_use_all_protseqs()**
- **rpc_server_use_all_protseqs_if()**
- **rpc_server_use_protseq()**
- **rpc_server_use_protseq_ep()**
- **rpc_server_use_protseq_if()**

The returned binding vector can contain binding handles with dynamic endpoints and binding handles with well-known endpoints, depending on which of the preceding routines the server application called. The **rpc_intro(3rpc)** reference page contains an explanation of dynamic and well-known endpoints.

A server uses the vector of binding handles for exporting to the name service, for registering with the local endpoint map, or for conversion to string bindings.

If there are no binding handles (no registered protocol sequences), this routine returns the **rpc_s_no_bindings** status code and returns the value NULL to the *binding_vector* parameter.

The server is responsible for calling the **rpc_binding_vector_free()** routine to deallocate the memory used by the vector.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_no_bindings
No bindings.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_inq_if

Purpose

Returns the manager entry point vector registered for an interface; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_inq_if(
   rpc_if_handle_t if_handle
   uuid_t *mgr_type_uuid
   rpc_mgr_epv_t *mgr_epv
   unsigned32 *status);
```

Parameters

Input

if_handle
: Specifies the interface specification whose manager entry point vector (EPV) pointer is returned in the *mgr_epv* parameter.

mgr_type_uuid
: Specifies a type UUID for the manager whose EPV pointer is returned in the *mgr_epv* parameter.

  Specifying the value NULL (or a nil UUID) has this routine return a pointer to the manager EPV that is registered with *if_handle* and the nil type UUID of the manager.

Output

mgr_epv
: Returns a pointer to the manager EPV corresponding to *if_handle* and *mgr_type_uuid*.

status
: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

A server application calls the **rpc_server_inq_if()** routine to determine the manager EPV for a registered interface and type UUID of the manager.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

Related Information

Functions: **rpc_server_register_if(3rpc)**.

rpc_server_listen

Purpose

Tells the RPC runtime to listen for remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_listen(
   unsigned32 max_calls_exec
   unsigned32 *status);
```

Parameters

Input

max_calls_exec
Specifies the maximum number of concurrent executing remote procedure calls.

Use the value **rpc_c_listen_max_calls_default** to specify the default value.

Also, the five **rpc_server_use_ *_protseq *()** routines limit (according to their *max_call_requests* parameter) the number of concurrent remote procedure call requests that a server can accept.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_listen()** routine makes a server listen for remote procedure calls. DCE RPC allows a server to simultaneously process multiple calls. The *max_calls_exec* parameter specifies the maximum number of concurrent remote procedure calls the server executes. Each remote procedure call executes in a call execution thread. The implementation of the RPC architecture determines whether it reuses call execution threads for the execution of subsequent remote procedure calls or, instead, it creates a new thread for each execution of a subsequent remote procedure call.

The following conditions affect the number of concurrent remote procedure calls that a server can process:

- Sufficient network resources must be available to accept simultaneous call requests arriving over a particular protocol sequence. The value of *max_call_requests* in the five **rpc_server_use_ *_protseq* ()** routines advises the RPC runtime about the runtime's request of network resources.
- Enough call threads must be available to execute the simultaneous call requests once they have been accepted. The value of *max_calls_exec* in **rpc_server_listen()** specifies the number of call threads.

These conditions are independent of each other.

A server application that specifies a value for *max_calls_exec* greater than 1 is responsible for concurrency control among the remote procedures since each executes in a separate thread.

If the server receives more remote procedure calls than it can execute (more calls than the value of *max_calls_exec*), the RPC runtime accepts and queues additional remote procedure calls until a call execution thread is available. From the client's perspective, a queued remote procedure call appears the same as one that the server is actively executing. A client call remains blocked and in the queue until any one of the following events occurs:

- The remote procedure call is assigned to an available call execution thread and the call runs to completion.
- The client no longer can communicate with the server.
- The client thread is canceled and the remote procedure call does not complete within the cancel timeout limits.

The implementation of the RPC architecture determines the amount of queuing it provides.

The RPC runtime continues listening for remote procedure calls (that is, the routine does not return to the server) until one of the following events occurs:

- One of the server application's manager routines calls **rpc_mgmt_stop_server_listening()**.
- A client is allowed to, and makes, a remote **rpc_mgmt_stop_server_listening()** call to the server.

On receiving a request to stop listening, the RPC runtime stops accepting new remote procedure calls for all registered interfaces. Executing calls and existing queued calls are allowed to complete.

After all calls complete, **rpc_server_listen()** returns to the caller, which is a server application.

For more information about a server's listening for and handling incoming remote procedure calls, refer to the *OSF DCE Application Development Guide—Core Components* . It also contains information about canceled threads.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_already_listening
Server already listening.

rpc_s_max_calls_too_small
Maximum calls value too small.

rpc_s_no_protseqs_registered
No protocol sequences registered.

Related Information

Functions: **rpc_mgmt_server_stop_listening(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_server_register_auth_ident

Purpose

Registers user-to-user based authentication information with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_register_auth_ident(
   unsigned_char_p_t *server_princ_name
   unsigned32 authn_svc
   rpc_auth_identity_handle_t auth_identity
   unsigned32 *status);
```

Parameters

Input

server_princ_name
A pointer to the principal name to use for the server when authenticating remote procedure calls. The content of the name and its syntax is defined by the authentication service in use.

authn_svc
Specifies the authentication service to use when the server receives a remote procedure call request. The following authentication services are supported:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

rpc_c_authn_default
DCE default authentication service.

auth_identity
Specifies a handle for the data structure that contains the client's authentication and authorization credentials appropriate for the selected authentication and authorization services.

When using the **rpc_c_authn_dce_secret** authentication service and any authorization service, this value must be a **sec_login_handle_t**, which can be obtained from one of the following routines:

- **sec_login_setup_identity()**
- **sec_login_get_current_context()**
- **sec_login_import_context()**

Specify NULL to use the default security login context for the current address space.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_auth_ident()** routine registers an authentication service to use for authenticating remote procedure calls to a particular server principal. This routine is used for user-to-user authentication where the server principal's credentials are available, but not the server principal's long-term key. Use the **rpc_server_register_auth_info()** routine for server-key based authentication.

A server calls this routine once for each authentication service and principal name combination that it wants to register. The authentication service specified by a client (using the **rpc_binding_set_auth_info()** routine) must be one of the authentication services registered by the server. If it is not, the client's remote procedure call request fails with an **rpc_s_unknown_authn_service** status code.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_unknown_authn_service
Unknown authentication service.

sec_s_user_to_user_disabled
Account is not allowed to use user-to-user protocol registration.

sec_s_multiple_u2u_req
Server identity has already been registered.

sec_s_svr_type_conflict
Simultaneous registration of both keytable and identity is not suppported. Server has already registered with the **rpc_server_register_auth_info()** routine.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_server_register_auth_info(3rpc)**.

rpc_server_register_auth_info

Purpose

Registers server-key based authentication information with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_register_auth_info(
   unsigned_char_t *server_princ_name
   unsigned32 authn_svc
   rpc_auth_key_retrieval_fn_t get_key_fn
   void *arg
   unsigned32 *status);
```

Parameters

Input

server_princ_name
Specifies the principal name to use for the server when authenticating remote procedure calls using the service specified by *authn_svc*. The content of the nam e and its syntax is defined by the authentication service in use.

authn_svc
Specifies the authentication service to use when the server receives a remote procedure call request. The following authentication services are supported:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

rpc_c_authn_default
DCE default authentication service.

get_key_fn
Specifies the address of a server-provided routine that returns encryption keys.

The following C definition for **rpc_auth_key_ret rieval_fn_t** illustrates the prototype for the encryption key acquisition routine:

```
typedef void (*rpc_auth_key_retrieval_fn_t)
 (
 void      *arg,         /* in */
 unsigned_char_t *server_princ_name,  /* in */
 unsigned32   key_type,       /* in */
 unsigned32   key_ver,       /* in */
 void      **key,        /* out */
 unsigned32   *status       /* out */
 );
```

The RPC runtime passes the *server_princ_name* parameter value specified on the call to **rpc_server_register_auth_info( )**, as the *server_princ_name* parameter value, to the *get_key_fn* key acquisition routine. The RPC runtime automatically provides a value for the key version (*key_ver*) parameter. For a *key_ver* value of 0 (zero), the key acquisition routine must return the most recent key available. The routine returns the key in the *key* parameter.

Note:

The *key_type* parameter specifies a Kerberos encryption key type. Because currently the DCE supports only DES encryption, this parameter can be ignored.

If the key acquisition routine, when called from the **rpc_server_register_auth_info( )** routine, returns a status other than **rpc_s_ok**, the **rpc_server_register_auth_info( )** routine fails and returns the error status to the calling server.

If the key acquisition routine, when called by the RPC runtime while authenticating a client remote procedure call request, returns a status other than **rpc_s_ok**, the request fails and the RPC runtime returns the error status to the client.

arg Specifies an argument to pass to the *get_key_fn* key acquisition routine, if specified. (See the description of the *get_key_fn* parameter for details.)

Specify NULL for *arg* to use the default key table file, **/krb/v5srvtab**. The calling server must be root to access this file.

If *arg* is a key table filename, the file must have been created with the **ktadd** command. If the specified key table file resides in **/krb5**, you can supply only the filename. If the file does not reside in **/krb5**, you must supply the full pathname. You must prepend the file's absolute pathname with the prefix **FILE:**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_auth_info()** routine registers an authentication service to use for authenticating remote procedure calls to a particular server principal. This routine is used for server-key based authentication. Use the **rpc_server_register_auth_ident()** routine for user-to-user authentication.

A server calls this routine once for each authentication service and principal name combination that it wants to register. The authentication service specified by a client (using the **rpc_binding_set_auth_info()** routine) must be one of the authentication services registered by the server. If it is not, the client's remote procedure call request fails with an **rpc_s_unknown_authn_service** status code.

The following table shows the RPC runtime behavior for acquiring encryption keys for each supported authentication service. Note that if *authn_svc* is **rpc_c_authn_default**, then *get_key_fn* must be NULL.

Table 27. RPC Key Acquisition for Authentication Services

authn_svc	*get_key_fn*	*arg*	**Runtime Behavior**
rpc_c_authn_default	NULL	NULL	Uses the default method of encryption key acquisition from the default key table.
rpc_c_authn_default	NULL	non-NULL	Uses the default method of encryption key acquisition from the specified key table.
rpc_c_authn_default	non-NULL	Ignored	Error returned.
rpc_c_authn_none	Ignored	Ignored	No authentication performed.
rpc_c_authn_dce_secret	NULL	NULL	Uses the default method of encryption key acquisition from the default key table.
rpc_c_authn_dce_secret	NULL	non-NULL	Uses the default method of encryption key acquisition from the specified key table.
rpc_c_authn_dce_secret	non-NULL	NULL	Uses the specified encryption key acquisition routine to obtain keys from the default key table.

Table 28. RPC Key Acquisition for Authentication Services

authn_svc	*get_key_fn*	*arg*	**Runtime Behavior**
rpc_c_authn_dce_secret	non-NULL	non-NULL	Uses the specified encryption key acquisition routine to obtain keys from the specified key table.
rpc_c_authn_dce_public	Ignored	Ignored	(Reserved for future use.)

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_unknown_authn_service
Unknown authentication service.

rpc_s_key_func_not_allowed
authn_svc is **rpc_c_authn_default** and a nonnull value was supplied for *get_key_fn* parameter.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_server_register_auth_ident(3rpc)**.

rpc_server_register_if

Purpose

Registers an interface with the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_register_if(
   rpc_if_handle_t if_handle
   uuid_t *mgr_type_uuid
   rpc_mgr_epv_t mgr_epv
   unsigned32 *status);
```

Parameters

Input

if_handle
An IDL-generated data structure specifying the interface to register.

mgr_type_uuid
Specifies a type UUID to associate with the *mgr_epv* parameter. Specifying the value NULL (or a nil UUID) registers the *if_handle* with a nil type UUID.

mgr_epv
Specifies the manager routines' entry point vector. To use the IDL-generated default entry point vector, specify NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_register_if()** routine registers a server interface with the RPC runtime. A server can register an unlimited number of interfaces. Once registered, an interface is available to clients through any binding handle of the server, provided that the binding handle is compatible for the client.

A server must provide the following information to register an interface:

- An interface specification, which is a data structure generated by the IDL compiler. The server specifies the interface specification of the interface using the *if_handle* parameter.
- A type UUID and manager entry point vector (EPV), a data pair that determines which manager routine executes when a server receives a remote procedure call request from a client.

  The server specifies the type UUID and EPV using the *mgr_type_uuid* and *mgr_epv* parameters, respectively. Note that when a nonnil type UUID is specified, the server must also call the **rpc_object_set_type()** routine to register objects of this nonnil type.

A server that only offers a single manager for an interface calls **rpc_server_register_if()** once for that interface. In the simple case where the

single manager's entry point names are the same as the operation names in the IDL interface definition, the IDL-generated default manager EPV for the interface may be used. The value NULL in *mgr_epv* specifies the default manager EPV.

Note that if a server offers multiple implementations of an interface, the server code must register a separate manager entry point vector for each interface implementation.

Rules for Invoking Manager Routines

The RPC runtime dispatches an incoming remote procedure call to a manager that offers the requested RPC interface. When multiple managers are registered for an interface, the RPC runtime must select one of them. To select a manager, the RPC runtime uses the object UUID specified by the call's binding handle. The following table summarizes the rules applied for invoking manager routines.

Table 29. Rules for Invoking Manager Routines

Object UUID of Call [1]	Has Server Set Type of Object UUID? [2]	Has Server Set Type for Manager EPV? [3]	Dispatching Action
Nil	Not applicable[4]	Yes	Uses the manager with the nil type UUID.
Nil	Not applicable[4]	No	The RPC error (**rpc_s_unknown_mgr_type**). Rejects the remote procedure call.
Non-nil	Yes	Yes	Uses the manager with the same type UUID.

Table 30. Rules for Invoking Manager Routines

Object UUID of Call [1]	Has Server Set Type of Object UUID? [2]	Has Server Registered Type for Manager EPV? [3]	Dispatching Action
Non-nil	No	Ignored	Uses the manager with the nil type UUID. If no manager with the nil type UUID, **rpc_s_unknown_mgr_type**. Rejects the remote procedure call.
Non-nil	Yes	No	The error (**rpc_s_unknown_mgr_type**). Rejects the remote procedure call.

[1] This is the object UUID found in a binding handle for a remote procedure.

[2] By calling **rpc_object_set_type()** to specify the type UUID for an object.

[3] By calling **rpc_server_register_if()** using the same type UUID.

[4] The nil object UUID is always automatically assigned the nil type UUID. It is illegal to specify a nil object UUID in **rpc_object_set_type()**.

For more information about registering server interfaces and invoking manager routines, refer to the *OSF DCE Application Development Guide—Core Components*.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_type_already_registered
An interface with the given type of UUID is already registered.

Related Information

Functions: **rpc_binding_set_object(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_object_set_type(3rpc)**, **rpc_server_unregister_if(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_server_unregister_if

Purpose

Removes an interface from the RPC runtime; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_unregister_if(
   rpc_if_handle_t if_handle
   uuid_t *mgr_type_uuid
   unsigned32 *status);
```

Parameters

Input

if_handle
Specifies an interface specification to unregister (remove).

Specify NULL to remove all interfaces previously registered with the type UUID value given in the *mgr_type_uuid* parameter.

mgr_type_uuid
Specifies the type UUID for the manager entry point vector (EPV) to remove. This needs to be the same value as provided in a call to the **rpc_server_register_if()** routine.

Specify NULL to remove the interface given in the *if_handle* parameter for all previously registered type UUIDs.

Specify a nil UUID to remove the IDL-generated default manager EPV. In this case all manager EPVs registered with a nonnil type UUID remain registered.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_unregister_if()** routine removes the association between an interface and a manager entry point vector (EPV).

Specify the manager EPV to remove by providing, in the *mgr_type_uuid* parameter, the type UUID value specified in a call to the **rpc_server_register_if()** routine. Once removed, an interface is no longer available to client applications.

When an interface is removed, the RPC runtime stops accepting new calls for that interface. Executing calls (on that interface) are allowed to complete.

The table below summarizes the actions of this routine.

Table 31. Rules for Removing an Interface

if_handle	*mgr_type_uuid*	**Action**
nonNULL	non-NULL	Removes the manager EPV associated with the specified parameters.
nonNULL	NULL	Removes all manager EPVs associated with parameter *if_handle*.
NULL	non-NULL	Removes all manager EPVs associated with parameter *mgr_type_uuid*.
NULL	NULL	Removes all manager EPVs.

Note that when both of the parameters *if_handle* and *mgr_type_uuid* are given the value NULL, this call has the effect of preventing the server from receiving any new remote procedure calls since all the manager EPVs for all interfaces have been removed.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

Related Information

Functions: **rpc_server_register_if(3rpc)**.

rpc_server_use_all_protseqs

Purpose

Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_use_all_protseqs(
   unsigned32 max_call_requests
   unsigned32 *status);
```

Parameters

Input

max_call_requests
Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_all_protseqs()** routine registers all supported protocol sequences with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains a dynamic endpoint that the RPC runtime and operating system generated.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

After registering protocol sequences, a server typically calls the following routines:

rpc_server_inq_bindings()
Obtains a vector containing all of the server's binding handles.

rpc_ep_register()
Registers the binding handles with the local endpoint map.

rpc_ep_register_no_replace()
Registers the binding handles with the local endpoint map.

rpc_ns_binding_export()
Places the binding handles in the name service database for access by any client.

rpc_binding_vector_free()
Frees the vector of server binding handles.

rpc_server_register_if()
Registers with the RPC runtime those interfaces that the server offers.

rpc_server_listen()
Enables the reception of remote procedure calls.

To register protocol sequences selectively, a server calls one of the following routines:

- **rpc_server_use_protseq()**
- **rpc_server_use_all_protseqs_if()**
- **rpc_server_use_protseq_if()**
- **rpc_server_use_protseq_ep()**

For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_binding_vector_free(3rpc)**,

rpc_ep_register(3rpc), **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_use_all_protseqs_if

Purpose

Tells the RPC runtime to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_all_protseqs_if(
   unsigned32 max_call_requests
   rpc_if_handle_t if_handle
   unsigned32 *status);
```

Parameters

Input

max_call_requests

Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater that the value of *max_call_requests* and can vary for each protocol sequence.

Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

Note that in this version of DCE RPC, any number you specify is replaced by the default value.

Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

if_handle

Specifies an interface specification containing the protocol sequences and their corresponding endpoint information to use in creating binding handles. Each created binding handle contains a well-known (nondynamic) endpoint contained in the interface specification.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_all_protseqs_if()** routine registers all protocol sequences and associated endpoint address information provided in the IDL file with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the IDL file.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

If you want to register selected protocol sequences specified in the IDL, your server uses **rpc_server_use_protseq_if()**.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. (However, a server that uses only **rpc_server_use_all_protseqs_if()** does not subsequently call **rpc_ep_register()** or **rpc_ep_register_no_replace()**.) For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_cant_inq_socket
Cannot inquire endpoint from socket.

rpc_s_invalid_endpoint_format
Invalid interface handle.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_no_protseqs
No supported protocol sequences.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**,

rpc_server_use_protseq_if(3rpc).

rpc_server_use_protseq

Purpose

Tells the RPC runtime to use the specified protocol sequence for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_protseq(
   unsigned_char_t *protseq
   unsigned32 max_call_requests
   unsigned32 *status);
```

Parameters

Input

protseq
: Specifies a string identifier for the protocol sequence to register with the RPC runtime. (For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.)

max_call_requests
: Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

  The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

  Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

  Note that in this version of DCE RPC, any number you specify is replaced by the default value.

  Also, **rpc_server_listen()** limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq()** routine registers a single protocol sequence with the RPC runtime. A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains a dynamic endpoint that the RPC runtime and operating system generated.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

A server calls **rpc_server_use_all_protseqs()** to register all protocol sequences.

The explanation of the **rpc_server_use_all_protseqs()** routine contains a list of the routines a server typically calls after calling this routine. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_network_is_protseq_valid(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**, **rpc_server_use_protseq_if(3rpc)**.

rpc_server_use_protseq_ep

Purpose

Tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_server_use_protseq_ep(
   unsigned_char_t *protseq
   unsigned32 max_call_requests
   unsigned_char_t *endpoint
   unsigned32 *status);
```

Parameters

Input

protseq
: Specifies a string identifier for the protocol sequence to register with the RPC runtime. (For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.

max_call_requests
: Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

  The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

  Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

  Note that in this version of DCE RPC, any number you specify is replaced by the default value.

  Also, **rpc_server_listen()** limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

endpoint
: Specifies address information for an endpoint. This information is used in creating a binding handle for the protocol sequence specified in the *protseq* parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq_ep()** routine registers a protocol sequence and its specified endpoint address information with the RPC runtime. A server must register

at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences and endpoints.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the *endpoint* parameter.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_cant_create_socket
Cannot create socket.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**.

rpc_server_use_protseq_if

Purpose

Tells the RPC runtime to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls; used by server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_server_use_protseq_if(
   unsigned_char_t *protseq
   unsigned32 max_call_requests
   rpc_if_handle_t if_handle
   unsigned32 *status);
```

Parameters

Input

protseq
: Specifies a string identifier for the protocol sequence to register with the RPC runtime. For a list of string identifiers, see the table of valid protocol sequences in the **rpc_intro(3rpc)** reference page.

max_call_requests
: Specifies the maximum number of concurrent remote procedure call requests that the server can accept.

  The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. The actual number of these requests can be greater than the value of *max_call_requests* and can vary for each protocol sequence.

  Use the value **rpc_c_protseq_max_reqs_default** to specify the default parameter value.

  Note that in this version of DCE RPC, any number you specify is replaced by the default value.

  Also, the **rpc_server_listen()** routine limits (according to its *max_calls_exec* parameter) the amount of concurrent remote procedure call execution. See the **rpc_server_listen(3rpc)** reference page for more information.

if_handle
: Specifies an interface specification whose endpoint information is used in creating a binding for the protocol sequence specified in the *protseq* parameter. Each created binding handle contains a well-known (nondynamic) endpoint contained in the interface specification.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_server_use_protseq_if()** routine registers one protocol sequence with the RPC runtime, including its endpoint address information as provided in the specified IDL file.

A server must register at least one protocol sequence with the RPC runtime to receive remote procedure call requests. A server can call this routine multiple times to register additional protocol sequences.

For each protocol sequence registered by a server, the RPC runtime creates one or more binding handles. Each binding handle contains the well-known endpoint specified in the IDL file.

The *max_call_requests* parameter allows you to specify the maximum number of concurrent remote procedure call requests the server handles.

To register all protocol sequences from the IDL, a server calls the **rpc_server_use_all_protseqs_if()** routine.

The explanation of **rpc_server_use_all_protseqs()** contains a list of the routines a server typically calls after calling this routine. However, a server that uses only **rpc_server_use_protseq_if()** does not subsequently call **rpc_ep_register()** or **rpc_ep_register_no_replace()**. For an explanation of how a server can establish a client/server relationship without using the local endpoint map or the name service database, see the information on string bindings in the **rpc_intro(3rpc)** reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_calls_too_large_for_wk_ep
Maximum concurrent calls too large.

rpc_s_cant_bind_socket
Cannot bind to socket.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_invalid_rpc_protseq
Invalid protocol sequence.

rpc_s_max_descs_exceeded
Exceeded maximum number of network descriptors.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_vector_free(3rpc)**, **rpc_ep_register(3rpc)**, **rpc_ep_register_no_replace(3rpc)**, **rpc_ns_binding_export(3rpc)**, **rpc_server_inq_bindings(3rpc)**, **rpc_server_listen(3rpc)**, **rpc_server_register_if(3rpc)**, **rpc_server_use_all_protseqs(3rpc)**, **rpc_server_use_all_protseqs_if(3rpc)**, **rpc_server_use_protseq(3rpc)**, **rpc_server_use_protseq_ep(3rpc)**.

rpc_sm_allocate

Purpose

Allocates memory within the RPC stub memory management scheme.

Synopsis

```
#include <rpc.h>

idl_void_p_t rpc_sm_allocate(
   unsigned long size
   unsigned32 *status);
```

Parameters

Input

size Specifies, in bytes, the size of memory to be allocated.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_allocate()** to allocate memory within the RPC stub memory management scheme. Before a call to this routine, the stub memory management environment must have been established. For manager code that is called from the stub, the stub itself normally establishes the necessary environment. When **rpc_sm_allocate()** is used by code that is not called from the stub, the application must establish the required memory management environment by calling **rpc_sm_enable_allocate()**.

When the stub establishes the memory management environment, the stub itself frees any memory allocated by **rpc_sm_allocate()**. The application can free such memory before returning to the calling stub by calling **rpc_sm_free()**.

When the application establishes the memory management environment, it must free any memory allocated, either by calling **rpc_sm_free()** or by calling **rpc_sm_disable_allocate()**.

Multiple threads may call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory within the stub memory management environment. To do so, the threads must share the same stub memory management thread handle. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

Return Values

A pointer to the allocated memory.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_free(3rpc)**, **rpc_sm_enable_allocate(3rpc)**, **rpc_sm_disable_allocate(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**.

rpc_sm_client_free

Purpose

Frees memory returned from a client stub

Synopsis

```
#include <rpc.h>

void  rpc_sm_client_free(
   idl_void_p_t node_to_free
   unsigned32 *status);
```

Parameters

Input

node_to_free
Specifies a pointer to memory returned from a client stub.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_client_free()** routine releases memory allocated and returned from a client stub. The thread calling **rpc_sm_client_free()** must have the same thread handle as the thread that made the RPC call. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_free(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_client_alloc_free(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**, **rpc_sm_swap_client_alloc_free(3rpc)**.

rpc_sm_destroy_client_context

Purpose

Reclaims the client memory resources for a context handle, and sets the context handle to null

Synopsis

```
#include <rpc.h>

void  rpc_sm_destroy_client_context(
   idl_void_p_t p_unusable_context_handle
   unsigned32 *status);
```

Parameters

Input

p_unusable_context_handle
Specifies the context handle that can no longer be accessed.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_destroy_client_context()** routine is used by client applications to reclaim the client resources used in maintaining an active context handle. Applications call this routine after a communications error makes the context handle unusable. When the **rpc_sm_destroy_client_context()** routine reclaims the memory resources, it also sets the context handle to null.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_sm_disable_allocate

Purpose

Releases resources and allocated memory within the stub memory management scheme

Synopsis

```
#include <rpc.h>

void  rpc_sm_disable_allocate(
   unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_disable_allocate()** routine releases all resources acquired by a call to **rpc_sm_enable_allocate()**, and any memory allocated by calls to **rpc_sm_allocate()** after the call to **rpc_sm_enable_allocate()** was made.

The **rpc_sm_enable_allocate()** and **rpc_sm_disable_allocate()** routines must be used in matching pairs.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_enable_allocate(3rpc)**.

rpc_sm_enable_allocate

Purpose

Enables the stub memory managment environment

Synopsis

```
#include <rpc.h>

void  rpc_sm_enable_allocate(
   unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications can call **rpc_sm_enable_allocate()** to establish a stub memory management environment in cases where one is not established by the stub itself. A stub memory management environment must be established before any calls are made to **rpc_sm_allocate()**. For server manager code called from the stub, the stub memory management environment is normally established by the stub itself. Code that is called from other contexts needs to call **rpc_sm_enable_allocate()** before calling **rpc_sm_allocate()**.

Note:

For a discussion of how spawned threads acquire a stub memory management environment, see the **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** reference pages.

Return Values

None

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_disable_allocate(3rpc)**.

rpc_sm_free

Purpose

Frees memory allocated by the **rpc_sm_allocate()** routine

Synopsis

```
#include <rpc.h>

void  rpc_sm_free(
   idl_void_p_t node_to_free
   unsigned32 *status);
```

Parameters

Input

node_to_free
Specifies a pointer to memory allocated by **rpc_sm_allocate()**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_free()** to release memory allocated by **rpc_sm_allocate()**.

When the stub allocates memory within the stub memory management environment, manager code called from the stub can also use **rpc_sm_free()** to release memory allocated by the stub.

The thread calling **rpc_sm_free()** must have the same thread handle as the thread that allocated the memory with **rpc_sm_allocate()**. Applications pass thread handles from thread to thread by calling **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()**.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**, **rpc_sm_set_thread_handle(3rpc)**.

rpc_sm_get_thread_handle

Purpose

Gets a thread handle for the stub memory management environment

Synopsis

```
#include <rpc.h>

rpc_ss_thread_handle_t rpc_sm_get_thread_handle(
   unsigned32 *status);
```

Parameters

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

Applications call **rpc_sm_get_thread_handle()** to get a thread handle for the current stub memory management environment. A thread that is managing memory within the stub memory managment scheme calls pc_sm_get_thread_handle() to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle()** with this handle, is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle()** to get a thread handle before spawning new threads that will manage the same memory. The spawned threads then call **rpc_sm_set_thread_handle()** with the handle provided by the parent thread.

Note:

Typically, **rpc_sm_get_thread_handle()** is called by a server manager routine before it spawns additional threads. Normally the stub sets up the memory management environment for the manager routine. The manager calls **rpc_sm_get_thread_handle()** to make this environment available to the spawned threads.

A thread may also use **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** to save and restore its memory management environment.

Return Values

A thread handle.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_set_thread_handle(3rpc**.

rpc_sm_set_client_alloc_free

Purpose

Sets the memory allocation and freeing mechanisms used by the client stubs

Synopsis

```
#include <rpc.h>

void rpc_sm_set_client_alloc_free(
   idl_void_p_t (* p_allocate) (
   unsigned long  size),
   void (* p_free) (
   idl_void_p_t  ptr),
   unsigned32
* status);
```

Parameters

Input

p_allocate
Specifies a memory allocator routine.

p_free Specifies a memory free routine. This routine is used to free memory allocated with the routine specified by *p_allocate*.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_set_client_alloc_free()** routine overrides the default routines that the client stub uses to manage memory.

Note:

The default memory management routines are ISO **malloc()** and ISO **free()** except when the remote call occurs within manager code in which case the default memory management routines are **rpc_sm_allocate()** and **rpc_sm_free()**.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**.

rpc_sm_set_thread_handle

Purpose

Sets a thread handle for the stub memory management environment

Synopsis

```
#include <rpc.h>

void rpc_sm_set_thread_handle(
   rpc_ss_thread_handle_t id
   unsigned32 *status);
```

Parameters

Input

id Specifies a thread handle returned by a call to the routine **rpc_sm_get_thread_handle()**.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application thread calls **rpc_sm_set_thread_handle()** to set a thread handle for memory management within the stub memory management environment. A thread that is managing memory within the stub memory managment scheme calls **rpc_sm_get_thread_handle()** to get a thread handle for its current stub memory management environment. A thread that calls **rpc_sm_set_thread_handle()** with this handle is able to use the same memory management environment.

When multiple threads call **rpc_sm_allocate()** and **rpc_sm_free()** to manage the same memory, they must share the same thread handle. The thread that established the stub memory management environment calls **rpc_sm_get_thread_handle()** to get a thread handle before spawning new threads that will manage the same memory. The spawned threads then call **rpc_sm_set_thread_handle()** with the handle provided by the parent thread.

Note:

Typically, **rpc_sm_set_thread_handle()** is called by a thread spawned by a server manager routine. Normally the stub sets up the memory management environment for the manager routine and the manager calls **rpc_sm_get_thread_handle()** to get a thread handle. Each spawned thread then calls **rpc_sm_get_thread_handle()** to get access to the manager's memory management environment.

A thread may also use **rpc_sm_get_thread_handle()** and **rpc_sm_set_thread_handle()** to save and restore its memory management environment.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_get_thread_handle(3rpc)**.

rpc_sm_swap_client_alloc_free

Purpose

Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client

Synopsis

```
#include <rpc.h>

void rpc_sm_swap_client_alloc_free (
   idl_void_p_t (* p_allocate) (
   unsigned long  size),
   void (* p_free) (
   idl_void_p_t  ptr),
   idl_void_p_t (** p_p_old_allocate) (
   unsigned long  size),
   void (** p_p_old_free) (
   idl_void_p_t ptr),
   unsigned32 * status);
```

Parameters

Input

p_allocate
Specifies a new memory allocation routine.

p_free Specifies a new memory free routine.

Output

p_p_old_allocate
Returns the memory allocation routine in use before the call to this routine.

p_p_old_free
Returns the memory free routine in use before the call to this routine.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_sm_swap_client_alloc_free()** routine exchanges the current allocate and free mechanisms used by the client stubs for routines supplied by the caller.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_sm_allocate(3rpc)**, **rpc_sm_free(3rpc)**, **rpc_sm_set_client_alloc_free(3rpc)**.

rpc_ss_allocate

Purpose

Allocates memory within the RPC stub memory management scheme; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

idl_void_p_t rpc_ss_allocate(
   idl_size_t size);
```

Parameters

Input

size Specifies, in bytes, the size of memory to be allocated.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

Usually, the **rpc_ss_allocate()** routine is used in the manager code that is called from a server stub. Memory allocated by **rpc_ss_allocate()** is released by the server stub after marshalling any output parameters at the end of the remote call in which the memory was allocated. If you want to release memory allocated by **rpc_ss_allocate()** before returning from the manager code use **rpc_ss_free()**.

You can also use **rpc_ss_free()** in manager code to release memory pointed to by a full pointer (**ptr**) in an input parameter.

When the server uses **rpc_ss_allocate()**, the server stub creates the environment the routine needs. If the parameters of the operation include any pointers other than those used for passing parameters by reference, the environment is set up automatically.

If you need to use **rpc_ss_allocate()** in a manager code routine that does not have a pointer in any of its parameters, use an ACF and apply the **enable_allocate** attribute to the relevant operation. This causes the generated server stub to set up the necessary environment.

Note that memory allocated by allocators other than **rpc_ss_allocate()** is not released when the operation on the server side completes execution.

If you want to use **rpc_ss_allocate()** outside the code called from a server stub, you must first create an environment for it by calling **rpc_ss_enable_allocate()**.

See the *OSF DCE Application Development Guide—Core Components* for more information.

Return Values

A pointer to the allocated memory.

An exception, **rpc_x_no_memory**, when no memory is available for allocation.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_disable_allocate(3rpc)**, **rpc_ss_enable_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

rpc_ss_bind_authn_client

Purpose

Authenticates a client's identity to a server from a client stub; a pointer to the server binding handle for the remote procedure call to which the routine will add authentication and authorization context

Synopsis

```
#include <rpc.h>

void  rpc_ss_bind_authn_client(
   rpc_binding_handle_t *binding
   if_handle_t if_handle
   error_status_t *status);
```

Parameters

Input/Output

binding
A pointer to the server binding handle for the remote procedure call to which the routine will add authentication and authorization context.

Input

if_handle
A stub-generated data structure that specifies the interface of interest. The routine can use this parameter to resolve a partial binding or to distinguish between interfaces.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_ss_bind_authn_client()** routine is a DCE-supplied binding callout routine for use with the **binding_callout** ACF interface attribute.

The **binding_callout** attribute enables applications to specify the name of a routine that the client stub will call automatically to modify a server binding handle with additional information before it initiates a remote procedure call. This attribute is especially useful for applications using the automatic binding method, where it is the client stub that obtains the binding handle, rather than the application code. The **binding_callout** attribute provides these applications with a way to gain access to a server binding handle from the client stub, since the handle is not accessible from the application code.

Applications can specify **rpc_ss_bind_authn_client()** to the **binding_callout** ACF interface attribute in order to authenticate the client's identity to a server from the client stub before the remote procedure call to the server is initiated. This routine performs one-way authentication: the client does not care which server principal receives the remote procedure call request, but the server verifies that the client is who the client claims to be.

The routine sets the protection level used, the authentication identity, and the authentication service used to their default values. See the **rpc_binding_set_auth_info(3rpc)** reference page for more information on these default values. It sets the authorization service to perform authorization based on the client's principal name.

Applications can also specify user-written binding callout routines with the **binding_callout** attribute to modify server binding handles from client stubs with other types of information. For more information on using the **binding_callout** ACF attribute, see the *OSF DCE Application Development Guide—Core Components*.

Return Values

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
Success.

rpc_s_no_more_bindings
Directs the client stub not to look for another server binding.

Related Information

Functions: **rpc_binding_set_auth_info(3rpc)**, **rpc_ep_resolve_binding(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**.

Books: *OSF DCE Application Development Guide—Introduction and Style Guide*, *OSF DCE Application Development Guide—Core Components*.

rpc_ss_client_free

Purpose

Frees memory returned from a client stub; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_client_free(
   idl_void_p_t node_to_free);
```

Parameters

Input

node_to_free
Specifies a pointer to memory returned from a client stub.

Description

The **rpc_ss_client_free()** routine releases memory allocated and returned from a client stub. The thread calling **rpc_ss_client_free()** must have the same thread handle as the thread that made the RPC call.

This routine enables a routine to deallocate dynamically allocated memory returned by an RPC call without knowledge of the memory management environment from which it was called.

Note that while this routine is always called from client code, the code can be executing as part of another server.

Return Values

No value is returned.

Related Information

Functions: **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_client_alloc_free(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**, **rpc_ss_swap_client_alloc_free(3rpc)**.

rpc_ss_destroy_client_context

Purpose

Reclaims the client memory resources for the context handle, and sets the context handle to NULL; used by client applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_ss_destroy_client_context(
   void *p_unusable_context_handle);
```

Parameters

Input

p_unusable_context_handle
Specifies the context handle that can no longer be accessed.

Description

The **rpc_ss_destroy_client_context()** routine is used by the client application to reclaim the client resources used in maintaining an active context handle. Only call this after a communications error makes the context handle unusable. When **rpc_ss_destroy_client_context()** reclaims the memory resources, it also sets the context handle to null.

Return Values

No value is returned.

The **rpc_ss_destroy_client_context()** routine raises no exceptions.

rpc_ss_disable_allocate

Purpose

Releases resources and allocated memory; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_disable_allocate(void);
```

Description

The **rpc_ss_disable_allocate()** routine releases (disables) all resources acquired by a call to **rpc_ss_enable_allocate()**, and any memory allocated by calls to **rpc_ss_allocate()** after the call to **rpc_ss_enable_allocate()** was made.

The **rpc_ss_enable_allocate()** and **rpc_ss_disable_allocate()** routines must be used in matching pairs.

For information about rules for using memory management routines, see the *OSF DCE Application Development Guide—Core Components*.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_enable_allocate(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_ss_enable_allocate

Purpose

Enables the allocation of memory by the **rpc_ss_allocate()** routine when not in manager code; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_enable_allocate(void);
```

Description

In sophisticated servers, it may be necessary to call manager code routines from different environments. This occurs, for example, when the application is both a client and a server of the same interface. Therefore, a manager code routine may need to be called both by the application code and by the stub code. If code, other than manager code, calls the **rpc_ss_allocate()** routine, it must first call **rpc_ss_enable_allocate()** to initialize the memory management environment that **rpc_ss_allocate()** uses.

For information about rules for using memory management routines, see the *OSF DCE Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_disable_allocate(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_ss_free

Purpose

Frees memory allocated by the **rpc_ss_allocate()** routine; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_free(
   idl_void_p_t node_to_free);
```

Parameters

Input

node_to_free
Specifies a pointer to memory allocated by **rpc_ss_allocate()**.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

The **rpc_ss_free()** routine releases memory allocated by **rpc_ss_allocate()**. The thread calling **rpc_ss_free()** must have the same thread handle as the thread that allocated the memory with **rpc_ss_allocate()**. Use it only in an environment where **rpc_ss_allocate()** is used.

If the manager code allocates memory with **rpc_ss_allocate()** and the memory is not released by **rpc_ss_free()** during manager code execution, then the server stub automatically releases the memory when the manager code completes execution and returns control to the stub.

Manager code can also use **rpc_ss_free()** to release memory that is pointed to by a full pointer in an input parameter.

For information about rules for using memory management routines, see the *OSF DCE Application Development Guide—Core Components*.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_ss_get_thread_handle

Purpose

Gets a thread handle for the manager code before it spawns additional threads, or for the client code when it becomes a server; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

rpc_ss_thread_handle_t rpc_ss_get_thread_handle(void);
```

Description

The **rpc_ss_get_thread_handle()** routine is used by a server manager thread when it spawns additional threads. To spawn additional threads that are able to perform memory management, the server manager code calls **rpc_ss_get_thread_handle()** and passes the thread handle to each spawned thread. Each spawned thread that uses **rpc_ss_allocate()** and **rpc_ss_free()** for memory management must first call **rpc_ss_set_thread_handle()**, using the handle obtained by the original manager thread.

The **rpc_ss_get_thread_handle()** routine can also be used when a program changes from being a client to being a server. The program gets a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**. When the program reverts to being a client it re−establishes the client environment by calling **rpc_ss_set_thread_handle()**, supplying the previously obtained handle as a parameter.

Return Values

A thread handle.

Examples

This function determines the thread handle, creates a thread, and passes the thread handle to the thread so it can share the memory management environment of the calling thread.

```
#include <pthread.h>
#include <idlbase.h>

pthread_t Launch_thread(
  int (*routine_to_launch)(
    pthread_addr_t th
    )
  )
{
  rpc_ss_thread_handle_t th = rpc_ss_get_thread_handle();
  pthread_t t;

  /*
   * Create the thread and pass to it the thread handle
   * so it can use rpc_ss_set_thread_handle.
```

```
   */
  pthread_create (&t, pthread_attr_default,
    (pthread_startroutine_t)routine_to_launch,
    (pthread_addr_t)th);

  return t;
}
```

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_set_thread_handle(3rpc)**.

rpc_ss_set_client_alloc_free

Purpose

Sets the memory allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage memory for pointed-to nodes; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_set_client_alloc_free (
   idl_void_p_t (* p_allocate) (
   unsigned long  size),
   void (* p_free) (
   idl_void_p_t * ptr));
```

Parameters

Input

p_allocate
Specifies a pointer to a routine that has the same procedure declaration as the **malloc()** routine and that is used by the client stub to allocate memory.

p_free Specifies a pointer to a routine that has the same procedure declaration as the **free()** routine and that is used to free memory that was allocated using the routine pointed at by *p_allocate*.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

The **rpc_ss_set_client_alloc_free()** routine overrides the default routines that the client stub uses to manage memory for pointed-to nodes. The default memory management routines are **malloc()** and **free()**, except when the remote call occurs within manager code, in which case the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

For information about rules for using memory management routines, see the *OSF DCE Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_ss_set_thread_handle

Purpose

Sets the thread handle for either a newly created spawned thread or for a server that was formerly a client and is ready to be a client again; used by server or possibly by client applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_ss_set_thread_handle(
   rpc_ss_thread_handle_t id);
```

Parameters

Input

id A thread handle returned by a call to **rpc_ss_get_thread_handle()**.

Description

The **rpc_ss_set_thread_handle()** routine is used by a thread spawned in the manager code to associate itself with the main RPC manager thread. Each spawned thread that uses **rpc_ss_allocate()** and **rpc_ss_free()** for memory management must call **rpc_ss_set_thread_handle()**, using the handle that the main RPC manager thread obtained through **rpc_ss_get_thread_handle()**.

The **rpc_ss_set_thread_handle()** routine can also be used by a program that originally was a client, then became a server, and is now reverting to a client. The program must re−establish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the handle it received (through **rpc_ss_get_thread_handle()**) prior to becoming a server, as a parameter.

Return Values

An exception, **rpc_x_no_memory**, when there is insufficient memory available to set up necessary data structures.

Examples

When this function is invoked within a spawned thread, its argument is the thread handle of the calling thread. This example assumes the data passed to the thread consists of only the middle thread.

```
#include <pthread.h>
#include <dce/idlbase.h>

 int helper_thread (
   pthread_addr_t th
   )
 {
   /*
   * Set the memory management environment to match
   * the parent environment.
```

```
   */
   rpc_ss_set_thread_handle(rpc_ss_thread_handle_t)th;
   /*
   * Real work of this thread follows here ...
   */
 }
```

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_get_thread_handle(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_ss_swap_client_alloc_free

Purpose

Exchanges the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client; used by client applications

Synopsis

```
#include <dce/rpc.h>

void rpc_ss_swap_client_alloc_free(
   idl_void_p_t (* p_allocate) (
   idl_size_t  size),
   void (* p_free) (
   idl_void_p_t  ptr),
   idl_void_p_t (** p_p_old_allocate) (
   idl_size_t  size),
   void (** p_p_old_free) (
   idl_void_p_t  ptr));
```

Parameters

Input

p_allocate
: Specifies a pointer to a routine that has the same procedure declaration as the **malloc()** routine and that is used for allocating client stub memory.

p_free
: Specifies a pointer to a routine that has the same procedure declaration as the **free()** routine and that is used for freeing client stub memory.

Output

p_p_old_allocate
: Specifies a pointer to a pointer to a routine that has the same procedure declaration as the **malloc()** routine. A pointer to the routine that was previously used to allocate client stub memory is returned in this parameter.

p_p_old_free
: Specifies a pointer to a pointer to a routine that has the same procedure declaration as the **free()** routine. A pointer to the routine that was previously used to free client stub memory is returned in this parameter.

Note that in ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Description

The **rpc_ss_swap_client_alloc_free()** routine exchanges the current client allocate and free mechanism used by the client stubs for one supplied by the caller. If it is appropriate for the client code called by an application to use a certain memory allocation and freeing mechanism, regardless of its caller's state, the client code can swap its own mechanism into place on entry, replacing its caller's mechanism. It can then swap the caller's mechanism back into place prior to returning.

For information about rules for using memory management routines, see the *OSF DCE Application Development Guide—Core Components*.

Return Values

An exception, **rpc_x_no_memory**, is returned when there is insufficient memory available to set up necessary data structures.

Errors

A representative list of errors that might be returned is not shown here. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **rpc_ss_allocate(3rpc)**, **rpc_ss_free(3rpc)**, **rpc_ss_set_client_alloc_free(3rpc)**.

Books: *OSF DCE Application Development Guide—Core Components*.

rpc_string_binding_compose

Purpose

Combines the components of a string binding into a string binding; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void rpc_string_binding_compose(
   unsigned_char_t *obj_uuid
   unsigned_char_t *protseq
   unsigned_char_t *network_addr
   unsigned_char_t *endpoint
   unsigned_char_t *options
   unsigned_char_t **string_binding
   unsigned32 *status);
```

Parameters

Input

obj_uuid
Specifies a NULL-terminated string representation of an object UUID.

protseq
Specifies a NULL-terminated string representation of a protocol sequence.

network_addr
Specifies a NULL-terminated string representation of a network address.

endpoint
Specifies a NULL-terminated string representation of an endpoint.

options
Specifies a NULL-terminated string representation of network options.

Output

string_binding
Returns a pointer to a NULL-terminated string representation of a binding handle.

Specify NULL to prevent the routine from returning this argument. In this case the application does not call **rpc_string_free()**.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_binding_compose()** routine combines string binding handle components into a string binding handle.

The RPC runtime allocates memory for the string returned in the *string_binding* parameter. The application calls **rpc_string_free()** to deallocate that memory.

Specify NULL or provide a null string (**\0**) for each input string that has no data.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_string_binding_parse(3rpc)**, **rpc_string_free(3rpc)**, **uuid_to_string(3rpc)**.

rpc_string_binding_parse

Purpose

Returns, as separate strings, the components of a string binding; used by client or server applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_string_binding_parse(
   unsigned_char_t *string_binding
   unsigned_char_t **obj_uuid
   unsigned_char_t **protseq
   unsigned_char_t **network_addr
   unsigned_char_t **endpoint
   unsigned_char_t **network_options
   unsigned32 *status);
```

Parameters

Input

string_binding
Specifies a NULL-terminated string representation of a binding.

Output

obj_uuid
Returns a pointer to a NULL-terminated string representation of an object UUID.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

protseq
Returns a pointer to a NULL-terminated string representation of a protocol sequence.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

network_addr
Returns a pointer to a NULL-terminated string representation of a network address.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

endpoint
Returns a pointer to a NULL-terminated string representation of an endpoint.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

network_options
Returns a pointer to a NULL-terminated string representation of network options.

Specify NULL to prevent the routine from returning this parameter. In this case the application does not call **rpc_string_free()**.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_binding_parse()** routine parses a string representation of a binding handle into its component fields.

The RPC runtime allocates memory for each component string the routine returns. The application calls **rpc_string_free()** once for each returned string to deallocate the memory for that string.

If any field of the *string_binding* field is empty, **rpc_string_binding_parse()** returns the empty string in the corresponding output parameter.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_invalid_string_binding
Invalid string binding.

Related Information

Functions: **rpc_binding_from_string_binding(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_string_binding_compose(3rpc)**, **rpc_string_free(3rpc)**, **uuid_from_string(3rpc)**.

rpc_string_free

Purpose

Frees a character string allocated by the runtime; used by client, server, or management applications

Synopsis

```
#include <dce/rpc.h>

void  rpc_string_free(
   unsigned_char_t **string
   unsigned32 *status);
```

Parameters

Input/Output

string Specifies the address of the pointer to the character string to free.

The value NULL is returned.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_string_free()** routine deallocates the memory occupied by a character string returned by the RPC runtime.

An application must call this routine once for each character string allocated and returned by calls to other RPC runtime routines. The names of these routines appear at the end of this reference page.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

Related Information

Functions: **dce_error_inq_text(3rpc)**, **rpc_binding_inq_auth_client(3rpc)**, **rpc_binding_inq_auth_info(3rpc)**, **rpc_binding_to_string_binding(3rpc)**, **rpc_mgmt_ep_elt_inq_next(3rpc)**, **rpc_mgmt_inq_server_princ_name(3rpc)**, **rpc_ns_binding_inq_entry_name(3rpc)**, **rpc_ns_entry_expand_name(3rpc)**,

rpc_ns_group_mbr_inq_next(3rpc), **rpc_ns_profile_elt_inq_next(3rpc)**, **rpc_string_binding_compose(3rpc)**, **rpc_string_binding_parse(3rpc)**, **uuid_to_string(3rpc)**.

rpc_tower_to_binding

Purpose

Returns a binding handle from a tower representation

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_to_binding(
   byte_p_t prot_tower
   rpc_binding_handle_t *binding
   unsigned32 *status);
```

Parameters

Input

prot_tower
Specifies a single protocol tower to convert to a binding handle.

Output

binding
Returns the server binding handle.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **rpc_tower_to_binding()** routine creates a server binding handle a canonical representation of a protocol tower.

When an application finishes using the *binding* parameter, the application calls the **rpc_binding_free()** routine to release the memory used by the binding handle.

The **rpc_intro(3rpc)** reference page contains an explanation of binding handles.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_invalid_arg
Invalid argument.

rpc_s_invalid_endpoint_format
Invalid endpoint format.

rpc_s_protseq_not_supported
Protocol sequence not supported on this host.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_vector_free(3rpc)**, **rpc_tower_vector_from_binding(3rpc)**.

rpc_tower_vector_free

Purpose

Releases memory associated with a tower vector

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_vector_free(
   rpc_tower_vector_p_t *twr_vector
   unsigned32 *status);
```

Parameters

Input

twr_vector
Specifies the tower vector to be freed. On return, its value is NULL.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The status code is either **rpc_s_ok** or a value returned from a called routine.

Description

The **rpc_tower_vector_free()** routine releases memory associated with a tower vector, including the towers as well as the vector.

Return Values

No value is returned.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_to_binding(3rpc)**, **rpc_tower_vector_from_binding(3rpc)**.

rpc_tower_vector_from_binding

Purpose

Creates a tower vector from a binding handle

Synopsis

```
#include <dce/rpc.h>

void rpc_tower_vector_from_binding(
   rpc_if_handle_t if_spec
   rpc_binding_handle_t binding
   rpc_tower_vector_p_t *twr_vector
   unsigned32 *status);
```

Parameters

Input

if_spec
The interface specification that will be combined with a binding handle to form a tower vector.

binding
The binding handle that will be combined with a interface specification to form a tower vector.

Output

twr_vector
Returns the allocated tower vector.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The status code is either **rpc_s_ok**, or **rpc_s_no_interfaces**, or a value returned from a called routine.

Description

The **rpc_tower_vector_from_binding()** routine creates a vector of towers from a binding handle. After the caller is finished with the tower vector, the **rpc_tower_vector_free()** routine must be called to release the memory used by the vector.

Return Values

No value is returned.

Related Information

Functions: **rpc_binding_copy(3rpc)**, **rpc_binding_free(3rpc)**, **rpc_tower_to_binding(3rpc)**, **rpc_tower_vector_free(3rpc)**.

uuid_compare

Purpose

Compares two UUIDs and determines their order; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

signed32  uuid_compare(
   uuid_t *uuid1
   uuid_t *uuid2
   unsigned32 *status);
```

Parameters

Input

uuid1 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid2*.

Use the value NULL to specify a nil UUID for this parameter.

uuid2 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid1*.

Use the value NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_compare()** routine compares two UUIDs and determines their order. A nil UUID is considered the first element in order. The order of UUIDs is defined by the RPC architecture and is not a temporal (related to time) ordering. Comparing two specific UUIDs always returns the same result regardless of the implementation or system architecture.

You can use this routine to sort data with UUIDs as a key.

Return Values

Returns one of the following constants:

−1 The *uuid1* parameter precedes the *uuid2* parameter in order.

0 The *uuid1* parameter is equal to the *uuid2* parameter in order.

1 The *uuid1* parameter follows the *uuid2* parameter in order.

Note that a value of 0 (zero) has the same meaning as if **uuid_equal** (*&uuid1*, *&uuid2*) returned a value of TRUE.

A nil UUID is the first UUID in order. This means the following:

- If *uuid1* is NULL and *uuid2* is nonnil, the routine returns -1.
- If *uuid1* is NULL and *uuid2* is NULL, the routine returns 0.
- If *uuid1* is nonnil and *uuid2* is NULL, the routine returns 1.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_equal(3rpc)**, **uuid_is_nil(3rpc)**.

uuid_create

Purpose

Creates a new UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_create(
   uuid_t *uuid
   unsigned32 *status);
```

Parameters

Input

None.

Output

uuid Returns the new UUID.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_create()** routine creates a new UUID.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_getconf_failure
Cannot get network interface device configuration.

uuid_s_no_address
Cannot get Ethernet hardware address.

uuid_s_socket_failure
Cannot create socket.

Related Information

Functions: **uuid_create_nil(3rpc)**, **uuid_from_string(3rpc)**, **uuid_to_string(3rpc)**.

uuid_create_nil

Purpose

Creates a nil UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_create_nil(
   uuid_t *nil_uuid
   unsigned32 *status);
```

Parameters

Input

None.

Output

nil_uuid
Returns a nil UUID.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_create_nil()** routine creates a nil UUID.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

Related Information

Functions: **uuid_create(3rpc)**.

uuid_equal

Purpose

Determines if two UUIDs are equal; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

boolean32 uuid_equal(
   uuid_t *uuid1
   uuid_t *uuid2
   unsigned32 *status);
```

Parameters

Input

uuid1 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid2*. Supply the value NULL to specify a nil UUID for this parameter.

uuid2 Specifies a pointer to a UUID. This UUID is compared with the UUID specified in *uuid1*. Supply the value NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_equal()** routine compares two UUIDs and determines if they are equal.

Return Values

The possible return values and their meanings are as follows:

TRUE The *uuid1* parameter is equal to the *uuid2* parameter. Parameter *status* contains the status code **uuid_s_ok**.

FALSE
The *uuid1* parameter is not equal to the *uuid2* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_compare(3rpc)**.

uuid_from_string

Purpose

Converts a string UUID to its binary representation; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_from_string(
   unsigned_char_t *string_uuid
   uuid_t *uuid
   unsigned32 *status);
```

Parameters

Input

string_uuid
Specifies a string representation of a UUID. Supply the value NULL or the null string (**\0**) to specify a nil UUID.

Output

uuid Returns the binary form of the UUID specified by the *string_uuid* parameter into the address specified by this parameter.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

An application calls the **uuid_from_string()** routine to convert a string UUID to its binary representation.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

uuid_s_invalid_string_uuid
Invalid format for a string UUID.

Related Information

Functions: **uuid_to_string(3rpc)**.

uuid_hash

Purpose

Creates a hash value for a UUID; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

unsigned16 uuid_hash(
   uuid_t *uuid
   unsigned32 *status);
```

Parameters

Input

uuid Specifies the UUID for which a hash value is created. Supply NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_hash()** routine generates a hash value for a specified UUID.

Note that the return value for a single *uuid* value may differ across platforms.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

Return Values

Returns a hash value for the specified UUID.

uuid_is_nil

Purpose

Determines if a UUID is nil; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

boolean32 uuid_is_nil(
   uuid_t *uuid
   unsigned32 *status);
```

Parameters

Input

uuid Specifies a UUID to test as a nil UUID. Supply NULL to specify a nil UUID for this parameter.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_is_nil()** routine determines whether the specified UUID is a nil UUID. This routine yields the same result as if an application did the following:

- Called the **uuid_create_nil()** routine.
- Called the **uuid_equal()** routine to compare the returned nil UUID to the UUID specified in the *uuid* parameter.

Return Values

The possible return values and their meanings are as follows:

TRUE The *uuid* parameter is a nil UUID. Parameter *status* contains the status code **uuid_s_ok**.

FALSE
The *uuid* parameter is not a nil UUID.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **uuid_compare(3rpc)**, **uuid_create_nil(3rpc)**, **uuid_equal(3rpc)**.

uuid_to_string

Purpose

Converts a UUID from a binary representation to a string representation; used by client, server, or management applications

Synopsis

```
#include <dce/uuid.h>

void uuid_to_string(
   uuid_t *uuid
   unsigned_char_t **string_uuid
   unsigned32 *status);
```

Parameters

Input

uuid Specifies a UUID in its binary format. Supply NULL to specify a nil UUID for this parameter.

Output

string_uuid
Returns a pointer to the string representation of the UUID specified in the *uuid* parameter. Specify NULL for this parameter to prevent the routine from returning this information.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **uuid_to_string()** routine converts a UUID from its binary representation to its string representation.

The RPC runtime allocates memory for the string returned in the *string_uuid* parameter. The application calls **rpc_string_free()** to deallocate that memory. It is not necessary to call **rpc_string_free()** when you supply NULL for the *string_uuid* parameter.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

uuid_s_ok
Success.

uuid_s_bad_version
Bad UUID version.

Related Information

Functions: **rpc_string_free(3rpc)**, **uuid_from_string(3rpc)**.

wchar_t_from_netcs

Purpose

Converts international character data from a network code set to a local code set prior to unmarshalling; used by client and server applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_from_netcs(
   rpc_binding_handle_t binding
   unsigned32 network_code_set_value
   idl_byte *network_data
   unsigned32 network_data_length
   unsigned32 local_buffer_size
   wchar_t *local_data
   unsigned32 *local_data_length
   error_status_t *status);
```

Parameters

Input

binding
: Specifies the target binding handle from which to obtain code set conversion information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next( )** or **rpc_ns_binding_select( )**routine.

network_code_set_value
: The registered hexadecimal integer value that represents the code set that was used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the receiving tag. When the caller is the server stub, this value is the sending tag.

network_data
: A pointer to the international character data that has been received, in the network code set encoding.

network_data_length
: The number of **idl_byte** data elements to be converted. For a varying array or a conformant varyihg array, the value is the local value of the **length_is** variable. For a conformant array, the value is the local value of the **size_is** variable. For a fixed array, the value is the array size specified in the interface definition.

local_buffer_size
: A pointer to the buffer size to be allocated to contain the converted data, in units of **wchar_t**. The value specified in this parameter is the local buffer size returned by the **wchar_t_local_size( )** routine.

Output

local_data
: A pointer to the converted data, in **wchar_t** format.

local_data_length
The length of the converted data, in units of **wchar_t**. NULL is specified if a fixed array or varying array is to be converted.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_from_netcs()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_from_netcs()** routine is one of the DCE RPC stub code set conversion routines that RPC stubs use before they marshall or unmarshall data to convert international character data to and from local and network code sets.

Client and server stubs call the **wchar_t_*_netcs** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application.

Client and server stubs call the **wchar_t_from_netcs()** routine before they unmarshall the international character data received from the network. The routine takes a binding handle, a code set value that identifies the code set used to transfer international character data over the network, the address of the network data, in **idl_byte** format, that may need to be converted, and the data length, in units of **idl_byte**.

The routine compares the sending code set to the local code set currently in use. If the routine finds that code set conversion is necessary, (because the local code set differs from the code set specified to be used on the network), it determines which host code set converter to call to convert the data and then invokes that converter.

The routine then returns the converted data, in **wchar_t** format. If the data is a conformant or conformant varying array, the routine also returns the length of the converted data, in units of **wchar_t**.

Prior to calling **wchar_t_from_netcs()**, client and server stubs call the **wchar_t_local_size()** routine to calculate the size of the buffer required to hold the converted data. Because **wchar_t_local_size()** cannot make this calculation for fixed and varying arrays, applications should either restrict use of **wchar_t_from_netcs()** to conformant and conformant varying arrays, or independently ensure that the buffer allocated for converted data is large enough.

Applications can specify local data types other than **cs_byte** and **wchar_t** (the local data types for which DCE RPC supplies stub code set conversion routines) with the **cs_char** ACF attribute. In this case, the application must also supply *local_type*_**to_netcs()** and *local_type*_**from_netcs()** stub conversion routines for this type.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain code set evaluation information. If this error occurs in the server stub, an exception is raised to the client application.

When the routine is running the host converter routines, the following errors can be returned:

- **rpc_s_ss_invalid_char_support**
- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, this routine calls the **dce_cs_loc_to_rgy()** routine and the host converter routines. If one of these routines returns an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_local_size

Purpose

Calculates the necessary buffer size for code set conversion from a network code set to a local code set prior to unmarshalling; used by client and server stubs, but not directly by applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_local_size(
   rpc_binding_handle_t binding
   unsigned32 network_code_set_value
   unsigned32 network_buffer_size
   idl_cs_convert_t *conversion_type
   unsigned32 *local_buffer_size
   error_status_t *status);
```

Parameters

Input

binding
: Specifies the target binding handle from which to obtain buffer size evaluation information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()** routine.

network_code_set_value
: The registered hexadecimal integer value that represents the code set used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the receiving tag. When the caller is the server stub, this value is the sending tag.

network_buffer_size
: The size, in units of **idl_byte**, of the buffer that is allocated for the international character data, For a conformant or conformant varying array, this value is the network value of the **size_is** variable for the array; that is, the value is the size of the unmarshalled string if no conversion is done.

Output

conversion_type
: A pointer to the enumerated type defined in **dce/idlbase.h** that indicates whether data conversion is necessary and whether or not the existing buffer is sufficient for storing the results of the conversion. Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, and **idl_byte** to **wchar_t** conversion always takes place, the conversion type returned is always **idl_cs_new_buffer_convert**.

local_buffer_size
: A pointer to the buffer size that needs to be allocated to contain the converted data, in units of **wchar_t**. This value is to be used as the local value of the **size_is** variable for the array, and is nonNULL only if a

conformant or conformant varying array is to be unmarshalled. A value of NULL in this parameter indicates that a fixed or varying array is to be unmarshalled.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_local_size()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_local_size()** routine is one of the four DCE RPC buffer sizing routines that RPC stubs use before they marshall or unmarshall data to determine whether or not the buffers allocated for code set conversion need to be enlarged to hold the converted data. The buffer sizing routines determine the type of conversion required and calculate the size of the necessary buffer (if a conformant or conformant varying array is to be marshalled or unmarshalled); the RPC stub then allocates a buffer of that size before it calls one of the code set conversion routines.

Client and server stubs call the two **wchar_t_*_size** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **wchar_t_local_size()** routine is used to evaluate buffer size requirements prior to unmarshalling data received over the network.

Applications do not call the **wchar_t_local_size()** routine directly. Client and server stubs call the routine before they unmarshall any data. The stubs pass the routine a binding handle and a code set value that identifies the code set that was used to transfer international character data over the network. The stubs also specify the network storage size of the data, in units of **idl_byte**.

Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, **wchar_t_local_size()** always sets *conversion_type* to **idl_cs_new_buffer_convert**, regardless of whether it is called from a client or server stub, or whether client and server code set tag information has been stored in the binding handle by a code sets evaluation or tag-setting routine. If a conformant or conformant varying array is to be unmarshalled, the routine then calculates a new buffer size by dividing the value of *network_buffer_size* by the number of bytes required to encode one **wchar_t** unit. The routine returns the new buffer size in the *local_buffer_size* argument. The size is specified in units of **wchar_t**, which is the local representation used for international character data in wide character format.

When a fixed or varying array is being unmarshalled, the **wchar_t_local_size()** routine cannot calculate the required buffer size and does not return a value in the *local_buffer_size* argument.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain the information necessary to evaluate the code set. If this error occurs in the server stub, an exception is raised to the client application.

When invoked from the server stub, this routine calls the routines **dce_cs_loc_to_rgy()** and **rpc_rgy_get_max_bytes()**. If either of these routines returns an error, the **wchar_t_local_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_net_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_net_size

Purpose

Calculates the necessary buffer size for code set conversion from a local code set to a network code set prior to marshalling; used by client and server stubs but not directly by applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_net_size(
   rpc_binding_handle_t binding
   unsigned32 network_code_set_value
   unsigned32 local_buffer_size
   idl_cs_convert_t *conversion_type
   unsigned32 *network_buffer_size
   error_status_t *status);
```

Parameters

Input

binding
: Specifies the target binding handle from which to obtain buffer size evaluation information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next()** or **rpc_ns_binding_select()** routine.

network_code_set_value
: The registered hexadecimal integer value that represents the code set to be used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the sending tag. When the caller is the server stub, this value is the receiving tag.

local_buffer_size
: The size, in units of **wchar_t**, of the buffer that is allocated for the international character data. For a conformant or conformant varying array, this value is the local value of the **size_is** variable for the array; that is, the value is the size of the marshalled string if no conversion is done.

Output

conversion_type
: A pointer to the enumerated type defined in **dce/idlbase.h** that indicates whether data conversion is necessary and whether or not the existing buffer is sufficient for storing the results of the conversion. Because **wchar_t** to **idl_byte** require different numbers of bytes to encode one character, and **wchar_t** to **idl_byte** conversion always takes place, the conversion type returned is always **idl_cs_new_buffer_convert**.

network_buffer_size
: A pointer to the buffer size that needs to be allocated to contain the converted data, in units of **idl_byte**. This value is to be used as the network value of the **size_is** variable for the array, and is non-NULL only if a

conformant or conformant varying array is to be marshalled. A value of NULL in this parameter indicates that a fixed or varying array is to be marshalled.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_net_size()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_net_size()** routine is one of the four DCE RPC buffer sizing routines that RPC stubs use before they marshall or unmarshall data to determine whether or not the buffers allocated for code set conversion need to be enlarged to hold the converted data. The buffer sizing routines determine the type of conversion required and calculate the size of the necessary buffer (if a conformant or conformant varying array is to be marshalled or unmarshalled); the RPC stub then allocates a buffer of that size before it calls one of the code set conversion routines.

Client and server stubs call the two **wchar_t_*_size** routines when the **wchar_t** type has been specified as the local data type using the **cs_char** attribute in the attribute configuration file for the application. The **wchar_t_net_size()** routine is used to evaluate buffer size requirements prior to marshalling data to be sent over the network.

Applications do not call the **wchar_t_net_size()** routine directly. Client and server stubs call the routine before they marshall any data. The stubs pass the routine a binding handle and a code set value that identifies the code set to be used to transfer international character data over the network. The stubs also specify the local storage size of the data, in units of **wchar_t**.

Because **wchar_t** and **idl_byte** require different numbers of bytes to encode one character, **wchar_t_net_size()** always sets *conversion_type* to **idl_cs_new_buffer_convert**, regardless of whether it is called from a client or server stub, or whether client and server code set tag information has been stored in the binding handle by a code sets evaluation or tag-setting routine. If a conformant or conformant varying array is to be marshalled, the routine then calculates a new buffer size by multiplying the value of *local_buffer_size* by the number of bytes required to encode one **wchar_t** unit. The routine returns the new buffer size in the *network_buffer_size* argument. The size is specified in units of **idl_byte**, which is the network representation used for international character data.

When a fixed or varying array is being marshalled, the **wchar_t_net_size()** routine cannot calculate the required buffer size and does not return a value in the *network_buffer_size* argument.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain the information necessary to evaluate the code set. If this error occurs in the server stub, an exception is raised to the client application.

When invoked from the server stub, this routine calls the routines **dcs_cs_loc_to_rgy()** and **rpc_rgy_get_max_bytes()**. If either of these routines returns an error, the **wchar_t_net_size()** routine raises an exception to the client application.

Related Information

Functions: **cs_byte_local_size(3rpc)**, **cs_byte_net_size(3rpc)**, **dcs_cs_loc_to_rgy(3rpc)**, **rpc_rgy_get_max_bytes(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_to_netcs(3rpc)**.

wchar_t_to_netcs

Purpose

Converts international character data from a local code set to a network code set prior to marshalling; used by client and server applications

Synopsis

```
#include <dce/codesets_stub.h>

void wchar_t_to_netcs(
   rpc_binding_handle_t binding
   unsigned32 network_code_set_value
   wchar_t *local_data
   unsigned32 local_data_length
   idl_byte *network_data
   unsigned32 *network_data_length
   error_status_t *status);
```

Parameters

Input

binding
Specifies the target binding handle from which to obtain code set conversion information. When called from the client stub, this value is the binding handle of a compatible server returned by the **rpc_ns_binding_import_next( )** or **rpc_ns_binding_select( )** routine.

network_code_set_value
The registered hexadecimal integer value that represents the code set to be used to transmit character data over the network. In general, the network code set is the code set that the client application's code sets evaluation routine has determined to be compatible for this client and server. When the caller is the client stub, this value is the sending tag. When the caller is the server stub, this value is the receiving tag.

local_data
A pointer to the international character data to be transmitted, in the local code set encoding.

local_data_length
The number of **wchar_t** data elements to be converted. For a varying array or a conformant varying array, this value is the local value of the **length_is** variable. For a conformant array, this value is the local value of the **size_is** variable. For a fixed array, the value is the array size specified in the interface definition.

Output

network_data
A pointer to the converted data, in **idl_byte** format.

network_data_length
A pointer to the length of the converted data, in units of **idl_byte**. NULL is specified if a fixed or varying array is to be converted.

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Description

The **wchar_t_to_netcs()** routine belongs to a set of DCE RPC routines for use by client and server applications that are transferring international character data in a heterogeneous character set and code sets environment.

The **wchar_t_to_netcs()** routine is one of the DCE RPC stub code set conversion routines that RPC stubs use before they marshall or unmarshall data to convert international character data to and from local and network code sets.

Client and server stubs call the **wchar_t_*_netcs()** routines when the **wchar_t** type has been specified as the local data type with the **cs_char** attribute in the attribute configuration file for the application.

Client and server stubs call the **wchar_t_to_netcs()** routine before they marshall any data. The routine takes a binding handle, a code set value that identifies the code set to be used to transfer international character data over the network, the address of the data to be converted, and the length of the data, in units of **wchar_t**.

The routine first converts the character data from **wchar_t** values to **idl_byte** values. The routine next compares the sending code set to the local code set currently in use. If the routine finds that code set conversion is necessary, (because the local code set differs from the code set specified to be used on the network), it determines which host code set converter to call to convert the data and then invokes that converter.

The routine then returns the converted data, in **idl_byte** format. If the data is a conformant or conformant varying array, the routine also returns the length of the converted data, in units of **idl_byte**.

Prior to calling **wchar_t_to_netcs()**, client and server stubs call the **wchar_t_net_size()** routine to calculate the size of the buffer required to hold the converted data. Because **wchar_t_net_size()** cannot make this calculation for fixed and varying arrays, applications should either restrict use of **wchar_t_to_netcs()** to conformant and conformant varying arrays, or independently ensure that the buffer allocated for converted data is large enough.

Applications can specify local data types other than **cs_byte** and **wchar_t** (the local data types for which DCE RPC supplies stub support routines for code set conversion) with the **cs_char** ACF attribute. In this case, the application must also supply *local_type*_**to_netcs()** and *local_type*_**from_netcs()** stub conversion routines for the application-defined local type.

Permissions Required

No permissions are required.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The binding handle does not contain code set evaluation information. If this error occurs in the server stub, an exception is raised to the client application.

When this routine is running the host converter routines, the following errors can be returned:

- **rpc_s_ss_invalid_char_input**
- **rpc_s_ss_short_conv_buffer**

When invoked from the server stub, this routine calls the **dce_cs_loc_to_rgy()** routine and host converter routines. If any of these routines returns an error, an exception is raised to the client application.

Related Information

Functions: **cs_byte_from_netcs(3rpc)**, **cs_byte_to_netcs(3rpc)**, **dce_cs_loc_to_rgy(3rpc)**, **wchar_t_from_netcs(3rpc)**, **wchar_t_local_size(3rpc)**, **wchar_t_net_size(3rpc)**,

Chapter 4. DCE Directory Service

xds_intro

Purpose

Introduction to X/OPEN Directory Services (XDS) functions

Synopsis

```
#include <xom.h>
 #include <xds.h>
 #include <xdsext.h>
```

Description

This **xds_intro** reference page lists the XDS interface functions in the following table. XDS provides a C language binding.

Table 32. Service Interface Functions—xds_intro(3xds)

Function	Description
dsX_extract_attr_values()	Extracts attribute values from an OM object.
ds_abandon()	Function not supported.
ds_add_entry()	Adds a leaf entry to the directory information tree (DIT).
ds_bind()	Opens a session with a directory user agent.
ds_compare()	Compares a purported attribute value with the attribute value stored in the directory for a particular entry.
ds_initialize()	Initializes the interface.
ds_list()	Enumerates the immediate subordinates of a particular directory entry.
ds_modify_entry()	Performs an atomic modification of a directory entry.
ds_modify_rdn()	Changes the relative distinguished name (RDN) of a leaf entry.
ds_read()	Queries information on a directory entry by name.
ds_receive_result()	Function partially supported.
ds_remove_entry()	Removes a leaf entry from the DIT.
ds_search()	Finds entries of interest in a portion of the DIT.
ds_shutdown()	Shuts down the interface.
ds_unbind()	Unbinds from a directory session.
ds_version()	Negotiates features of the interface and service.
gds_decode_alt_addr()	Used by DME applications for alternate address mapping.
gds_encode_alt_addr()	Used by DME applications for alternate address mapping.

The Distributed Computing Environment (DCE) XDS interface does not support asynchronous operations within the same thread. Thus, **ds_abandon()** is redundant. A **ds_abandon()** call returns with a **DS_C_ABANDON_FAILED** (**DS_E_TOO_LATE**) error. For **ds_receive_result()**, if there are any outstanding operations (when multiple threads issue XDS calls in parallel), this function returns **DS_SUCCESS** with the *completion_flag_return* parameter set to **DS_OUTSTANDING_OPERATIONS**.

If no XDS calls are outstanding, **ds_receive_result()** returns with *DS_status* set to **DS_SUCCESS**, and with the *completion_flag_return* parameter set to **DS_NO_OUTSTANDING_OPERATION**.

The following differences exist between Global Directory Service (GDS) and Cell Directory Service (CDS):

- All functions operate on the GDS namespace.

- CDS does not support the **ds_modify_rdn()** or **ds_search()**. If either of these two functions is attempted on CDS, the error message **DS_C_SERVICE_ERROR** is returned (**DS_E_UNWILLING_TO_PERFORM**).
- In CDS, no X.500 schema rules apply. There is
  - No concept of an object class.
  - No mandatory attributes for a given object.
  - No set of attributes expressly permitted for a given object.
  - No predefined definition of single and multivalued attributes.

The absence of these schema rules means that the usual errors, which are returned by GDS for breach of schema rules, are not returned by CDS.

The CDS naming DIT is modeled on a typical file system architecture, where directories are used for storing objects and directories can contain subdirectories. Leaf objects in the CDS DIT are similar to X.500 naming objects. However, subtree objects are called directories as in a file system directory. All new objects must be added to an existing directory. CDS directory objects cannot be added, removed, modified, or compared using the XDS programming interface.

In CDS, the naming attribute of an object is not stored in the object. Consequently, in CDS, **ds_read()** never returns this attribute. Note that the **ds_compare()** routine applied to this attribute returns with **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).

Notes

See the notes in the relevant reference page for function-specific differences.

XDS functions check for NULL pointers and will return an error. The pointers are only checked at the function interface. The check is only for NULL and not for validity. If NULL pointers are passed, this may result in an undetermined behavior.

decode_alt_addr

Purpose

Converts an alternate address attribute from internal GDS format to a structured format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

int  decode_alt_addr(
   const D2_str *in
   D2_alt_addr **out);
```

Parameters

in A pointer to a **D2_str** structure that contains the alternate address attribute in an internal GDS format.

Description

The **decode_alt_addr()** routine converts a linearized string that is stored in a structure **D2_str** into a structured alternate address format stored in a **D2_alt_addr** structure. This function is provided for use by DME applications. It converts an alternate address attribute from an internal GDS format (linear octet string) to a structured format for application usage.

in->d2_size contains the length of the encoded octet string.

in->d2_value is a pointer to the beginning of the encoded octet string.

The **decode_alt_addr()** routine allocates memory for the structured alternate address. The parameter (**out*) contains the address of the memory area that should later be freed by the application.

The **D2_alt_addr** structure contains one field **D2_str** for the address, followed by a structured field for the set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the beginning of the address (not zero-terminated). The second component of the **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To read a set of object identifiers, the address of the first **D2_obj_id** structure should be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be read.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means that **decode_alt_addr()** does no BER conversion for object identifiers.

Return Values

***out* A pointer to the structure **D2_alt_addr** that stores the alternate address attribute in a structured format.

int 0 if successful.

-1 if unsuccessful (**malloc()** failure).

Related Information

Functions: **encode_alt_addr(3xds)**.

dsX_extract_attr_values

Purpose

Extracts attribute values from an OM object

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <xdsext.h>

OM_return_code  dsX_extract_attr_values(
   OM_private_object object
   OM_object_identifier attribute_type
   OM_boolean local_strings
   OM_public_object *values
   OM_value_position *total_number);
```

Parameters

Input

object The private object from which the attribute values are to be extracted. Objects of type **DS_C_ATTRIBUTE_LIST** or **DS_C_ENTRY_INFO** are supported.

attribute_type
The attribute type from which the values are to extracted.

local_strings
Indicates if results should be converted to a local string format.

Output

values The *values* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS**. It points to a public object containing an array of OM descriptors with the extracted attribute values.

total_number
Contains the total number of attribute values that have been extracted.

Note that the total includes only the attribute descriptors in the *values* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **dsX_extract_attr_values()** routine is used to extract the attribute values associated with the specified attribute type from an OM object. The OM object must be of type **DS_C_ATTRIBUTE_LIST** or **DS_C_ENTRY_INFO**. It returns an object containing an array of OM descriptors.

Notes

The memory space for the *values* return parameter is allocated by **dsX_extract_attr_values()**. The calling application is responsible for releasing this memory with the **om_delete()** routine.

Return Values

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of possible error values that can be returned in *OM_return_code*. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

ds_add_entry

Purpose

Adds a leaf entry to the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_add_entry(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_object entry
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context
(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant. Note that **DS_DONT_DEREFERENCE_ALIASES** and **DS_SIZE_LIMIT** do not apply to this operation.

name (Object(**DS_C_NAME**)). The name of the entry to be added. The immediate superior of the new entry is determined by removing the last RDN component, which belongs to the new entry.

The immediate superior must exist in the same Directory Service Agent, or the function can fail with **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**). Any aliases in the name are *not* dereferenced.

entry (Object(**DS_C_ATTRIBUTE_LIST**)). The attribute information that, together with that from the RDN, constitutes the entry to be created. Note that an instance of OM class **DS_C_ENTRY_INFO** can be supplied as the value of this parameter, since OM class **DS_C_ENTRY_INFO** is a subclass of OM class **DS_C_ATTRIBUTE_LIST**.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_add_entry()** function adds a leaf entry to the directory. The entry can be either an object or an alias. The directory checks that the resulting entry conforms to the directory schema.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_add_entry()** operation:

- Only leaf objects (that is, objects that are not CDS directory objects) can be added to CDS through the XDS interface. In other words, the immediate superior of the new entry must exist.
- Only the **DS_A_COMMON_NAME** and **DS_A_MEMBER** attributes are valid for the **DS_O_GROUP_OF_NAMES** object in CDS.
- GDS-structured attribute types are not supported by CDS. If an attempt is made to add a GDS-structured attribute type to CDS, then it returns with a **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).

Since CDS does not implement the X.500 schema rules, some CDS objects may not contain mandatory attributes like object class and so on.

Return Values

DS_status
: **DS_SUCCESS** is returned if the entry was added; otherwise, an error is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

The **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**) error, referred to earlier in this reference page, need not be returned if there is local agreement between the DSAs to allow the entry to be added.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_bind

Purpose

Opens a session with the directory

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_bind(
   OM_object session
   OM_workspace workspace
   OM_private_object *bound_session_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). Specifies a particular directory service provider, together with other details of the service required. This parameter can be either a public object or a private object. The **DS_DEFAULT_SESSION** constant can also be used as the value of this parameter, causing a new session to be created with default values for all its OM attributes.

workspace
Specifies the workspace obtained from a call to **ds_initialize()** that is to be associated with the session. All function results from directory operations using this session will be returned as private objects in this workspace. If the *session* parameter is a private object, it must be a private object in this workspace.

Output

bound_session_return
(Object(**DS_C_SESSION**)). Upon successful completion, this parameter contains an instance of a directory session that can be used as a parameter to other functions (for example, **ds_read()**). This is a new private object if the value of the *session* parameter was **DS_DEFAULT_SESSION** or a public object; otherwise, it is that value supplied as a parameter. The function supplies default values for any of the OM attributes that are not present in the *session* parameter instance supplied as a parameter. It also sets the value of the **DS_FILE_DESCRIPTOR** OM attribute to **DS_NO_VALID_FILE_DESCRIPTOR**, since the functionality is not supported.

Description

The **ds_bind()** function sets up a communications link to the DSA.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ

between the two services. (See the **xds_intro(3xds)** reference page for XDS functions at the start of this chapter for general differences between operations on GDS and CDS.)

Note that in order to use CDS when GDS is not active, **ds_bind()** must be called with the value of the *session* parameter set to **DS_DEFAULT_SESSION**.

Return Values

DS_status
DS_SUCCESS is returned if the function is completed successfully; otherwise, it indicates the error that has occurred.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_SESSION**
- **DS_E_BAD_WORKSPACE**
- **DS_E_MISCELLANEOUS**
- **DS_E_NOT_SUPPORTED**
- **DS_E_TOO_MANY_SESSIONS**

The function can return the following directory errors:

- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

Related Information

Functions: **ds_unbind(3xds)**.

ds_compare

Purpose

Compares an attribute value with the attribute value stored in the directory for a particular entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_compare(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_object ava
   OM_private_object *result_return
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context
(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** does not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are dereferenced unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

ava (Object(**DS_C_AVA**)). The attribute value assertion that specifies the attribute type and value to be compared with those in the entry.

Output

result_return
(Object(**DS_C_COMPARE_RESULT**)). Upon successful completion, the result contains flags indicating whether the values matched and whether the comparison was made against the original entry. It also contains the DN of the target object if an alias is dereferenced.

invoke_id_return
(Integer). Not supported.

Description

The **ds_compare()** function compares the value supplied in the given *ava* parameter with the value or values of the same attribute type in the named entry.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_compare()** operation:

- In CDS, the naming attribute of an object is not stored in the attribute list of an object. Thus in CDS, a **ds_compare()** of the purported naming attribute value with the naming attribute value of the directory object always fails to match.
- GDS-structured types are not supported by CDS. If a GDS-structured attribute type is used as a parameter to **ds_compare()** on a CDS object, then it returns with the error **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**).
- In CDS, **ds_compare()** can only be used on leaf objects; otherwise, a **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**) is returned.
- In CDS, if the *name* parameter is a CDS soft link and the **Dont_Dereference_Aliases** context parameter is set to **TRUE**, the only allowed attribute for comparison is the **DS_A_ALIASED_OBJECT_NAME** attribute. This attribute is compared with the Distinguished Name of the soft link target.

Return Values

DS_status
: Indicates whether the comparison is completed or not. If successful, **DS_SUCCESS** is returned. Note that the operation fails and an error is returned either if the target object is not found or if it does not have an attribute of the required type.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_initialize

Purpose

Initializes the XDS interface

Synopsis

```
#include <xom.h>
#include <xds.h>

OM_workspace  ds_initializ(void);
```

Description

The **ds_initialize()** function performs any necessary initialization of the XDS application program interface (API), including the creation of a workspace. It must be called before any other directory interface functions are called. If it is subsequently called before **ds_shutdown()**, the function returns NULL.

Return Values

OM_workspace
Upon successful completion, **OM_workspace** contains a handle to a workspace in which OM objects can be created and manipulated. Objects created in this workspace, and only such objects, can be used as parameters to the other directory interface functions. This function returns NULL if it fails.

Related Information

Functions: **ds_shutdown(3xds)**.

ds_list

Purpose

Enumerates the immediate subordinates of a particular directory entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_list(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_private_object *result_return
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context
(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the object entry whose immediate subordinates are to be listed. Any aliases in the name are dereferenced unless this is prohibited by the service control attribute **DS_DONT_DEREFERENCE_ALIASES** of the **DS_C_CONTEXT** object.

Output

result_return
(Object(**DS_C_LIST_RESULT**)). Upon successful completion, the result contains some information about the target object's immediate subordinates. It also contains the DN of the target object, if an alias was dereferenced to find it. Aliases in the subordinate names are not dereferenced. In addition, there can be a partial outcome qualifier, which indicates that the result is incomplete. It also explains the reason for this (for example, because the time limit expired), and it contains information that can be helpful when attempting to complete the operation.

invoke_id_return
(Integer). Not supported.

Description

The **ds_list()** function is used to obtain a list of the immediate subordinates of the named entry. The list can be incomplete in some circumstances; for example, if the results exceed **DS_SIZE_LIMIT**.

Return Values

DS_status
Takes the value **DS_SUCCESS** if the named object is located (even if there are no subordinates) and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_modify_entry

Purpose

Performs an atomic modification on a directory entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_modify_entry(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_object changes
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context

(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name

(Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are *not* dereferenced.

changes
(Object(**DS_C_ENTRY_MOD_LIST**)). A sequence of modifications to the named entry.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_modify_entry()** routine is used to make a series of one or more of the following changes to a single directory entry:

- Add a new attribute (**DS_ADD_ATTRIBUTE**).
- Remove an attribute (**DS_REMOVE_ATTRIBUTE**).
- Add attribute values (**DS_ADD_VALUES**).
- Remove attribute values (**DS_REMOVE_VALUES**).

Values can be replaced by a combination of adding values and removing values in a single operation. The RDN can only be changed by using **ds_modify_rdn()**.

The result of the operation is as if each modification is made in the order specified in the *changes* parameter. If any of the individual modifications fails, then a **DS_C_ATTRIBUTE_ERROR** is reported and the entry is left as it was prior to the whole operation. The operation is atomic; that is, either all or none of the changes are made. The directory checks that the resulting entry conforms to the directory schema.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_modify_entry()** operation:

- Naming schema rules do not apply in CDS. Thus, the following attribute errors are never returned by CDS:
  - **DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**
  - **DS_E_ATTRIBUTE_OR_VALUE_EXISTS**
- Naming operations that would normally return these errors succeed in CDS. In particular, the addition of an attribute that already exists does not return with an error. Instead, the values of the attribute to be added are combined with the values of the existing attribute.
- GDS-structured attribute types are not supported by CDS. If a GDS-structured attribute type is used as a parameter to **ds_modify_entry()** on a CDS object, then it returns with a **DS_C_ATTRIBUTE_ERROR** (**DS_E_CONSTRAINT_VIOLATION**). In CDS, **ds_modify_entry()** can only be used on leaf objects; otherwise, a **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**) is returned.
- In CDS, if the *name* parameter is a CDS soft link and the **Dont_Dereference_Alias** flag is set to **TRUE**, the soft link entry itself is modified. In this case, the only allowed modifications are to the **DS_A_ALIASED_OBJECT_NAME** attribute.

Return Values

DS_status
: Takes the value **DS_SUCCESS** if all the modifications succeeded and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**

- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

The following situations apply to GDS:

- An attempt to use **DS_ADD_ATTRIBUTE** to add an existing attribute results in a **DS_C_ATTRIBUTE_ERROR**.
- An attempt to use **DS_ADD_VALUES** to add an existing value results in a **DS_C_ATTRIBUTE_ERROR**, as does an attempt to add a value to a nonexistent attribute type.
- An attempt to use **DS_REMOVE_ATTRIBUTE** to remove a nonexisting attribute results in a **DS_C_ATTRIBUTE_ERROR**, whereas an attempt to remove an attribute that is part of the object's RDN results in a **DS_C_UPDATE_ERROR**.
- An attempt to use **DS_REMOVE_VALUES** to remove a nonexisting value results in a **DS_C_ATTRIBUTE_ERROR**, whereas an attempt to remove a value of an attribute that is part of the object's RDN, or to modify the object class attribute, results in a **DS_C_UPDATE_ERROR**.

ds_modify_rdn

Purpose

Changes the RDN of a leaf entry

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_modify_rdn(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_object new_RDN
   OM_boolean delete_old_RDN
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context

(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name

(Object(**DS_C_NAME**)). The current name of the target leaf entry. Any aliases in the name are *not* dereferenced. The immediate superior must *not* have any nonspecific subordinate references; if it does, the function can fail with a **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**).

A nonspecific subordinate reference is an indication that another DSA holds some number of children, but does not indicate their RDNs. This means that it is not possible to check the uniqueness of the requested new RDN within a single DSA.

new_RDN
(Object(**DS_C_RELATIVE_NAME**)). The requested new RDN. If an attribute value in the new RDN does not already exist in the entry (either as part of the old RDN or as a nondistinguished value), the new value is added. If it cannot be added, an error is reported.

delete_old_RDN
(Boolean). If this value is **OM_TRUE**, all attribute values that are in the old RDN but not in the new RDN are deleted. If the value is **OM_FALSE**, the old values should remain in the entry (not as part of the RDN). The value must be **OM_TRUE** when a single value attribute in the RDN has its value changed by the operation. If this operation removes the last attribute value of an attribute, that attribute is deleted.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_modify_rdn()** function is used to change the RDN of a leaf entry (either an object entry or an alias entry).

Notes

CDS does not support **ds_modify_rdn()**, and returns with **DS_C_SERVICE_ERROR** (**DS_E_UNWILLING_TO_%PERFORM**).

Return Values

DS_status
Indicates whether the name of the entry is changed (**DS_SUCCESS** is returned); otherwise, an error is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

The **DS_C_UPDATE_ERROR** (**DS_E_AFFECTS_MULTIPLE_DSAS**) error, referred to earlier in this reference page, need not be returned if there is local agreement between the DSAs to allow the entry to be modified.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_read

Purpose

Queries information on an entry by name

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_read(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_object selection
   OM_private_object *result_return
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context
(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** does not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are dereferenced unless prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

selection
(Object(**DS_C_ENTRY_INFO_SELECTION**)). Specifies what information from the entry is requested. Information about no attributes, all attributes, or just for a named set can be chosen. Attribute types are always returned, but the attribute values need not be returned. The possible values of this parameter are given in the *OSF DCE Application Development Guide—Directory Services*.

Output

result_return
(Object(**DS_C_READ_RESULT**)). Upon successful completion, the result contains the DN of the target object, and a flag indicating whether the result came from the original entry or a copy, as well as any requested attribute types and values. Attribute information is only returned if access rights are sufficient.

invoke_id_return
(Integer). Not supported.

Description

The **ds_read()** function is used to extract information from an explicitly named entry. It can also be used to verify a DN.

Notes

Although the user ideally is not aware whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. (See the **xds_intro(3xds)** reference page for XDS functions for the general differences between operations on GDS and CDS.)

Note the following issues for the **ds_read()** operation:

- Since CDS does not implement the X.500 schema rules, some CDS objects may not contain mandatory attributes like object class and so on. In CDS, a read of an alias object fails if the **DS_A_ALIASED_OBJECT_NAME** does not exist. Instead, CDS returns with **DS_C_NAME_ERROR** (**DS_E_NO_SUCH_OBJECT**).
- In CDS, the naming attribute of an object is not stored in the attribute list for the object. Thus in CDS, **ds_read()** does not return this attribute in the attribute list for an object.

Return Values

DS_status
: Indicates whether or not the read operation is completed. This is **DS_SUCCESS** if completed.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_ATTRIBUTE**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

Note that the directory error **DS_C_ATTRIBUTE_ERROR** (**DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE**) is reported in GDS if an explicit list of attributes is specified by the *selection* parameter, but none of them are present in the entry. This error is not reported if any of the selected attributes are present.

A **DS_C_SECURITY_ERROR** (**DS_E_INSUFFICIENT_ACCESS_RIGHTS**) is only reported where access rights preclude the reading of all requested attribute values.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_remove_entry

Purpose

Removes a leaf entry from the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_remove_entry(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**). The directory session against which this operation is performed. This must be a private object.

context
(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. Note that **DS_SIZE_LIMIT** and **DS_DONT_DEREFERENCE_ALIASES** do not apply to this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name (Object(**DS_C_NAME**)). The name of the target object entry. Any aliases in the name are *not* dereferenced.

Output

invoke_id_return
(Integer). Not supported.

Description

The **ds_remove_entry()** function is used to remove a leaf entry from the directory (either an object entry or an alias entry).

Return Values

DS_status
Indicates whether or not the entry was deleted.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The function can return the following directory errors:

- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**
- **DS_C_UPDATE_ERROR**

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_search

Purpose

Finds entries of interest in a part of the DIT

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_search(
   OM_private_object session
   OM_private_object context
   OM_object name
   OM_sint subset
   OM_object filter
   OM_boolean search_aliases
   OM_object selection
   OM_private_object *result_return
   OM_sint *invoke_id_return);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session against which this operation is performed. This must be a private object.

context

(Object(**DS_C_CONTEXT**)). The directory context to be used for this operation. This parameter must be a private object or the **DS_DEFAULT_CONTEXT** constant.

name

(Object(**DS_C_NAME**)). The name of the object entry that forms the base of **ds_search()**. Any aliases in the name are dereferenced, unless dereferencing is prohibited by the **DS_DONT_DEREFERENCE_ALIASES** service control attribute of the **DS_C_CONTEXT** object.

subset (Integer). Specifies the portion of the DIT to be searched. Its value must be one of the following:

- **DS_BASE_OBJECT** Searches just the given object entry.
- **DS_ONE_LEVEL** Searches just the immediate subordinates of the given object entry.
- **DS_WHOLE_SUBTREE** Searches the given object and all its subordinates.

filter (Object(**DS_C_FILTER**)). The filter is used to eliminate entries from the search that are not wanted. Information is only returned on entries that satisfy the filter. The **DS_NO_FILTER** constant can be used as the value of this parameter if all entries are searched and none eliminated. This corresponds to a filter with a **DS_FILTER_TYPE** value of **DS_AND** and no values of the **DS_FILTER** or **DS_FILTER_ITEM** OM attributes.

search_aliases
(Boolean). Any aliases in the subordinate entries being searched are

dereferenced if the value of this parameter is **OM_TRUE**, and they are not dereferenced if its value is **OM_FALSE**.

selection
(Object(**DS_C_ENTRY_INFO_SELECTION**)). Specifies what information from the entry is requested. Information about no attributes, all attributes, or just for a named set can be chosen. Attribute types are always returned, but the attribute values need not be. The possible values of this parameter are listed in the *OSF DCE Application Development Guide—Directory Services*.

Output

result_return
(Object(**DS_C_SEARCH_RESULT**)). If completion is successful, the result contains the requested information from each object in the search space that satisfied the filter. The DN of the target object is present if an alias is dereferenced. In addition, there may be a partial outcome qualifier, which indicates that the result is incomplete. It also explains why it is not complete and how it could be completed.

invoke_id_return
(Integer). Not supported.

Description

The **ds_search()** function is used to search a portion of the directory and return selected information from entries of interest. The information may be incomplete in some circumstances; for example, if the results exceed **DS_SIZE_LIMIT**.

Notes

CDS does not support **ds_search()**, and it returns with **DS_C_SERVICE_ERROR** (**DS_E_UNWILLING_TO_PERFORM**).

Return Values

DS_status
Takes the value **DS_SUCCESS** if the named object is located and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_ARGUMENT**
- **DS_E_BAD_CONTEXT**
- **DS_E_BAD_NAME**
- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**
- **DS_E_MISSING_TYPE**
- **DS_E_TOO_MANY_OPERATIONS**

The following directory errors can be returned by the function:

- **DS_C_ATTRIBUTE_ERROR**
- **DS_C_NAME_ERROR**
- **DS_C_REFERRAL**
- **DS_C_SECURITY_ERROR**
- **DS_C_SERVICE_ERROR**

Note that an unfiltered search of just the base object succeeds even if none of the requested attributes are found, while the **ds_read()** call fails with the same selected attributes.

A **DS_C_SECURITY_ERROR** (**DS_E_INSUFFICIENT_ACCESS_RIGHTS**) is only reported where access rights preclude the reading of all requested attribute values.

This function can return a **DS_C_COMMUNICATIONS_ERROR**, as well as the error constant **DS_NO_WORKSPACE**.

ds_shutdown

Purpose

Deletes a directory workspace

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_shutdown(
   OM_workspace workspace);
```

Parameters

Input

workspace
Specifies the workspace (obtained from a call to **ds_initialize()**) that is to be deleted.

Description

The **ds_shutdown()** function deletes the workspace established by **ds_initialize()** and enables the service to release resources. All sessions associated with the workspace must be terminated by calling **ds_unbind()** prior to calling **ds_shutdown()**. No other directory function can reference the specified workspace after it has been deleted. However, **om_delete()** and **om_instance()** may be called if referring to public objects.

Return Values

DS_status
DS_SUCCESS if the function completed successfully; otherwise, it indicates the error that has occurred.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SERVICE_ERROR** (value **DS_E_BUSY**) if **ds_shutdown()** is called before all directory connections have been released with **ds_unbind()**.

This function can return the error constant **DS_NO_WORKSPACE**.

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors.

Related Information

Functions: **ds_initialize(3xds)**.

ds_unbind

Purpose

Unbinds from a directory session

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_unbind(
   OM_private_object session);
```

Parameters

Input

session
(Object(**DS_C_SESSION**)). The directory session to be unbound. This must be a private object. The value of the **DS_FILE_DESCRIPTOR** OM attribute is **DS_NO_VALID_FILE_DESCRIPTOR** if the function succeeds. The remaining OM attributes are unchanged.

Description

The **ds_unbind()** function terminates the given directory session and makes the parameter unavailable for use with other interface functions (except **ds_bind()**).

The unbound session can be used again as a parameter to **ds_bind()** possibly after modification by the OM functions. When it is no longer required, it must be deleted by using the OM functions.

Return Values

DS_status
Takes the value **DS_SUCCESS** if the *session* parameter is unbound and takes an error value if not.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or one of the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_SESSION**
- **DS_E_MISCELLANEOUS**

If **ds_unbind()** is called while there are outstanding directory operations (from other threads), then this function will return a **DS_SERVICE_ERROR** with the value **DS_E_BUSY**.

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors. However, this function can return the error constant **DS_NO_WORKSPACE**.

Related Information

Functions: **ds_bind(3xds)**.

ds_version

Purpose

Negotiates features of the interface and service

Synopsis

```
#include <xom.h>
#include <xds.h>

DS_status  ds_version(
   DS_feature feature_list[ ]
   OM_workspace workspace);
```

Parameters

Input

workspace

Specifies the workspace obtained from a call to **om_initialize()** for which the features are to be negotiated. The features will be in effect for operations that use the workspace or directory sessions associated with the workspace.

Input/Output

*feature_list***[ ]**

(**DS_feature**). On input contains an ordered sequence of features, each represented by an object identifier. The sequence is terminated by an object identifier having no components (a length of 0 (zero) and any value for the data pointer.)

If the function completed successfully, an ordered sequence of boolean values are returned, with the same number of elements as the *feature_list*[] parameter. If **OM_TRUE**, each value indicates that the corresponding feature is now part of the interface. If **OM_FALSE**, each value indicates that the corresponding feature is not available.

This result is combined with the *feature_list*[ ] parameter as a single array of structures of type **DS_feature**, which is defined as follows:

```
typedef struct
 {
 OM_object_identifier feature;
 OM_boolean      activated;
 }
 DS_feature;
```

Description

The **ds_version()** function negotiates features of the interface, which are represented by object identifiers. The **DS_BASIC_DIR_CONTENTS_PKG**, **DS_STRONG_AUTHENT_PKG**, and the **MHS_DIR_USER_PKG** specified in the *OSF DCE Application Development Guide—Directory Services*are negotiable features in this specification. Features can also include vendor extensions, such as the **DSX_GDS_PKG**, and new features in future versions of the XDS specification. Versions are negotiated after a workspace is initialized with **ds_initialize()**.

Return Values

DS_status
Takes the value **DS_SUCCESS** if the function completed successfully.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

This function can return a **DS_C_SYSTEM_ERROR** or the following **DS_C_LIBRARY_ERROR** errors:

- **DS_E_BAD_WORKSPACE**
- **DS_E_MISCELLANEOUS**

This function does not return a **DS_C_COMMUNICATIONS_ERROR** or any directory errors. However, this function can return the error constant **DS_NO_WORKSPACE**.

encode_alt_addr

Purpose

Converts an alternate address attribute structure into an internal GDS format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

int  encode_alt_addr(
   const D2_alt_addr *in
   D2_str **out);
```

Parameters

in A pointer to an alternate address attribute in a structured format.

Description

The **encode_alt_addr()** converts an alternate address stored in a **D2_alt_addr** structure into a linearized string that is stored in a structure of type **D2_str** . This function is provided for use by DME applications. It converts a structured alternate address attribute into a linear octet string for internal use by GDS.

The **D2_alt_addr** structure contains one field of type **D2_str** for storing the address, followed by a structured field for a set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the start of the address (not zero-terminated). The second component of **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To store additional object identifiers, the address of the first **D2_obj_id** structure has to be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be added.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means there is no BER conversion done by **encode_alt_addr()**.

encode_alt_addr() will allocate memory for the encoded string. (**out*) contains the address of the memory area that should later be freed by the application.

Return Values

***out* A pointer to the structure **D2_str** which stores the alternate address attribute in an internal GDS format.

*(*out)->d2_size* will contain the length of the encoded octet string.

*(*out)->d2_value* will be a pointer to the beginning of the encoded octet string. This string is not zero-terminated.

int

0 If successful.

–1 If unsuccessful (**malloc()** failure).

Related Information

Functions: **decode_alt_addr(3xds)**.

gds_decode_alt_addr

Purpose

Converts an alternate address attribute from internal GDS format to a structured format

Synopsis

```
#include <xom.h>
#include <xds.h>
#include <dce/d2dir.h>

d2_ret_val  gds_decode_alt_addr(
   const D2_str *in
   D2_alt_addr **out);
```

Parameters

Input

in A pointer to a **D2_str** structure that contains the alternate address attribute in an internal GDS format.

Output

out A pointer to the structure **D2_alt_addr** that stores the alternate address attribute in a structured format.

Description

The **gds_decode_alt_addr()** function converts a linearized string that is stored in a structure **D2_str** into a structured alternate address format stored in a **D2_alt_addr** structure. This function is provided for use by DME applications. It converts an alternate address attribute from an internal GDS format (linear octet string) to a structured format for application usage.

The *in->d2_size* parameter contains the length of the encoded octet string;*in->d2_value* is a pointer to the beginning of the encoded octet string.

The **gds_decode_alt_addr()** function allocates memory for the structured alternate address. The *(*out)* parameter contains the address of the memory area that should later be freed by the application.

The **D2_alt_addr** structure contains one field **D2_str** for the address, followed by a structured field for the set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the beginning of the address (not zero-terminated). The second component of the **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To read a set of object identifiers, the address of the first **D2_obj_id** structure should be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be read.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means that **gds_decode_alt_addr()** does no BER conversion for object identifiers.

Return Values

d2_ret_val

D2_NOERROR (that is, 0) if successful.

D2_ERROR (that is, -1), if unsuccessful (**malloc()** failure).

Related Information

Functions: **gds_encode_alt_addr(3xds)**.

gds_encode_alt_addr

Purpose

Converts an alternate address attribute structure into an internal GDS format

Synopsis

```
#include <xom.h>#include <xds.h>#include <dce/d2dir.h>

d2_ret_val  gds_encode_alt_addr(
   const D2_alt_addr *in
   D2_str **out);
```

Parameters

Input

in A pointer to an alternate address attribute in a structured format.

Output

out A pointer to the structure **D2_str** that stores the alternate address attribute in an internal GDS format.

The *(*out)->d2_size* parameter will contain the length of the encoded octet string; the *(*out)->d2_value* parameter will be a pointer to the beginning of the encoded octet string. This string is not zero-terminated.

Description

The **gds_encode_alt_addr()** function converts an alternate address stored in a **D2_alt_addr** structure into a linearized string that is stored in a structure of type **D2_str**. This function is provided for use by DME applications. It converts a structured alternate address attribute into a linear octet string for internal use by GDS.

The **D2_alt_addr** structure contains one field of type **D2_str** for storing the address, followed by a structured field for a set of object identifiers. The structure **D2_str** consists of the length of the address and a pointer to the start of the address (not zero-terminated). The second component of **D2_alt_addr** contains the number of object identifiers and the address of the first **D2_obj_id** structure. To store additional object identifiers, the address of the first **D2_obj_id** structure has to be increased by **sizeof(D2_obj_id)** bytes for each object identifier to be added.

The structure **D2_obj_id** consists of the length of the object identifier and a pointer to the beginning of the object identifier (not zero-terminated). Each object identifier is treated as an octet string; that means there is no BER conversion done by **gds_encode_alt_addr()**.

The **gds_encode_alt_addr()** function will allocate memory for the encoded string. The (**out*) parameter contains the address of the memory area that should later be freed by the application.

Return Values

d2_ret_val

D2_NOERROR (that is, 0), if successful.

D2_ERROR (that is, -1), if unsuccessful (**malloc()** failure).

Related Information

Functions: **gds_decode_alt_addr(3xds)**.

xds_intro

Purpose

Introduction to XDS header files

Description

There are nine XDS headers, as follows:

xds.h Contains definitions for the XDS functions and directory service package.

xdsbdcp.h
Contains definitions for the basic directory contents package.

xdssap.h
Contains definitions for the strong authentication package.

xdscds.h
Contains definitions for the cell directory service.

xdsdme.h
Contains definitions for the DME specific directory object and attribute.

xdsgds.h
Contains definitions for the global directory service package.

xdsmdup.h
Contains definitions for the MHS directory user package.

xmhp.h
Contains definitions for the MHS directory objects/attributes.

xmsga.h
Contains definitions for the message store general attributes.

The **xds.h** header file is a mandatory include for all applications using the XDS API.

The **xdsbdcp.h**,**xdsmdup.h**, and **xdssap.h** headers are part of the X/Open XDS specifications. They are required when using the basic directory contents package, MHS directory user package, and strong authentication package respectively.

The **xdsgds.h** and **xdscds.h** headers are DCE extensions to the XDS API. The **xdsgds.h** header is required when using the GDS package. The **xdscds.h** header is required when using CDS.

The **xmhp.h** and **xmsga.h** headers are required when using the MHS directory user package.

The **xdsdme.h** header is required when using the DME specific directory object class and attribute.

xds.h

Purpose

Definitions for the directory service package

Synopsis

```
#include  <xom.h>
#include  <xds.h>
```

Description

The **xds.h** header declares the interface functions, the structures passed to and from those functions, and the defined constants used by the functions and structures.

All application programs that include this header must first include the **xom.h** object management header.

```
#ifndef XDS_HEADER
#define XDS_HEADER

/* DS package object identifier */

/* { iso(1) identified-organization(3) icd-ecma(12)
   member-company(2) dec(1011) xopen(28) dsp(0) } */

#define OMP_O_DS_SERVICE_PKG    "\x2B\x0C\x02\x87\x73\x1C\x00"

/*Defined constants */

/* Intermediate object identifier macro */

#define dsP_c(X)    OMP_O_DS_SERVICE_PKG #X


/* OM class names (prefixed by DS_C_) */

/* Every application program which makes use of a class or other */
/* Object Identifier must explicitly import it into every    */
/* compilation unit (C source program) which uses it. Each such */
/* class or Object Identifier name must be explicitly exported  */
/* from just one compilation unit.                */

/* In the header file, OM class constants are prefixed with the */
/* OMP_O prefix to denote that they are OM classes. However,  */
/* when using the OM_IMPORT and OM_EXPORT macros, the base    */
/* names (without the OMP_O prefix) should be used.       */
/* For example:                          */
/*    OM_IMPORT (DS_C_AVA)                 */


#define OMP_O_DS_C_ABANDON_FAILED          dsP_c(\x85\x3D)
#define OMP_O_DS_C_ACCESS_POINT            dsP_c(\x85\x3E)
#define OMP_O_DS_C_ADDRESS                 dsP_c(\x85\x3F)
#define OMP_O_DS_C_ATTRIBUTE               dsP_c(\x85\x40)
#define OMP_O_DS_C_ATTRIBUTE_ERROR         dsP_c(\x85\x41)
#define OMP_O_DS_C_ATTRIBUTE_LIST          dsP_c(\x85\x42)
#define OMP_O_DS_C_ATTRIBUTE_PROBLEM       dsP_c(\x85\x43)
#define OMP_O_DS_C_AVA                     dsP_c(\x85\x44)
```

```
#define OMP_O_DS_C_COMMON_RESULTS          dsP_c(\x85\x45)
#define OMP_O_DS_C_COMMUNICATIONS_ERROR    dsP_c(\x85\x46)
#define OMP_O_DS_C_COMPARE_RESULT          dsP_c(\x85\x47)
#define OMP_O_DS_C_CONTEXT                 dsP_c(\x85\x48)
#define OMP_O_DS_C_CONTINUATION_REF        dsP_c(\x85\x49)
#define OMP_O_DS_C_DS_DN                   dsP_c(\x85\x4A)
#define OMP_O_DS_C_DS_RDN                  dsP_c(\x85\x4B)
#define OMP_O_DS_C_ENTRY_INFO              dsP_c(\x85\x4C)
#define OMP_O_DS_C_ENTRY_INFO_SELECTION    dsP_c(\x85\x4D)
#define OMP_O_DS_C_ENTRY_MOD               dsP_c(\x85\x4E)
#define OMP_O_DS_C_ENTRY_MOD_LIST          dsP_c(\x85\x4F)
#define OMP_O_DS_C_ERROR                   dsP_c(\x85\x50)
#define OMP_O_DS_C_EXT                     dsP_c(\x85\x51)
#define OMP_O_DS_C_FILTER                  dsP_c(\x85\x52)
#define OMP_O_DS_C_FILTER_ITEM             dsP_c(\x85\x53)
#define OMP_O_DS_C_LIBRARY_ERROR           dsP_c(\x85\x54)
#define OMP_O_DS_C_LIST_INFO               dsP_c(\x85\x55)
#define OMP_O_DS_C_LIST_INFO_ITEM          dsP_c(\x85\x56)
#define OMP_O_DS_C_LIST_RESULT             dsP_c(\x85\x57)
#define OMP_O_DS_C_NAME                    dsP_c(\x85\x58)
#define OMP_O_DS_C_NAME_ERROR              dsP_c(\x85\x59)
#define OMP_O_DS_C_OPERATION_PROGRESS      dsP_c(\x85\x5A)
#define OMP_O_DS_C_PARTIAL_OUTCOME_QUAL    dsP_c(\x85\x5B)
#define OMP_O_DS_C_PRESENTATION_ADDRESS    dsP_c(\x85\x5C)
#define OMP_O_DS_C_READ_RESULT             dsP_c(\x85\x5D)
#define OMP_O_DS_C_REFERRAL                dsP_c(\x85\x5E)
#define OMP_O_DS_C_RELATIVE_NAME           dsP_c(\x85\x5F)
#define OMP_O_DS_C_SEARCH_INFO             dsP_c(\x85\x60)
#define OMP_O_DS_C_SEARCH_RESULT           dsP_c(\x85\x61)
#define OMP_O_DS_C_SECURITY_ERROR          dsP_c(\x85\x62)
#define OMP_O_DS_C_SERVICE_ERROR           dsP_c(\x85\x63)
#define OMP_O_DS_C_SESSION                 dsP_c(\x85\x64)
#define OMP_O_DS_C_SYSTEM_ERROR            dsP_c(\x85\x65)
#define OMP_O_DS_C_UPDATE_ERROR            dsP_c(\x85\x66)


/* OM attribute names */

#define DS_ACCESS_POINTS                ((OM_type) 701)
#define DS_ADDRESS                      ((OM_type) 702)
#define DS_AE_TITLE                     ((OM_type) 703)
#define DS_ALIASED_RDNS                 ((OM_type) 704)
#define DS_ALIAS_DEREFERENCED           ((OM_type) 705)
#define DS_ALIAS_ENTRY                  ((OM_type) 706)
#define DS_ALL_ATTRIBUTES               ((OM_type) 707)
#define DS_ASYNCHRONOUS                 ((OM_type) 708)
#define DS_ATTRIBUTES                   ((OM_type) 709)
#define DS_ATTRIBUTES_SELECTED          ((OM_type) 710)
#define DS_ATTRIBUTE_TYPE               ((OM_type) 711)
#define DS_ATTRIBUTE_VALUE              ((OM_type) 712)
#define DS_ATTRIBUTE_VALUES             ((OM_type) 713)
#define DS_AUTOMATIC_CONTINUATION       ((OM_type) 714)
#define DS_AVAS                         ((OM_type) 715)
#define DS_CHAINING_PROHIB              ((OM_type) 716)
#define DS_CHANGES                      ((OM_type) 717)
#define DS_CRIT                         ((OM_type) 718)
#define DS_DONT_DEREFERENCE_ALIASES     ((OM_type) 719)
#define DS_DONT_USE_COPY                ((OM_type) 720)
#define DS_DSA_ADDRESS                  ((OM_type) 721)
#define DS_DSA_NAME                     ((OM_type) 722)
#define DS_ENTRIES                      ((OM_type) 723)
#define DS_ENTRY                        ((OM_type) 724)
#define DS_EXT                          ((OM_type) 725)
#define DS_FILE_DESCRIPTOR              ((OM_type) 726)
#define DS_FILTERS                      ((OM_type) 727)
#define DS_FILTER_ITEMS                 ((OM_type) 728)
#define DS_FILTER_ITEM_TYPE             ((OM_type) 729)
```

```
#define DS_FILTER_TYPE                  ((OM_type) 730)
#define DS_FINAL_SUBSTRING              ((OM_type) 731)
#define DS_FROM_ENTRY                   ((OM_type) 732)
#define DS_IDENT                        ((OM_type) 733)
#define DS_INFO_TYPE                    ((OM_type) 734)
#define DS_INITIAL_SUBSTRING            ((OM_type) 735)
#define DS_ITEM_PARAMETERS              ((OM_type) 736)
#define DS_LIMIT_PROBLEM                ((OM_type) 737)
#define DS_LIST_INFO                    ((OM_type) 738)
#define DS_LOCAL_SCOPE                  ((OM_type) 739)
#define DS_MATCHED                      ((OM_type) 740)
#define DS_MOD_TYPE                     ((OM_type) 741)
#define DS_NAME_RESOLUTION_PHASE        ((OM_type) 742)
#define DS_NEXT_RDN_TO_BE_RESOLVED      ((OM_type) 743)
#define DS_N_ADDRESSES                  ((OM_type) 744)
#define DS_OBJECT_NAME                  ((OM_type) 745)
#define DS_OPERATION_PROGRESS           ((OM_type) 746)
#define DS_PARTIAL_OUTCOME_QUAL         ((OM_type) 747)
#define DS_PERFORMER                    ((OM_type) 748)
#define DS_PREFER_CHAINING              ((OM_type) 749)
#define DS_PRIORITY                     ((OM_type) 750)
#define DS_PROBLEM                      ((OM_type) 751)
#define DS_PROBLEMS                     ((OM_type) 752)
#define DS_P_SELECTOR                   ((OM_type) 753)
#define DS_RDN                          ((OM_type) 754)
#define DS_RDNS                         ((OM_type) 755)
#define DS_RDNS_RESOLVED                ((OM_type) 756)
#define DS_REQUESTOR                    ((OM_type) 757)
#define DS_SCOPE_OF_REFERRAL            ((OM_type) 758)
#define DS_SEARCH_INFO                  ((OM_type) 759)
#define DS_SIZE_LIMIT                   ((OM_type) 760)
#define DS_SUBORDINATES                 ((OM_type) 761)
#define DS_S_SELECTOR                   ((OM_type) 762)
#define DS_TARGET_OBJECT                ((OM_type) 763)
#define DS_TIME_LIMIT                   ((OM_type) 764)
#define DS_T_SELECTOR                   ((OM_type) 765)
#define DS_UNAVAILABLE_CRIT_EXT         ((OM_type) 766)
#define DS_UNCORRELATED_LIST_INFO       ((OM_type) 767)
#define DS_UNCORRELATED_SEARCH_INFO     ((OM_type) 768)
#define DS_UNEXPLORED                   ((OM_type) 769)


/* DS_Filter_Item_Type: */

enum DS_Filter_Item_Type {
    DS_EQUALITY          = 0,
    DS_SUBSTRINGS        = 1,
    DS_GREATER_OR_EQUAL  = 2,
    DS_LESS_OR_EQUAL     = 3,
    DS_PRESENT           = 4,
    DS_APPROXIMATE_MATCH = 5
};


/* DS_Filter_Type: */

enum DS_Filter_Type {
    DS_ITEM = 0,
    DS_AND  = 1,
    DS_OR   = 2,
    DS_NOT  = 3
};


/* DS_Information_Type: */

enum DS_Information_Type {
```

```
    DS_TYPES_ONLY       = 0,
    DS_TYPES_AND_VALUES = 1
};


/* DS_Limit_Problem: */

enum DS_Limit_Problem {
    DS_NO_LIMIT_EXCEEDED        = -1,
    DS_TIME_LIMIT_EXCEEDED      = 0,
    DS_SIZE_LIMIT_EXCEEDED      = 1,
    DS_ADMIN_LIMIT_EXCEEDED     = 2
};


/* DS_Modification_Type: */

enum DS_Modification_Type {
    DS_ADD_ATTRIBUTE    = 0,
    DS_REMOVE_ATTRIBUTE = 1,
    DS_ADD_VALUES       = 2,
    DS_REMOVE_VALUES    = 3
};


/* DS_Name_Resolution_Phase: */

enum DS_Name_Resolution_Phase {
    DS_NOT_STARTED = 1,
    DS_PROCEEDING  = 2,
    DS_COMPLETED   = 3
};


/* DS_Priority: */

enum DS_Priority {
    DS_LOW    = 0,
    DS_MEDIUM = 1,
    DS_HIGH   = 2
};


/* DS_Problem: */

enum DS_Problem {
DS_E_ADMIN_LIMIT_EXCEEDED           = 1,
DS_E_AFFECTS_MULTIPLE_DSAS          = 2,
DS_E_ALIAS_DEREFERENCING_PROBLEM    = 3,
DS_E_ALIAS_PROBLEM                  = 4,
DS_E_ATTRIBUTE_OR_VALUE_EXISTS      = 5,
DS_E_BAD_ARGUMENT                   = 6,
DS_E_BAD_CLASS                      = 7,
DS_E_BAD_CONTEXT                    = 8,
DS_E_BAD_NAME                       = 9,
DS_E_BAD_SESSION                    = 10,
DS_E_BAD_WORKSPACE                  = 11,
DS_E_BUSY                           = 12,
DS_E_CANNOT_ABANDON                 = 13,
DS_E_CHAINING_REQUIRED              = 14,
DS_E_COMMUNICATIONS_PROBLEM         = 15,
DS_E_CONSTRAINT_VIOLATION           = 16,
DS_E_DIT_ERROR                      = 17,
DS_E_ENTRY_EXISTS                   = 18,
DS_E_INAPPROP_AUTHENTICATION        = 19,
DS_E_INAPPROP_MATCHING              = 20,
DS_E_INSUFFICIENT_ACCESS_RIGHTS     = 21,
```

```
DS_E_INVALID_ATTRIBUTE_SYNTAX       = 22,
DS_E_INVALID_ATTRIBUTE_VALUE        = 23,
DS_E_INVALID_CREDENTIALS            = 24,
DS_E_INVALID_REF                    = 25,
DS_E_INVALID_SIGNATURE              = 26,
DS_E_LOOP_DETECTED                  = 27,
DS_E_MISCELLANEOUS                  = 28,
DS_E_MISSING_TYPE                   = 29,
DS_E_MIXED_SYNCHRONOUS              = 30,
DS_E_NAMING_VIOLATION               = 31,
DS_E_NO_INFO                        = 32,
DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE     = 33,
DS_E_NO_SUCH_OBJECT                 = 34,
DS_E_NO_SUCH_OPERATION              = 35,
DS_E_NOT_ALLOWED_ON_NON_LEAF        = 36,
DS_E_NOT_ALLOWED_ON_RDN             = 37,
DS_E_NOT_SUPPORTED                  = 38,
DS_E_OBJECT_CLASS_MOD_PROHIB        = 39,
DS_E_OBJECT_CLASS_VIOLATION         = 40,
DS_E_OUT_OF_SCOPE                   = 41,
DS_E_PROTECTION_REQUIRED            = 42,
DS_E_TIME_LIMIT_EXCEEDED            = 43,
DS_E_TOO_LATE                       = 44,
DS_E_TOO_MANY_OPERATIONS            = 45,
DS_E_TOO_MANY_SESSIONS              = 46,
DS_E_UNABLE_TO_PROCEED              = 47,
DS_E_UNAVAILABLE                    = 48,
DS_E_UNAVAILABLE_CRIT_EXT           = 49,
DS_E_UNDEFINED_ATTRIBUTE_TYPE       = 50,
DS_E_UNWILLING_TO_PERFORM           = 51
};


/* DS_Scope_Of_Referral: */

enum DS_Scope_Of_Referral {
    DS_DMD     = 0,
    DS_COUNTRY = 1
};


/* Typedefs */

typedef OM_private_object DS_status;

typedef struct
{
  OM_object_identifier feature;
  OM_boolean      activated;
} DS_feature;


/* OM_object constants */

#define DS_DEFAULT_CONTEXT             ((OM_object) 0)
#define DS_DEFAULT_SESSION             ((OM_object) 0)
#define DS_OPERATION_NOT_STARTED       ((OM_object) 0)
#define DS_NO_FILTER                   ((OM_object) 0)
#define DS_NULL_RESULT                 ((OM_object) 0)
#define DS_SELECT_ALL_TYPES            ((OM_object) 1)
#define DS_SELECT_ALL_TYPES_AND_VALUES ((OM_object) 2)
#define DS_SELECT_NO_ATTRIBUTES        ((OM_object) 0)
#define DS_SUCCESS                     ((DS_status) 0)
#define DS_NO_WORKSPACE                ((DS_status) 1)


/* ds_search() subset */
```

```
#define DS_BASE_OBJECT                ((OM_sint) 0)
#define DS_ONE_LEVEL                  ((OM_sint) 1)
#define DS_WHOLE_SUBTREE              ((OM_sint) 2)


/* ds_receive_result() completion_flag_return */

#define DS_COMPLETED_OPERATION        ((OM_uint) 1)
#define DS_OUTSTANDING_OPERATIONS     ((OM_uint) 2)
#define DS_NO_OUTSTANDING_OPERATION   ((OM_uint) 3)


/* asynchronous operations limit (implementation-defined) */

#define DS_MAX_OUTSTANDING_OPERATIONS  0 /* no asynchronous */
                      /* operation    */
/*asynchronous event posting */

#define DS_NO_VALID_FILE_DESCRIPTOR   -1


/* Function Prototypes */

DS_status ds_abandon(
    OM_private_object  session,
    OM_sint       invoke_id
);

DS_status ds_add_entry(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_object      entry,
    OM_sint       *invoke_id_return
);

DS_status ds_bind(
    OM_object      session,
    OM_workspace    workspace,
    OM_private_object  *bound_session_return
);

DS_status ds_compare(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_object      ava,
    OM_private_object  *result_return,
    OM_sint       *invoke_id_return
);

OM_workspace ds_initialize(
    void
);

DS_status ds_list(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_private_object  *result_return,
    OM_sint       *invoke_id_return
);

DS_status ds_modify_entry(
    OM_private_object  session,
    OM_private_object  context,
```

```
    OM_object      name,
    OM_object      changes,
    OM_sint       *invoke_id_return
);


DS_status ds_modify_rdn(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_object      new_RDN,
    OM_boolean     delete_old_RDN,
    OM_sint       *invoke_id_return
);

DS_status ds_read(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_object      selection,
    OM_private_object  *result_return,
    OM_sint       *invoke_id_return
);

DS_status ds_receive_result(
    OM_private_object  session,
    OM_uint       *completion_flag_return,
    DS_status      *operation_status_return,
    OM_private_object  *result_return,
    OM_sint       *invoke_id_return
);

DS_status ds_remove_entry(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_sint       *invoke_id_return
);

DS_status ds_search(
    OM_private_object  session,
    OM_private_object  context,
    OM_object      name,
    OM_sint       subset,
    OM_object      filter,
    OM_boolean     search_aliases,
    OM_object      selection,
    OM_private_object  *result_return,
    OM_sint       *invoke_id_return
);

DS_status ds_shutdown(
    OM_workspace    workspace
);


DS_status ds_unbind(
    OM_private_object  session
);

DS_status ds_version(
    DS_feature     feature_list[]
    OM_workspace    workspace
);


#endif /* XDS_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

xdsbdcp.h

Purpose

Definitions for the basic directory contents package

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsbdcp.h>
```

Description

The **xdsbdcp.h** header defines the object identifiers of directory attribute types and object classes supported by the basic directory contents package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

Object identifiers are defined for the (directory) attribute types that are specified in the following list. The actual values of the object identifiers are listed in the *OSF DCE Application Development Guide—Directory Services*.

```
#ifndef XDSBDCP_HEADER
#define XDSBDCP_HEADER

/* BDC package object identifier */

/* { iso(1) identified-organization(3) icd-ecma(12)
   member-company(2) dec(1011) xopen(28) bdcp(1) } */

#define OMP_O_DS_BASIC_DIR_CONTENTS_PKG \
 "\x2B\x0C\x02\x87\x73\x1c\x01"


/* Intermediate object identifier macros */

#ifndef dsP_attributeType   /* joint-iso-ccitt(2)     */
                /* ds(5) attributeType(4) ... */
#define dsP_attributeType (X)  ("\x55\x04" #X)
#endif

#ifndef dsP_objectClass    /* joint-iso-ccitt(2)     */
                /* ds(5) objectClass(6) ...  */
#define dsP_objectClass(X)   ("\x55\x06" #X)
#endif

#define dsP_bdcp_c(X) (OMP_O_DS_BASIC_DIR_CONTENTS_PKG #X)


/* OM class names (prefixed by DS_C_)            */
/* Directory attribute types (prefixed by DS_A_)      */
/* Directory object classes (prefixed by DS_O_)       */

/* Every application program which makes use of a class or */
/* other Object Identifier must explicitly import it into  */
/* every compilation unit (C source program) which uses it. */
/* Each such class or Object Identifier name must be    */
```

```
/* explicitly exported from just one compilation unit.   */

/* In the header file, OM class constants are prefixed with */
/* the OMP_O prefix to denote that they are OM classes.   */
/* However, when using the OM_IMPORT and OM_EXPORT macros, */
/* the base names (without the OMP_O prefix) should be used.*/
/* For example:                          */
/*     OM_IMPORT (DS_O_COUNTRY)             */


/* Directory attribute types */

#define OMP_O_DS_A_ALIASED_OBJECT_NAME    dsP_attributeType(\x01)
#define OMP_O_DS_A_BUSINESS_CATEGORY      dsP_attributeType(\x0F)
#define OMP_O_DS_A_COMMON_NAME            dsP_attributeType(\x03)
#define OMP_O_DS_A_COUNTRY_NAME           dsP_attributeType(\x06)
#define OMP_O_DS_A_DESCRIPTION            dsP_attributeType(\x0D)
#define OMP_O_DS_A_DEST_INDICATOR         dsP_attributeType(\x1B)
#define OMP_O_DS_A_FACSIMILE_PHONE_NBR    dsP_attributeType(\x17)
#define OMP_O_DS_A_INTERNAT_ISDN_NBR      dsP_attributeType(\x19)
#define OMP_O_DS_A_KNOWLEDGE_INFO         dsP_attributeType(\x02)
#define OMP_O_DS_A_LOCALITY_NAME          dsP_attributeType(\x07)
#define OMP_O_DS_A_MEMBER                 dsP_attributeType(\x1F)
#define OMP_O_DS_A_OBJECT_CLASS           dsP_attributeType(\x00)
#define OMP_O_DS_A_ORG_NAME               dsP_attributeType(\x0A)
#define OMP_O_DS_A_ORG_UNIT_NAME          dsP_attributeType(\x0B)
#define OMP_O_DS_A_OWNER                  dsP_attributeType(\x20)
#define OMP_O_DS_A_PHYS_DELIV_OFF_NAME    dsP_attributeType(\x13)
#define OMP_O_DS_A_POST_OFFICE_BOX        dsP_attributeType(\x12)
#define OMP_O_DS_A_POSTAL_ADDRESS         dsP_attributeType(\x10)
#define OMP_O_DS_A_POSTAL_CODE            dsP_attributeType(\x11)
#define OMP_O_DS_A_PREF_DELIV_METHOD      dsP_attributeType(\x1C)
#define OMP_O_DS_A_PRESENTATION_ADDRESS   dsP_attributeType(\x1D)
#define OMP_O_DS_A_REGISTERED_ADDRESS     dsP_attributeType(\x1A)
#define OMP_O_DS_A_ROLE_OCCUPANT          dsP_attributeType(\x21)
#define OMP_O_DS_A_SEARCH_GUIDE           dsP_attributeType(\x0E)
#define OMP_O_DS_A_SEE_ALSO               dsP_attributeType(\x22)
#define OMP_O_DS_A_SERIAL_NBR             dsP_attributeType(\x05)
#define OMP_O_DS_A_STATE_OR_PROV_NAME     dsP_attributeType(\x08)
#define OMP_O_DS_A_STREET_ADDRESS         dsP_attributeType(\x09)
#define OMP_O_DS_A_SUPPORT_APPLIC_CONTEXT dsP_attributeType(\x1E)
#define OMP_O_DS_A_SURNAME                dsP_attributeType(\x04)
#define OMP_O_DS_A_PHONE_NBR              dsP_attributeType(\x14)
#define OMP_O_DS_A_TELETEX_TERM_IDENT     dsP_attributeType(\x16)
#define OMP_O_DS_A_TELEX_NBR              dsP_attributeType(\x15)
#define OMP_O_DS_A_TITLE                  dsP_attributeType(\x0C)
#define OMP_O_DS_A_USER_PASSWORD          dsP_attributeType(\x23)
#define OMP_O_DS_A_X121_ADDRESS           dsP_attributeType(\x18)


/* Directory object classes */

#define OMP_O_DS_O_ALIAS                dsP_objectClass(\x01)
#define OMP_O_DS_O_APPLIC_ENTITY        dsP_objectClass(\x0C)
#define OMP_O_DS_O_APPLIC_PROCESS       dsP_objectClass(\x0B)
#define OMP_O_DS_O_COUNTRY              dsP_objectClass(\x02)
#define OMP_O_DS_O_DEVICE               dsP_objectClass(\x0E)
#define OMP_O_DS_O_DSA                  dsP_objectClass(\x0D)
#define OMP_O_DS_O_GROUP_OF_NAMES       dsP_objectClass(\x09)
#define OMP_O_DS_O_LOCALITY             dsP_objectClass(\x03)
#define OMP_O_DS_O_ORG                  dsP_objectClass(\x04)
#define OMP_O_DS_O_ORG_PERSON           dsP_objectClass(\x07)
#define OMP_O_DS_O_ORG_ROLE             dsP_objectClass(\x08)
#define OMP_O_DS_O_ORG_UNIT             dsP_objectClass(\x05)
#define OMP_O_DS_O_PERSON               dsP_objectClass(\x06)
#define OMP_O_DS_O_RESIDENTIAL_PERSON   dsP_objectClass(\x0A)
#define OMP_O_DS_O_TOP                  dsP_objectClass(\x00)
```

```
/* OM class names */

#define OMP_O_DS_C_FACSIMILE_PHONE_NBR  dsP_bdcp_c(\x86\x21)
#define OMP_O_DS_C_POSTAL_ADDRESS       dsP_bdcp_c(\x86\x22)
#define OMP_O_DS_C_SEARCH_CRITERION     dsP_bdcp_c(\x86\x23)
#define OMP_O_DS_C_SEARCH_GUIDE         dsP_bdcp_c(\x86\x24)
#define OMP_O_DS_C_TELETEX_TERM_IDENT   dsP_bdcp_c(\x86\x25)
#define OMP_O_DS_C_TELEX_NBR            dsP_bdcp_c(\x86\x26)


/* OM attribute names */

#define DS_ANSWERBACK        ((OM_type) 801)
#define DS_COUNTRY_CODE      ((OM_type) 802)
#define DS_CRITERIA          ((OM_type) 803)
#define DS_OBJECT_CLASS      ((OM_type) 804)
#define DS_PARAMETERS        ((OM_type) 805)
#define DS_POSTAL_ADDRESS    ((OM_type) 806)
#define DS_PHONE_NBR         ((OM_type) 807)
#define DS_TELETEX_TERM      ((OM_type) 808)
#define DS_TELEX_NBR         ((OM_type) 809)


/* DS_Preferred_Delivery_Method: */

#define DS_ANY_DELIV_METHOD      0
#define DS_MHS_DELIV             1
#define DS_PHYS_DELIV            2
#define DS_TELEX_DELIV           3
#define DS_TELETEX_DELIV         4
#define DS_G3_FACSIMILE_DELIV    5
#define DS_G4_FACSIMILE_DELIV    6
#define DS_IA5_TERMINAL_DELIV    7
#define DS_VIDEOTEX_DELIV        8
#define DS_PHONE_DELIV           9


/* Upper bounds on string lengths and the number of repeated OM */
/* attribute values                       */

#define DS_VL_A_BUSINESS_CATEGORY       ((OM_value_length) 128)
#define DS_VL_A_COMMON_NAME             ((OM_value_length) 64)
#define DS_VL_A_DESCRIPTION             ((OM_value_length) 1024)
#define DS_VL_A_DEST_INDICATOR          ((OM_value_length) 128)
#define DS_VL_A_INTERNAT_ISDN_NBR       ((OM_value_length) 16)
#define DS_VL_A_LOCALITY_NAME           ((OM_value_length) 128)
#define DS_VL_A_ORG_NAME                ((OM_value_length) 64)
#define DS_VL_A_ORG_UNIT_NAME           ((OM_value_length) 64)
#define DS_VL_A_PHYS_DELIV_ OFF_NAME    ((OM_value_length) 128)
#define DS_VL_A_POST_OFFICE_BOX         ((OM_value_length) 40)
#define DS_VL_A_POSTAL_CODE             ((OM_value_length) 40)
#define DS_VL_A_SERIAL_NBR              ((OM_value_length) 64)
#define DS_VL_A_STATE_OR_PROV_NAME      ((OM_value_length) 128)
#define DS_VL_A_STREET_ADDRESS          ((OM_value_length) 128)
#define DS_VL_A_SURNAME                 ((OM_value_length) 64)
#define DS_VL_A_PHONE_NBR               ((OM_value_length) 32)
#define DS_VL_A_TITLE                   ((OM_value_length) 64)
#define DS_VL_A_USER_PASSWORD           ((OM_value_length) 128)
#define DS_VL_A_X121_ADDRESS            ((OM_value_length) 15)
#define DS_VL_ANSWERBACK                ((OM_value_length) 8)
#define DS_VL_COUNTRY_CODE              ((OM_value_length) 4)
#define DS_VL_POSTAL_ADDRESS            ((OM_value_length) 30)
#define DS_VL_PHONE_NBR                 ((OM_value_length) 32)
#define DS_VL_TELETEX_TERM              ((OM_value_length) 1024)
```

```
#define DS_VL_TELEX_NBR                 ((OM_value_length) 14)
#define DS_VN_POSTAL_ADDRESS            ((OM_value_length) 6)

#endif /* XDSBDCP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

xdscds.h

Purpose

Definitions for the Cell Directory Service (CDS)

Synopsis

#include <xom.h>#include
<xds.h>#include
<xdscds.h>

Description

The **xdscds.h** header declares the object identifiers of directory attribute types supported by CDS.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSCDS_HEADER
#define XDSCDS_HEADER

/* iso(1) identified-organization(3) osf(22) dce(1) cds(3)
  = "\x2B\x16\x01\x03" */

/* Cell Directory Service attribute types */

#define OMP_O_DSX_A_CDS_Members         "\x2B\x16\x01\x03\x0A"
#define OMP_O_DSX_A_CDS_GroupRevoke     "\x2B\x16\x01\x03\x0B"
#define OMP_O_DSX_A_CDS_CTS             "\x2B\x16\x01\x03\x0C"
#define OMP_O_DSX_A_CDS_UTS             "\x2B\x16\x01\x03\x0D"
#define OMP_O_DSX_A_CDS_ACS             "\x2B\x16\x01\x03\x0E"
#define OMP_O_DSX_A_CDS_Class           "\x2B\x16\x01\x03\x0F"
#define OMP_O_DSX_A_CDS_ClassVersion    "\x2B\x16\x01\x03\x10"
#define OMP_O_DSX_A_CDS_ObjectUID       "\x2B\x16\x01\x03\x11"
#define OMP_O_DSX_A_CDS_Address         "\x2B\x16\x01\x03\x12"
#define OMP_O_DSX_A_CDS_Replicas        "\x2B\x16\x01\x03\x13"
#define OMP_O_DSX_A_CDS_AllUpTo         "\x2B\x16\x01\x03\x14"
#define OMP_O_DSX_A_CDS_Convergence     "\x2B\x16\x01\x03\x15"
#define OMP_O_DSX_A_CDS_InCHName        "\x2B\x16\x01\x03\x16"
#define OMP_O_DSX_A_CDS_ParentPointer   "\x2B\x16\x01\x03\x17"
#define OMP_O_DSX_A_CDS_DirecoryVersion "\x2B\x16\x01\x03\x18"
#define OMP_O_DSX_A_CDS_UpgradeTo       "\x2B\x16\x01\x03\x19"
#define OMP_O_DSX_A_CDS_LinkTarget      "\x2B\x16\x01\x03\x1B"
#define OMP_O_DSX_A_CDS_LinkTimeout     "\x2B\x16\x01\x03\x1C"
#define OMP_O_DSX_A_CDS_Towers          "\x2B\x16\x01\x03\x1E"
#define OMP_O_DSX_A_CDS_CHName          "\x2B\x16\x01\x03\x20"
#define OMP_O_DSX_A_CDS_CHLastAddress   "\x2B\x16\x01\x03\x22"
#define OMP_O_DSX_A_CDS_CHUpPointers    "\x2B\x16\x01\x03\x23"
#define OMP_O_DSX_A_CDS_CHState         "\x2B\x16\x01\x03\x24"

/* iso(1) identified-organization(3) osf(22) dce(1) gds(2)
  = "\x2B\x16\x01\x02" */

#define OMP_O_DSX_UUID                  "\x2B\x16\x01\x01\x01"
#define OMP_O_DSX_TYPELESS_RDN          "\x2B\x16\x01\x01\x02"
#define OMP_O_DSX_NORMAL_SIMPLE_NAME    "\x2B\x16\x01\x03\x00"
#define OMP_O_DSX_BINARY_SIMPLE_NAME    "\x2B\x16\x01\x03\x02"

#endif /*XDSCDS_HEADER*/
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

xdsdme.h

Purpose

Definitions for the DME NMO requirements.

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsdme.h>
```

Description

The **xdsdme.h** header declares the object identifiers of directory attribute types and directory object classes supported for DME use.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSDME_HEADER
#define XDSDME_HEADER


/* Intermediate object identifier macros */

/* iso(1) identified-organization(3) osf(22) dme(2)
  components(1) nmo(2) dmeNmoAttributeType(1) ...
*/

#define dsP_NMOattributeType(X)  "\x2B\x16\x02\x01\x02\x01" #X

/* iso(1) identified-organization(3) osf(22) dme(2)
  components(1) nmo(2) dmeNmoObjectClass(2) ...
*/

#define dsP_NMOobjectClass(X)   "\x2B\x16\x02\x01\x02\x02" #X

/* Directory attribute types (prefixed by DSX_A_)
  Directory object classes (prefixed by DSX_O_)
*/


/* Directory attribute types */

#define OMP_O_DSX_A_ALTERNATE_ADDRESS dsP_NMOattributeType(\x01)


/* Directory object classes */

#define OMP_O_DSX_O_DME_NMO_AGENT dsP_NMOobjectClass(\x01)


#endif /* XDSDME_HEADER */
```

Related Information

Books: *OSF DCE Application Development Guide—Directory Services*.

xdsgds.h

Purpose

Definitions for the global directory service package

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsgds.h>
```

Description

The **xdsgds.h** header declares the object identifiers of directory attribute types and directory object classes supported by the GDS package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSGDS_HEADER
#define XDSGDS_HEADER


/* GDS package object identifier */
/* iso(1) identified-organization(3) icd-ecma(0012)
  member-company(2) siemens-units(1107) sni(1) directory(3)
  xds-api(100)gdsp(0) */

#define OMP_O_DSX_GDS_PKG  \
"\x2B\x0C\x02\x88\x53\x01\x03\x64\x00"

/*Intermediate object identifier macros */

/* iso(1) identified-organization(3) icd-ecma(0012)
  member-company(2) siemens-units(1107) sni(1) directory(3)
  attribute-type(4) ...*/

#define dsP_GDSattributeType(X) \
("\x2B\x0C\x02\x88\x53\x01\x03\x04" #X)

/* iso(1) identified-organization(3) icd-ecma(0012)
  member-company(2) siemens-units(1107) sni(1) directory(3)
  object-class(6) ...*/

#define dsP_GDSobjectClass(X) \
("\x2B\x0C\x02\x88\x53\x01\x03\x06" #X)

#define dsP_gdsp_c(X)  OMP_O_DSX_GDS_PKG #X


/* OM class names (prefixed by DSX_C_)
  Directory attribute types (prefixed by DSX_A_)
  Directory object classes (prefixed by DSX_O_)
*/


/* Directory attribute types */

#define OMP_O_DSX_A_MASTER_KNOWLEDGE dsP_GDSattributeType(\x00)
```

```
#define OMP_O_DSX_A_ACL              dsP_GDSattributeType(\x01)
#define OMP_O_DSX_A_TIME_STAMP       dsP_GDSattributeType(\x02)
#define OMP_O_DSX_A_SHADOWED_BY      dsP_GDSattributeType(\x03)
#define OMP_O_DSX_A_SRT              dsP_GDSattributeType(\x04)
#define OMP_O_DSX_A_OCT              dsP_GDSattributeType(\x05)
#define OMP_O_DSX_A_AT               dsP_GDSattributeType(\x06)
#define OMP_O_DSX_A_DEFAULT_DSA      dsP_GDSattributeType(\x08)
#define OMP_O_DSX_A_LOCAL_DSA        dsP_GDSattributeType(\x09)
#define OMP_O_DSX_A_CLIENT           dsP_GDSattributeType(\x0A)
#define OMP_O_DSX_A_DNLIST           dsP_GDSattributeType(\x0B)
#define OMP_O_DSX_A_SHADOWING_JOB    dsP_GDSattributeType(\x0C)
#define OMP_O_DSX_A_CDS_CELL         dsP_GDSattributeType(\x0D)
#define OMP_O_DSX_A_CDS_REPLICA      dsP_GDSattributeType(\x0E)


/* Directory object classes */

#define OMP_O_DSX_O_SCHEMA           dsP_GDSobjectClass(\x00)

/* OM class names */

#define OMP_O_DSX_C_GDS_SESSION     dsP_gdsp_c(\x00)
#define OMP_O_DSX_C_GDS_CONTEXT     dsP_gdsp_c(\x01)
#define OMP_O_DSX_C_GDS_ACL         dsP_gdsp_c(\x02)
#define OMP_O_DSX_C_GDS_ACL_ITEM    dsP_gdsp_c(\x03)


/* OM attribute names */

#define DSX_PASSWORD            ((OM_type) 850)
#define DSX_DIR_ID              ((OM_type) 851)
#define DSX_DUAFIRST            ((OM_type) 852)
#define DSX_DONT_STORE          ((OM_type) 853)
#define DSX_NORMAL_CLASS        ((OM_type) 854)
#define DSX_PRIV_CLASS          ((OM_type) 855)
#define DSX_RESIDENT_CLASS      ((OM_type) 856)
#define DSX_USEDSA              ((OM_type) 857)
#define DSX_DUA_CACHE           ((OM_type) 858)
#define DSX_MODIFY_PUBLIC       ((OM_type) 859)
#define DSX_READ_STANDARD       ((OM_type) 860)
#define DSX_MODIFY_STANDARD     ((OM_type) 861)
#define DSX_READ_SENSITIVE      ((OM_type) 862)
#define DSX_MODIFY_SENSITIVE    ((OM_type) 863)
#define DSX_INTERPRETATION      ((OM_type) 864)
#define DSX_USER                ((OM_type) 865)
#define DSX_PREFER_ADM_FUNCS    ((OM_type) 866)
#define DSX_AUTH_MECHANISM      ((OM_type) 867)
#define DSX_AUTH_INFO           ((OM_type) 868) /* future use */
#define DSX_SIGN_MECHANISM      ((OM_type) 869) /* future use */
#define DSX_PROT_REQUEST        ((OM_type) 870) /* future use */


/* DSX_Interpretation */

enum DSX_Interpretation {
    DSX_SINGLE_OBJECT   = 0,
    DSX_ROOT_OF_SUBTREE = 1
};


enum DSX_Auth_Mechanism {
    DSX_DEFAULT        = 1,
    DSX_SIMPLE         = 2,
    DSX_SIMPLE_PROT1   = 3,
    DSX_SIMPLE_PROT2   = 4,
    DSX_DCE_AUTH       = 5,
    DSX_STRONG         = 6
```

```
};


enum DSX_Prot_Request {
    DSX_NONE        = 0,
    DSX_SIGNED      = 1
};


/* upper bound on string lengths*/

#define DSX_VL_PASSWORD     ((OM_value_length) 16)

#endif /* XDSGDS_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

xdsmdup.h

Purpose

Definitions for the MHS directory user package

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsmdup.h>
```

Description

The **xdsmdup.h** header declares the object identifiers of directory attribute types and object classes supported by the MHS directory user package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the object management header **xom.h** and the **xds.h** header.

```
#ifndef XDSMDUP_HEADER
#define XDSMDUP_HEADER

#ifndef XMHP_HEADER
#include <xmhp.h>
#endif /* XMHP_HEADER */

/* MDUP package object identifier */

/* { iso(1) identified-organization(3) icd-ecma(12)
   member-company(2) dec(1011) xopen(28) mdup(3) } */

#define OMP_O_DS_MHS_DIR_USER_PKG \
"\x2B\x0C\x02\x87\x73\x1C\x03"

/* Intermediate object identifier macros */

/* { joint-iso-ccitt(2) mhs-motis(6) arch(5) at(2) } */

#define dsP_MHSattributeType(X) ("\x56\x5\x2" #X)

/* { joint-iso-ccitt(2) mhs-motis(6) arch(5) oc(1) } */

#define dsP_MHSobjectClass(X)  ("\x56\x5\x1" #X)

#define dsP_mdup_c(X)   (OMP_O_DS_MHS_DIR_USER_PKG #X)


/* OM class names (prefixed DS_C_),               */
/* Directory attribute types (prefixed DS_A_),         */
/* and Directory object classes (prefixed DS_O_)        */

/* Every application program which makes use of a class or   */
/* other Object Identifier must explicitly import it into    */
/* every compilation unit (C source program) which uses it.   */
/* Each such class or Object Identifier name must be      */
/* explicitly exported from just one compilation unit.     */

/* In the header file, OM class constants are prefixed with   */
/* the OMP_O prefix to denote that they are OM classes.     */
```

```
/* However, when using the OM_IMPORT and OM_EXPORT macros,   */
/* the base names (without the OMP_O prefix) should be used.  */
/* For example:                                 */
/*    OM_IMPORT(DS_O_CERT_AUTHORITY)            */


/* Directory attribute types */

#define OMP_O_DS_A_DELIV_CONTENT_LENGTH   dsP_MHSattributeType(\x00)
#define OMP_O_DS_A_DELIV_CONTENT_TYPES    dsP_MHSattributeType(\x01)
#define OMP_O_DS_A_DELIV_EITS             dsP_MHSattributeType(\x02)
#define OMP_O_DS_A_DL_MEMBERS             dsP_MHSattributeType(\x03)
#define OMP_O_DS_A_DL_SUBMIT_PERMS        dsP_MHSattributeType(\x04)
#define OMP_O_DS_A_MESSAGE_STORE          dsP_MHSattributeType(\x05)
#define OMP_O_DS_A_OR_ADDRESSES           dsP_MHSattributeType(\x06)
#define OMP_O_DS_A_PREF_DELIV_METHODS     dsP_MHSattributeType(\x07)
#define OMP_O_DS_A_SUPP_AUTO_ACTIONS      dsP_MHSattributeType(\x08)
#define OMP_O_DS_A_SUPP_CONTENT_TYPES     dsP_MHSattributeType(\x09)
#define OMP_O_DS_A_SUPP_OPT_ATTRIBUTES    dsP_MHSattributeType(\x0A)


/* Directory object classes */

#define OMP_O_DS_O_MHS_DISTRIBUTION_LIST  dsP_MHSobjectClass(\x00)
#define OMP_O_DS_O_MHS_MESSAGE_STORE      dsP_MHSobjectClass(\x01)
#define OMP_O_DS_O_MHS_MESSAGE_TRANS_AG   dsP_MHSobjectClass(\x02)
#define OMP_O_DS_O_MHS_USER               dsP_MHSobjectClass(\x03)
#define OMP_O_DS_O_MHS_USER_AG            dsP_MHSobjectClass(\x04)


/* OM class names */

#define OMP_O_DS_C_DL_SUBMIT_PERMS        dsP_mdup_c(\x87\x05)


/* OM attribute names */

#define DS_PERM_TYPE         ( (OM_type) 901 )
#define DS_INDIVIDUAL        ( (OM_type) 902 )
#define DS_MEMBER_OF_DL      ( (OM_type) 903 )
#define DS_PATTERN_MATCH     ( (OM_type) 904 )
#define DS_MEMBER_OF_GROUP   ( (OM_type) 905 )


/* DS_Permission_Type */

enum DS_Permission_Type {
    DS_PERM_INDIVIDUAL      = 0,
    DS_PERM_MEMBER_OF_DL    = 1,
    DS_PERM_PATTERN_MATCH   = 2,
    DS_PERM_MEMBER_OF_GROUP = 3
};


#endif /* XDSMDUP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*, *X/Open CAE Specification (November 1991), API to Electronic Mail (X.400)*.

xdssap.h

Purpose

Definitions for the strong authentication package

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdssap.h>
```

Description

The **xdssap.h** header defines the object identifiers of directory attribute types and object classes supported by the strong authentication package. It also defines OM classes used to represent the values of the attribute types.

All application programs that include this header must first include the **xom.h** object management header and the **xds.h** header.

```
#ifndef XDSSAP_HEADER
#define XDSSAP_HEADER

/* Strong Authentication Package object identifier */
/* { iso(1) identified-organization(3) icd-ecma(12)
   member-company(2) dec(1011) xopen(28) sap(2) } */

#define OMP_O_DS_STRONG_AUTHENT_PKG \
 "\x2B\x0C\x02\x87\x73\x1c\x02"


/* Intermediate object identifier macros */

#ifndef dsP_attributeType   /* joint-iso-ccitt(2)     */
                /* ds(5) attributeType(4) ... */
#define dsP_attributeType (X)  ("\x55\x04" #X)
#endif

#ifndef dsP_objectClass    /* joint-iso-ccitt(2)     */
                /* ds(5) objectClass(6) ...  */
#define dsP_objectClass(X)   ("\x55\x06" #X)
#endif

#define dsP_sap_c(X) (OMP_O_DS_STRONG_AUTHENT_PKG #X)


/* OM class names (prefixed by DS_C_)            */
/* Directory attribute types (prefixed by DS_A_)      */
/* Directory object classes (prefixed by DS_O_)       */

/* Every application program which makes use of a class or */
/* other Object Identifier must explicitly import it into  */
/* every compilation unit (C source program) which uses it. */
/* Each such class or Object Identifier name must be    */
/* explicitly exported from just one compilation unit.   */

/* In the header file, OM class constants are prefixed with */
/* the OMP_O prefix to denote that they are OM classes.   */
/* However, when using the OM_IMPORT and OM_EXPORT macros, */
/* the base names (without the OMP_O prefix) should be used.*/
```

```
/* For example:                          */
/*     OM_IMPORT (DS_O_CERT_AUTHORITY)         */


/* Directory attribute types */

#define OMP_O_DS_A_AUTHORITY_REVOC_LIST  dsP_attributeType(\x26)
#define OMP_O_DS_A_CA_CERT               dsP_attributeType(\x25)
#define OMP_O_DS_A_CERT_REVOC_LIST       dsP_attributeType(\x27)
#define OMP_O_DS_A_CROSS_CERT_PAIR       dsP_attributeType(\x28)
#define OMP_O_DS_A_USER_CERT             dsP_attributeType(\x24)


/* Directory object classes */

#define OMP_O_DS_O_CERT_AUTHORITY        dsP_objectClass(\x10)
#define OMP_O_DS_O_STRONG_AUTHENT_USER   dsP_objectClass(\x0F)


/* OM class names */

#define OMP_O_DS_C_ALGORITHM_IDENT       dsP_sap_c(\x6\x35)
#define OMP_O_DS_C_CERT                  dsP_sap_c(\x6\x36)
#define OMP_O_DS_C_CERT_LIST             dsP_sap_c(\x6\x37)
#define OMP_O_DS_C_CERT_PAIR             dsP_sap_c(\x6\x38)
#define OMP_O_DS_C_CERT_SUBLIST          dsP_sap_c(\x6\x39)
#define OMP_O_DS_C_SIGNATURE             dsP_sap_c(\x6\x3A)


/* OM attribute names */

#define DS_ALGORITHM            ((OM_type) 821)
#define DS_FORWARD              ((OM_type) 822)
#define DS_ISSUER               ((OM_type) 823)
#define DS_LAST_UPDATE          ((OM_type) 824)
#define DS_ALGORITHM_PARAMETERS ((OM_type) 825)
#define DS_REVERSE              ((OM_type) 826)
#define DS_REVOCATION_DATE      ((OM_type) 827)
#define DS_REVOKED_CERTS        ((OM_type) 828)
#define DS_SERIAL_NUMBER        ((OM_type) 829)
#define DS_SERIAL_NUMBERS       ((OM_type) 830)
#define DS_SIGNATURE            ((OM_type) 831)
#define DS_SIGNATURE_VALUE      ((OM_type) 832)
#define DS_SUBJECT              ((OM_type) 833)
#define DS_SUBJECT_ALGORITHM    ((OM_type) 834)
#define DS_SUBJECT_PUBLIC_KEY   ((OM_type) 835)
#define DS_VALIDITY_NOT_AFTER   ((OM_type) 836)
#define DS_VALIDITY_NOT_BEFORE  ((OM_type) 837)
#define DS_VERSION              ((OM_type) 838)


/* DS_Version */

#define DS_V1988        ((OM_enumeration) 1)

/* Upper bounds on string lengths and the number of repeated OM */
/* attribute values                        */

#define DS_VL_LAST_UPDATE            ((OM_value_length) 17)
#define DS_VL_REVOC_DATE             ((OM_value_length) 17)
#define DS_VL_VALIDITY_NOT_AFTER     ((OM_value_length) 17)
#define DS_VL_VALIDITY_NOT_BEFORE    ((OM_value_length) 17)
#define DS_VN_REVOC_DATE             ((OM_value_length) 2)

#endif /* XDSSAP_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

xmhp.h

Purpose

Definitions for the MHS directory objects/attributes.

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsmdup.h>
#include  <xmhp.h>
```

Description

The **xmhp.h** header defines the constants used by the message handling packages. It is required when using the MHS directory user package. The **xdsmdup.h** header explicitly includes **xmhp.h**.

xmhp.h contains definitions for the X.400 message handling package. Some of these definitions are needed when negotiating use of the MDUP.

The following four message handling classes are referenced:

- **MH_C_G3_FAX_NBPS**
- **MH_C_OR_ADDRESS**
- **MH_C_OR_NAME**
- **MH_C_TELETEX_NBPS**

The only enumerations referenced are **Delivery Mode** and **Terminal Type**. For referenced OM attribute types and OM value lengths see the *OSF DCE Application Development Guide—Directory Services*.

```
/*
 Note that the identifier for the variable name of type OM_STRING
of a class in the Message Handling package can usually be
derived using the name of the class, preceded by "MH_C_", and
replacing a blank space with an underscore. To be in line with the
ANSI C language limitation, some words in the class names are
excepted and are abbreviated as below:

    BILATERAL_INFORMATION is abbreviated to BILATERAL_INFO
    DELIVERED                DELIV
    CONFIRMATION              CONFIRM
    CONFIRMATIONS              CONFIRMS
    PER_RECIPIENT_             PER_RECIP_
    DELIV_PER_RECIP_REPORT         DELIV_PER_RECIP_REP

*/

/* BEGIN MH PORTION OF INTERFACE */

/* SYMBOLIC CONSTANTS */

/* Class */

#define OMP_O_MH_C_ALGORITHM            "\x56\x06\x01\x02\x05\x0B\x00"
#define OMP_O_MH_C_ALGORITHM_AND_RESULT "\x56\x06\x01\x02\x05\x0B\x01"
```

```
#define OMP_O_MH_C_ASYMMETRIC_TOKEN     "\x56\x06\x01\x02\x05\x0B\x02"
#define OMP_O_MH_C_BILATERAL_INFO       "\x56\x06\x01\x02\x05\x0B\x03"
#define OMP_O_MH_C_COMMUNIQUE           "\x56\x06\x01\x02\x05\x0B\x04"
#define OMP_O_MH_C_CONTENT              "\x56\x06\x01\x02\x05\x0B\x05"
#define OMP_O_MH_C_DELIV_MESSAGE        "\x56\x06\x01\x02\x05\x0B\x06"
#define OMP_O_MH_C_DELIV_PER_RECIP_DR   "\x56\x06\x01\x02\x05\x0B\x07"
#define OMP_O_MH_C_DELIV_PER_RECIP_NDR  "\x56\x06\x01\x02\x05\x0B\x08"
#define OMP_O_MH_C_DELIV_PER_RECIP_REP  "\x56\x06\x01\x02\x05\x0B\x09"
#define OMP_O_MH_C_DELIV_REPORT         "\x56\x06\x01\x02\x05\x0B\x0A"
#define OMP_O_MH_C_DELIVERY_CONFIRM     "\x56\x06\x01\x02\x05\x0B\x0B"
#define OMP_O_MH_C_DELIVERY_ENVELOPE    "\x56\x06\x01\x02\x05\x0B\x0C"
#define OMP_O_MH_C_EITS                 "\x56\x06\x01\x02\x05\x0B\x0D"
#define OMP_O_MH_C_EXPANSION_RECORD     "\x56\x06\x01\x02\x05\x0B\x0E"
#define OMP_O_MH_C_EXTENSIBLE_OBJECT    "\x56\x06\x01\x02\x05\x0B\x0F"
#define OMP_O_MH_C_EXTENSION            "\x56\x06\x01\x02\x05\x0B\x10"
#define OMP_O_MH_C_EXTERNAL_TRACE_ENTRY "\x56\x06\x01\x02\x05\x0B\x11"
#define OMP_O_MH_C_G3_FAX_NBPS          "\x56\x06\x01\x02\x05\x0B\x12"
#define OMP_O_MH_C_GENERAL_CONTENT      "\x56\x06\x01\x02\x05\x0B\x13"
#define OMP_O_MH_C_INTERNAL_TRACE_ENTRY "\x56\x06\x01\x02\x05\x0B\x14"
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRM  "\x56\x06\x01\x02\x05\x0B\x15"
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRMS "\x56\x06\x01\x02\x05\x0B\x16"
#define OMP_O_MH_C_LOCAL_NDR            "\x56\x06\x01\x02\x05\x0B\x17"
#define OMP_O_MH_C_LOCAL_PER_RECIP_NDR  "\x56\x06\x01\x02\x05\x0B\x18"
#define OMP_O_MH_C_MESSAGE              "\x56\x06\x01\x02\x05\x0B\x19"
#define OMP_O_MH_C_MESSAGE_RD           "\x56\x06\x01\x02\x05\x0B\x1A"
#define OMP_O_MH_C_MTS_IDENTIFIER       "\x56\x06\x01\x02\x05\x0B\x1B"
#define OMP_O_MH_C_OR_ADDRESS           "\x56\x06\x01\x02\x05\x0B\x1C"
#define OMP_O_MH_C_OR_NAME              "\x56\x06\x01\x02\x05\x0B\x1D"
#define OMP_O_MH_C_PER_RECIP_DR         "\x56\x06\x01\x02\x05\x0B\x1E"
#define OMP_O_MH_C_PER_RECIP_NDR        "\x56\x06\x01\x02\x05\x0B\x1F"
#define OMP_O_MH_C_PER_RECIP_REPORT     "\x56\x06\x01\x02\x05\x0B\x20"
#define OMP_O_MH_C_PROBE                "\x56\x06\x01\x02\x05\x0B\x21"
#define OMP_O_MH_C_PROBE_RD             "\x56\x06\x01\x02\x05\x0B\x22"
#define OMP_O_MH_C_RD                   "\x56\x06\x01\x02\x05\x0B\x23"
#define OMP_O_MH_C_REDIRECTION_RECORD   "\x56\x06\x01\x02\x05\x0B\x24"
#define OMP_O_MH_C_REPORT               "\x56\x06\x01\x02\x05\x0B\x25"
#define OMP_O_MH_C_SECURITY_LABEL       "\x56\x06\x01\x02\x05\x0B\x26"
#define OMP_O_MH_C_SESSION              "\x56\x06\x01\x02\x05\x0B\x27"
#define OMP_O_MH_C_SUBMISSION_RESULTS   "\x56\x06\x01\x02\x05\x0B\x28"
#define OMP_O_MH_C_SUBMITTED_COMMUNIQUE "\x56\x06\x01\x02\x05\x0B\x29"
#define OMP_O_MH_C_SUBMITTED_MESSAGE    "\x56\x06\x01\x02\x05\x0B\x2A"
#define OMP_O_MH_C_SUBMITTED_MESSAGE_RD "\x56\x06\x01\x02\x05\x0B\x2B"
#define OMP_O_MH_C_SUBMITTED_PROBE      "\x56\x06\x01\x02\x05\x0B\x2C"
#define OMP_O_MH_C_SUBMITTED_PROBE_RD  "\x56\x06\x01\x02\x05\x0B\x2D"
#define OMP_O_MH_C_TELETEX_NBPS        "\x56\x06\x01\x02\x05\x0B\x2E"
#define OMP_O_MH_C_DELIVERY_REPORT     "\x56\x06\x01\x02\x05\x0B\x2F"
#define OMP_O_MH_C_MT_PUBLIC_DATA      "\x56\x06\x01\x02\x05\x0B\x30"
#define OMP_O_MH_C_TOKEN_PUBLIC_DATA   "\x56\x06\x01\x02\x05\x0B\x31"


/* Enumeration */

/* Action */
#define MH_AC_EXPANDED      ( (OM_enumeration) -2 )
#define MH_AC_REDIRECTED    ( (OM_enumeration) -1 )
#define MH_AC_RELAYED       ( (OM_enumeration) 0 )
#define MH_AC_REROUTED      ( (OM_enumeration) 1 )

/* Builtin EIT */
#define MH_BE_UNDEFINED     ( (OM_enumeration) 0 )
#define MH_BE_TELEX         ( (OM_enumeration) 1 )
#define MH_BE_IA5_TEXT      ( (OM_enumeration) 2 )
#define MH_BE_G3_FAX        ( (OM_enumeration) 3 )
#define MH_BE_G4_CLASS1     ( (OM_enumeration) 4 )
#define MH_BE_TELETEX       ( (OM_enumeration) 5 )
#define MH_BE_VIDEOTEX      ( (OM_enumeration) 6 )
#define MH_BE_MIXED_MODE    ( (OM_enumeration) 9 )
```

```
#define MH_BE_ODA           ( (OM_enumeration) 10 )
#define MH_BE_ISO_6937_TEXT ( (OM_enumeration) 11 )

/* Delivery Mode */
#define MH_DM_ANY           ( (OM_enumeration) 0 )
#define MH_DM_MTS           ( (OM_enumeration) 1 )
#define MH_DM_PDS           ( (OM_enumeration) 2 )
#define MH_DM_TELEX         ( (OM_enumeration) 3 )
#define MH_DM_TELETEX       ( (OM_enumeration) 4 )
#define MH_DM_G3_FAX        ( (OM_enumeration) 5 )
#define MH_DM_G4_FAX        ( (OM_enumeration) 6 )
#define MH_DM_IA5_TERMINAL  ( (OM_enumeration) 7 )
#define MH_DM_VIDEOTEX      ( (OM_enumeration) 8 )
#define MH_DM_TELEPHONE     ( (OM_enumeration) 9 )

/* Delivery Point */
#define MH_DP_PUBLIC_UA     ( (OM_enumeration) 0 )
#define MH_DP_PRIVATE_UA    ( (OM_enumeration) 1 )
#define MH_DP_MS            ( (OM_enumeration) 2 )
#define MH_DP_DL            ( (OM_enumeration) 3 )
#define MH_DP_PDAU          ( (OM_enumeration) 4 )
#define MH_DP_PDS_PATRON    ( (OM_enumeration) 5 )
#define MH_DP_OTHER_AU      ( (OM_enumeration) 6 )


/* Diagnostic */

#define MH_DG_NO_DIAGNOSTIC               ( (OM_enumeration) -1 )
#define MH_DG_OR_NAME_UNRECOGNIZED        ( (OM_enumeration) 0 )
#define MH_DG_OR_NAME_AMBIGUOUS           ( (OM_enumeration) 1 )
#define MH_DG_MTS_CONGESTED               ( (OM_enumeration) 2 )
#define MH_DG_LOOP_DETECTED               ( (OM_enumeration) 3 )
#define MH_DG_RECIPIENT_UNAVAILABLE       ( (OM_enumeration) 4 )
#define MH_DG_MAXIMUM_TIME_EXPIRED        ( (OM_enumeration) 5 )
#define MH_DG_EITS_UNSUPPORTED            ( (OM_enumeration) 6 )
#define MH_DG_CONTENT_TOO_LONG            ( (OM_enumeration) 7 )
#define MH_DG_IMPRACTICAL_TO_CONVERT      ( (OM_enumeration) 8 )
#define MH_DG_PROHIBITED_TO_CONVERT       ( (OM_enumeration) 9 )
#define MH_DG_CONVERSION_UNSUBSCRIBED     ( (OM_enumeration) 10 )
#define MH_DG_PARAMETERS_INVALID          ( (OM_enumeration) 11 )
#define MH_DG_CONTENT_SYNTAX_IN_ERROR     ( (OM_enumeration) 12 )
#define MH_DG_LENGTH_CONSTRAINT_VIOLATD   ( (OM_enumeration) 13 )
#define MH_DG_NUMBER_CONSTRAINT_VIOLATD   ( (OM_enumeration) 14 )
#define MH_DG_CONTENT_TYPE_UNSUPPORTED    ( (OM_enumeration) 15 )
#define MH_DG_TOO_MANY_RECIPIENTS         ( (OM_enumeration) 16 )
#define MH_DG_NO_BILATERAL_AGREEMENT      ( (OM_enumeration) 17 )
#define MH_DG_CRITICAL_FUNC_UNSUPPORTED   ( (OM_enumeration) 18 )
#define MH_DG_CONVERSION_LOSS_PROHIB      ( (OM_enumeration) 19 )
#define MH_DG_LINE_TOO_LONG               ( (OM_enumeration) 20 )
#define MH_DG_PAGE_TOO_LONG               ( (OM_enumeration) 21 )
#define MH_DG_PICTORIAL_SYMBOL_LOST       ( (OM_enumeration) 22 )
#define MH_DG_PUNCTUATION_SYMBOL_LOST     ( (OM_enumeration) 23 )
#define MH_DG_ALPHABETIC_CHARACTER_LOST   ( (OM_enumeration) 24 )
#define MH_DG_MULTIPLE_INFO_LOSSES        ( (OM_enumeration) 25 )
#define MH_DG_REASSIGNMENT_PROHIBITED     ( (OM_enumeration) 26 )
#define MH_DG_REDIRECTION_LOOP_DETECTED   ( (OM_enumeration) 27 )
#define MH_DG_EXPANSION_PROHIBITED        ( (OM_enumeration) 28 )
#define MH_DG_SUBMISSION_PROHIBITED       ( (OM_enumeration) 29 )
#define MH_DG_EXPANSION_FAILED            ( (OM_enumeration) 30 )
#define MH_DG_RENDITION_UNSUPPORTED       ( (OM_enumeration) 31 )
#define MH_DG_MAIL_ADDRESS_INCORRECT      ( (OM_enumeration) 32 )
#define MH_DG_MAIL_OFFICE_INCOR_OR_INVD   ( (OM_enumeration) 33 )
#define MH_DG_MAIL_ADDRESS_INCOMPLETE     ( (OM_enumeration) 34 )
#define MH_DG_MAIL_RECIPIENT_UNKNOWN      ( (OM_enumeration) 35 )
#define MH_DG_MAIL_RECIPIENT_DECEASED     ( (OM_enumeration) 36 )
#define MH_DG_MAIL_ORGANIZATION_EXPIRED   ( (OM_enumeration) 37 )
#define MH_DG_MAIL_REFUSED                ( (OM_enumeration) 38 )
```

```
#define MH_DG_MAIL_UNCLAIMED              ( (OM_enumeration) 39 )
#define MH_DG_MAIL_RECIPIENT_MOVED        ( (OM_enumeration) 40 )
#define MH_DG_MAIL_RECIPIENT_TRAVELLING   ( (OM_enumeration) 41 )
#define MH_DG_MAIL_RECIPIENT_DEPARTED     ( (OM_enumeration) 42 )
#define MH_DG_MAIL_NEW_ADDRESS_UNKNOWN    ( (OM_enumeration) 43 )
#define MH_DG_MAIL_FORWARDING_UNWANTED    ( (OM_enumeration) 44 )
#define MH_DG_MAIL_FORWARDING_PROHIB      ( (OM_enumeration) 45 )
#define MH_DG_SECURE_MESSAGING_ERROR      ( (OM_enumeration) 46 )
#define MH_DG_DOWNGRADING_IMPOSSIBLE      ( (OM_enumeration) 47 )

/* Explicit Conversion */

#define MH_EC_NO_CONVERSION           ( (OM_enumeration) -1 )
#define MH_EC_IA5_TEXT_TO_TELETEX     ( (OM_enumeration) 0 )
#define MH_EC_TELETEX_TO_TELEX        ( (OM_enumeration) 1 )
#define MH_EC_TELEX_TO_IA5_TEXT       ( (OM_enumeration) 2 )
#define MH_EC_TELEX_TO_TELETEX        ( (OM_enumeration) 3 )
#define MH_EC_TELEX_TO_G4_CLASS1      ( (OM_enumeration) 4 )
#define MH_EC_TELEX_TO_VIDEOTEX       ( (OM_enumeration) 5 )
#define MH_EC_IA5_TEXT_TO_TELEX       ( (OM_enumeration) 6 )
#define MH_EC_TELEX_TO_G3_FAX         ( (OM_enumeration) 7 )
#define MH_EC_IA5_TEXT_TO_G3_FAX      ( (OM_enumeration) 8 )
#define MH_EC_IA5_TEXT_TO_G4_CLASS1   ( (OM_enumeration) 9 )
#define MH_EC_IA5_TEXT_TO_VIDEOTEX    ( (OM_enumeration) 10 )
#define MH_EC_TELETEX_TO_IA5_TEXT     ( (OM_enumeration) 11 )
#define MH_EC_TELETEX_TO_G3_FAX       ( (OM_enumeration) 12 )
#define MH_EC_TELETEX_TO_G4_CLASS1    ( (OM_enumeration) 13 )
#define MH_EC_TELETEX_TO_VIDEOTEX     ( (OM_enumeration) 14 )
#define MH_EC_VIDEOTEX_TO_TELEX       ( (OM_enumeration) 15 )
#define MH_EC_VIDEOTEX_TO_IA5_TEXT    ( (OM_enumeration) 16 )
#define MH_EC_VIDEOTEX_TO_TELETEX     ( (OM_enumeration) 17 )

/* Postal Mode */

#define MH_PM_ORDINARY_MAIL            ( (OM_enumeration) 0 )
#define MH_PM_SPECIAL_DELIVERY         ( (OM_enumeration) 1 )
#define MH_PM_EXPRESS_MAIL             ( (OM_enumeration) 2 )
#define MH_PM_CC                       ( (OM_enumeration) 3 )
#define MH_PM_CC_WITH_TELEPHONE_ADVICE ( (OM_enumeration) 4 )
#define MH_PM_CC_WITH_TELEX_ADVICE     ( (OM_enumeration) 5 )
#define MH_PM_CC_WITH_TELETEX_ADVICE   ( (OM_enumeration) 6 )

/* Postal Report */

#define MH_PR_UNDELIVBLE_MAIL_VIA_PDS   ( (OM_enumeration) 0 )
#define MH_PR_NOTIFICN_VIA_PDS          ( (OM_enumeration) 1 )
#define MH_PR_NOTIFICN_VIA_MTS          ( (OM_enumeration) 2 )
#define MH_PR_NOTIFICN_VIA_MTS_AND_PDS  ( (OM_enumeration) 3 )

/* Priority */

#define MH_PTY_NORMAL  ( (OM_enumeration) 0 )
#define MH_PTY_LOW     ( (OM_enumeration) 1 )
#define MH_PTY_URGENT  ( (OM_enumeration) 2 )

/* Reason */

#define MH_RE_TRANSFER_FAILED             ( (OM_enumeration) 0 )
#define MH_RE_TRANSFER_IMPOSSIBLE         ( (OM_enumeration) 1 )
#define MH_RE_CONVERSION_NOT_PERFORMED    ( (OM_enumeration) 2 )
#define MH_RE_PHYSICAL_RENDITN_NOT_DONE   ( (OM_enumeration) 3 )
#define MH_RE_PHYSICAL_DELIV_NOT_DONE     ( (OM_enumeration) 4 )
#define MH_RE_RESTRICTED_DELIVERY         ( (OM_enumeration) 5 )
#define MH_RE_DIRECTORY_OPERATN_FAILED    ( (OM_enumeration) 6 )

/* Redirection Reason */
```

```
#define MH_RR_RECIPIENT_ASSIGNED            ( (OM_enumeration) 0 )
#define MH_RR_ORIGINATOR_REQUESTED          ( (OM_enumeration) 1 )
#define MH_RR_RECIPIENT_DOMAIN_ASSIGNED     ( (OM_enumeration) 2 )

/* Registration */

#define MH_RG_UNREGISTERED_MAIL             ( (OM_enumeration) 0 )
#define MH_RG_REGISTERED_MAIL               ( (OM_enumeration) 1 )
#define MH_RG_REGISTERED_MAIL_IN_PERSON     ( (OM_enumeration) 2 )

/* Report Request */

#define MH_RQ_NEVER           ( (OM_enumeration) 0 )
#define MH_RQ_NON_DELIVERY    ( (OM_enumeration) 1 )
#define MH_RQ_ALWAYS          ( (OM_enumeration) 2 )
#define MH_RQ_ALWAYS_AUDITED  ( (OM_enumeration) 3 )

/* Security Classification */

#define MH_SC_UNMARKED       ( (OM_enumeration) 0 )
#define MH_SC_UNCLASSIFIED   ( (OM_enumeration) 1 )
#define MH_SC_RESTRICTED     ( (OM_enumeration) 2 )
#define MH_SC_CONFIDENTIAL   ( (OM_enumeration) 3 )
#define MH_SC_SECRET         ( (OM_enumeration) 4 )
#define MH_SC_TOP_SECRET     ( (OM_enumeration) 5 )

/* Terminal Type */

#define MH_TT_TELEX          ( (OM_enumeration) 3 )
#define MH_TT_TELETEX        ( (OM_enumeration) 4 )
#define MH_TT_G3_FAX         ( (OM_enumeration) 5 )
#define MH_TT_G4_FAX         ( (OM_enumeration) 6 )
#define MH_TT_IA5_TERMINAL   ( (OM_enumeration) 7 )
#define MH_TT_VIDEOTEX       ( (OM_enumeration) 8 )


/* Integer */

/* Content Type */

#define MH_CTI_UNIDENTIFIED   ( (OM_integer) 0 )
#define MH_CTI_EXTERNAL       ( (OM_integer) 1 )
#define MH_CTI_P2_1984        ( (OM_integer) 2 )
#define MH_CTI_P2_1988        ( (OM_integer) 22 )


/* Object Identifier (Elements component) */

/* Content Type */
#define OMP_O_MH_CTO_INNER_MESSAGE   "\x56\x03\x03\x01"
#define OMP_O_MH_CTO_UNIDENTIFIED    "\x56\x03\x03\x00"

/* External EITs */
#define OMP_O_MH_EE_G3_FAX           "\x56\x03\x04\x03"
#define OMP_O_MH_EE_G4_CLASS_1       "\x56\x03\x04\x04"
#define OMP_O_MH_EE_IA5_TEXT         "\x56\x03\x04\x02"
#define OMP_O_MH_EE_MIXED_MODE       "\x56\x03\x04\x09"
#define OMP_O_MH_EE_TELETEX          "\x56\x03\x04\x05"
#define OMP_O_MH_EE_TELEX            "\x56\x03\x04\x01"
#define OMP_O_MH_EE_UNDEFINED        "\x56\x03\x04\x00"
#define OMP_O_MH_EE_VIDEOTEX         "\x56\x03\x04\x06"

/* Rendition Attributes */
#define OMP_O_MH_RA_BASIC_RENDITION  "\x56\x03\x05\x00"

/* Type */
```

```
#define MH_T_A3_WIDTH                    ( (OM_type) 200 )
#define MH_T_ACTION                      ( (OM_type) 201 )
#define MH_T_ACTUAL_RECIPIENT_NAME       ( (OM_type) 202 )
#define MH_T_ADMD_NAME                   ( (OM_type) 203 )
#define MH_T_ALGORITHM_DATUM             ( (OM_type) 204 )
#define MH_T_ALGORITHM_ID                ( (OM_type) 205 )
#define MH_T_ALGORITHM_RESULT            ( (OM_type) 206 )
#define MH_T_ALTERNATE_RECIP_ALLOWED     ( (OM_type) 207 )
#define MH_T_ALTERNATE_RECIPIENT_NAME    ( (OM_type) 208 )
#define MH_T_ARRIVAL_TIME                ( (OM_type) 209 )
#define MH_T_ATTEMPTED_ADMD_NAME         ( (OM_type) 210 )
#define MH_T_ATTEMPTED_COUNTRY_NAME      ( (OM_type) 211 )
#define MH_T_ATTEMPTED_MTA_NAME          ( (OM_type) 212 )
#define MH_T_ATTEMPTED_PRMD_IDENTIFIER   ( (OM_type) 213 )
#define MH_T_B4_LENGTH                   ( (OM_type) 214 )
#define MH_T_B4_WIDTH                    ( (OM_type) 215 )
#define MH_T_BILATERAL_INFO              ( (OM_type) 216 )
#define MH_T_BINARY_CONTENT              ( (OM_type) 217 )
#define MH_T_BUILTIN_EITS                ( (OM_type) 218 )
#define MH_T_BUREAU_FAX_DELIVERY         ( (OM_type) 219 )
#define MH_T_COMMON_NAME                 ( (OM_type) 220 )
#define MH_T_CONFIDENTIALITY_ALGORITHM   ( (OM_type) 221 )
#define MH_T_CONFIDENTIALITY_KEY         ( (OM_type) 222 )
#define MH_T_CONTENT                     ( (OM_type) 223 )
#define MH_T_CONTENT_CORRELATOR          ( (OM_type) 224 )
#define MH_T_CONTENT_EXTENSIONS          ( (OM_type) 225 )
#define MH_T_CONTENT_IDENTIFIER          ( (OM_type) 226 )
#define MH_T_CONTENT_LENGTH              ( (OM_type) 227 )
#define MH_T_CONTENT_RETURN_REQUESTED    ( (OM_type) 228 )
#define MH_T_CONTENT_TYPE                ( (OM_type) 229 )
#define MH_T_CONTROL_CHARACTER_SETS      ( (OM_type) 230 )
#define MH_T_CONVERSION_LOSS_PROHIBITED  ( (OM_type) 231 )
#define MH_T_CONVERSION_PROHIBITED       ( (OM_type) 232 )
#define MH_T_CONVERTED_EITS              ( (OM_type) 233 )
#define MH_T_COUNTRY_NAME                ( (OM_type) 234 )
#define MH_T_CRITICAL_FOR_DELIVERY       ( (OM_type) 235 )
#define MH_T_CRITICAL_FOR_SUBMISSION     ( (OM_type) 236 )
#define MH_T_CRITICAL_FOR_TRANSFER       ( (OM_type) 237 )
#define MH_T_DEFERRED_DELIVERY_TIME      ( (OM_type) 238 )
#define MH_T_DEFERRED_TIME               ( (OM_type) 239 )
#define MH_T_DELIVERY_CONFIRMS           ( (OM_type) 240 )
#define MH_T_DELIVERY_POINT              ( (OM_type) 241 )
#define MH_T_DELIVERY_TIME               ( (OM_type) 242 )
#define MH_T_DIRECTORY_NAME              ( (OM_type) 243 )
#define MH_T_DISCLOSURE_ALLOWED          ( (OM_type) 244 )
#define MH_T_DISTINGUISHED_RECIP_ADDR    ( (OM_type) 245 )
#define MH_T_DOMAIN_TYPE_1               ( (OM_type) 246 )
#define MH_T_DOMAIN_TYPE_2               ( (OM_type) 247 )
#define MH_T_DOMAIN_TYPE_3               ( (OM_type) 248 )
#define MH_T_DOMAIN_TYPE_4               ( (OM_type) 249 )
#define MH_T_DOMAIN_VALUE_1              ( (OM_type) 250 )
#define MH_T_DOMAIN_VALUE_2              ( (OM_type) 251 )
#define MH_T_DOMAIN_VALUE_3              ( (OM_type) 252 )
#define MH_T_DOMAIN_VALUE_4              ( (OM_type) 253 )
#define MH_T_ENVELOPES                   ( (OM_type) 254 )
#define MH_T_EVENT_HANDLE                ( (OM_type) 255 )
#define MH_T_EXPANSION_HISTORY           ( (OM_type) 256 )
#define MH_T_EXPANSION_PROHIBITED        ( (OM_type) 257 )
#define MH_T_EXPLICIT_CONVERSION         ( (OM_type) 258 )
#define MH_T_EXTENSION_TYPE              ( (OM_type) 259 )
#define MH_T_EXTENSION_VALUE             ( (OM_type) 260 )
#define MH_T_EXTENSIONS                  ( (OM_type) 261 )
#define MH_T_EXTERNAL_EITS               ( (OM_type) 262 )
#define MH_T_EXTERNAL_TRACE_INFO         ( (OM_type) 263 )
#define MH_T_FINE_RESOLUTION             ( (OM_type) 264 )
#define MH_T_FORWARDING_ADDRESS          ( (OM_type) 265 )
#define MH_T_FORWARDING_ADDR_REQUESTED   ( (OM_type) 266 )
```

```
#define MH_T_FORWARDING_PROHIBITED        ( (OM_type) 267 )
#define MH_T_G3_FAX_NBPS                  ( (OM_type) 268 )
#define MH_T_G4_FAX_NBPS                  ( (OM_type) 269 )
#define MH_T_GENERATION                   ( (OM_type) 270 )
#define MH_T_GIVEN_NAME                   ( (OM_type) 271 )
#define MH_T_GRAPHIC_CHARACTER_SETS       ( (OM_type) 272 )
#define MH_T_INFORMATION                  ( (OM_type) 273 )
#define MH_T_INITIALS                     ( (OM_type) 274 )
#define MH_T_INTEGRITY_CHECK              ( (OM_type) 275 )
#define MH_T_INTENDED_RECIPIENT_NAME      ( (OM_type) 276 )
#define MH_T_INTENDED_RECIPIENT_NUMBER    ( (OM_type) 277 )
#define MH_T_INTERNAL_TRACE_INFO          ( (OM_type) 278 )
#define MH_T_ISDN_NUMBER                  ( (OM_type) 279 )
#define MH_T_ISDN_SUBADDRESS              ( (OM_type) 280 )
#define MH_T_LATEST_DELIVERY_TIME         ( (OM_type) 281 )
#define MH_T_LOCAL_IDENTIFIER             ( (OM_type) 282 )
#define MH_T_MESSAGE_SEQUENCE_NUMBER      ( (OM_type) 283 )
#define MH_T_MISCELANEOUS_CAPABILITIES    ( (OM_type) 284 )
#define MH_T_MTA_CERTIFICATE              ( (OM_type) 285 )
#define MH_T_MTA_NAME                     ( (OM_type) 286 )
#define MH_T_MTA_REPORT_REQUEST           ( (OM_type) 287 )
#define MH_T_MTA_RESPONSIBILITY           ( (OM_type) 288 )
#define MH_T_MTS_IDENTIFIER               ( (OM_type) 289 )
#define MH_T_NAME                         ( (OM_type) 290 )
#define MH_T_NON_DELIVERY_DIAGNOSTIC      ( (OM_type) 291 )
#define MH_T_NON_DELIVERY_REASON          ( (OM_type) 292 )
#define MH_T_NUMERIC_USER_IDENTIFIER      ( (OM_type) 293 )
#define MH_T_ORGANIZATION_NAME            ( (OM_type) 294 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_1   ( (OM_type) 295 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_2   ( (OM_type) 296 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_3   ( (OM_type) 297 )
#define MH_T_ORGANIZATIONAL_UNIT_NAME_4   ( (OM_type) 298 )
#define MH_T_ORIG_AND_EXPANSION_HISTORY   ( (OM_type) 299 )
#define MH_T_ORIGIN_CHECK                 ( (OM_type) 300 )
#define MH_T_ORIGINAL_EITS                ( (OM_type) 301 )
#define MH_T_ORIGINALLY_INTENDED_RECIP    ( (OM_type) 302 )
#define MH_T_ORIGINATOR_CERTIFICATE       ( (OM_type) 303 )
#define MH_T_ORIGINATOR_NAME              ( (OM_type) 304 )
#define MH_T_ORIGINATOR_REPORT_REQUEST    ( (OM_type) 305 )
#define MH_T_ORIGINATOR_RETURN_ADDRESS    ( (OM_type) 306 )
#define MH_T_OTHER_RECIPIENT_NAMES        ( (OM_type) 307 )
#define MH_T_PAGE_FORMATS                 ( (OM_type) 308 )
#define MH_T_PER_RECIP_REPORTS            ( (OM_type) 309 )
#define MH_T_POSTAL_ADDRESS_DETAILS       ( (OM_type) 310 )
#define MH_T_POSTAL_ADDRESS_IN_FULL       ( (OM_type) 311 )
#define MH_T_POSTAL_ADDRESS_IN_LINES      ( (OM_type) 312 )
#define MH_T_POSTAL_CODE                  ( (OM_type) 313 )
#define MH_T_POSTAL_COUNTRY_NAME          ( (OM_type) 314 )
#define MH_T_POSTAL_DELIVERY_POINT_NAME   ( (OM_type) 315 )
#define MH_T_POSTAL_DELIV_SYSTEM_NAME     ( (OM_type) 316 )
#define MH_T_POSTAL_GENERAL_DELIV_ADDR    ( (OM_type) 317 )
#define MH_T_POSTAL_LOCALE                ( (OM_type) 318 )
#define MH_T_POSTAL_MODE                  ( (OM_type) 319 )
#define MH_T_POSTAL_OFFICE_BOX_NUMBER     ( (OM_type) 320 )
#define MH_T_POSTAL_OFFICE_NAME           ( (OM_type) 321 )
#define MH_T_POSTAL_OFFICE_NUMBER         ( (OM_type) 322 )
#define MH_T_POSTAL_ORGANIZATION_NAME     ( (OM_type) 323 )
#define MH_T_POSTAL_PATRON_DETAILS        ( (OM_type) 324 )
#define MH_T_POSTAL_PATRON_NAME           ( (OM_type) 325 )
#define MH_T_POSTAL_REPORT                ( (OM_type) 326 )
#define MH_T_POSTAL_STREET_ADDRESS        ( (OM_type) 327 )
#define MH_T_PREFERRED_DELIVERY_MODES     ( (OM_type) 328 )
#define MH_T_PRESENTATION_ADDRESS         ( (OM_type) 329 )
#define MH_T_PRIORITY                     ( (OM_type) 330 )
#define MH_T_PRIVACY_MARK                 ( (OM_type) 331 )
#define MH_T_PRIVATE_USE                  ( (OM_type) 332 )
#define MH_T_PRMD_IDENTIFIER              ( (OM_type) 333 )
```

```
#define MH_T_PRMD_NAME                    ( (OM_type) 334 )
#define MH_T_PROOF_OF_DELIVERY            ( (OM_type) 335 )
#define MH_T_PROOF_OF_DELIV_REQUESTED     ( (OM_type) 336 )
#define MH_T_PROOF_OF_SUBMISSION          ( (OM_type) 337 )
#define MH_T_PROOF_OF_SUBMISN_REQUEST     ( (OM_type) 338 )
#define MH_T_PUBLIC_INFORMATION           ( (OM_type) 339 )
#define MH_T_RANDOM_NUMBER                ( (OM_type) 340 )
#define MH_T_REASON                       ( (OM_type) 341 )
#define MH_T_REASSIGNMENT_PROHIBITED      ( (OM_type) 342 )
#define MH_T_RECIPIENT_CERTIFICATE        ( (OM_type) 343 )
#define MH_T_RECIPIENT_DESCRIPTORS        ( (OM_type) 344 )
#define MH_T_RECIPIENT_NAME               ( (OM_type) 345 )
#define MH_T_RECIPIENT_NUMBER             ( (OM_type) 346 )
#define MH_T_RECIP_NUMBER_FOR_ADVICE      ( (OM_type) 347 )
#define MH_T_REDIRECTION_HISTORY          ( (OM_type) 348 )
#define MH_T_REGISTRATION                 ( (OM_type) 349 )
#define MH_T_RENDITION_ATTRIBUTES         ( (OM_type) 350 )
#define MH_T_REPORT_ADDITIONAL_INFO       ( (OM_type) 351 )
#define MH_T_REPORT_DESTINATION           ( (OM_type) 352 )
#define MH_T_REPORTING_DL_NAME            ( (OM_type) 353 )
#define MH_T_REPORTING_MTA_CERTIFICATE    ( (OM_type) 354 )
#define MH_T_SECRET_INFORMATION           ( (OM_type) 355 )
#define MH_T_SECURITY_CATEGORY_DATA       ( (OM_type) 356 )
#define MH_T_SECURITY_CATEGORY_IDS        ( (OM_type) 357 )
#define MH_T_SECURITY_CLASSIFICATION      ( (OM_type) 358 )
#define MH_T_SECURITY_LABEL               ( (OM_type) 359 )
#define MH_T_SECURITY_POLICY_ID           ( (OM_type) 360 )
#define MH_T_SIGNATURE                    ( (OM_type) 361 )
#define MH_T_SUBJECT_EXT_TRACE_INFO       ( (OM_type) 362 )
#define MH_T_SUBJECT_MTS_IDENTIFIER       ( (OM_type) 363 )
#define MH_T_SUBMISSION_TIME              ( (OM_type) 364 )
#define MH_T_SUPPLEMENTARY_INFO           ( (OM_type) 365 )
#define MH_T_SURNAME                      ( (OM_type) 366 )
#define MH_T_TELETEX_NBPS                 ( (OM_type) 367 )
#define MH_T_TEMPORARY                    ( (OM_type) 368 )
#define MH_T_TERMINAL_IDENTIFIER          ( (OM_type) 369 )
#define MH_T_TERMINAL_TYPE                ( (OM_type) 370 )
#define MH_T_TIME                         ( (OM_type) 371 )
#define MH_T_TOKEN                        ( (OM_type) 372 )
#define MH_T_TWO_DIMENSIONAL              ( (OM_type) 373 )
#define MH_T_UNCOMPRESSED                 ( (OM_type) 374 )
#define MH_T_UNLIMITED_LENGTH             ( (OM_type) 375 )
#define MH_T_WORKSPACE                    ( (OM_type) 376 )
#define MH_T_X121_ADDRESS                 ( (OM_type) 377 )


/* Value Length */

#define MH_VL_ADMD_NAME                 ( (OM_value_length) 16 )
#define MH_VL_ATTEMPTED_ADMD_NAME       ( (OM_value_length) 16 )
#define MH_VL_ATTEMPTED_COUNTRY_NAME    ( (OM_value_length) 3 )
#define MH_VL_ATTEMPTED_PRMD_IDENTIFIER ( (OM_value_length) 16 )
#define MH_VL_COMMON_NAME               ( (OM_value_length) 64 )
#define MH_VL_CONTENT_CORRELATOR        ( (OM_value_length) 512 )
#define MH_VL_CONTENT_IDENTIFIER        ( (OM_value_length) 16 )
#define MH_VL_COUNTRY_NAME              ( (OM_value_length) 3 )
#define MH_VL_DOMAIN_TYPE               ( (OM_value_length) 8 )
#define MH_VL_DOMAIN_VALUE              ( (OM_value_length) 128 )
#define MH_VL_GENERATION                ( (OM_value_length) 3 )
#define MH_VL_GIVEN_NAME                ( (OM_value_length) 16 )
#define MH_VL_INFORMATION               ( (OM_value_length) 1024 )
#define MH_VL_INITIALS                  ( (OM_value_length) 5 )
#define MH_VL_ISDN_NUMBER               ( (OM_value_length) 15 )
#define MH_VL_ISDN_SUBADDRESS           ( (OM_value_length) 40 )
#define MH_VL_LATEST_DELIVERY_TIME      ( (OM_value_length) 7 )
#define MH_VL_LOCAL_IDENTIFIER          ( (OM_value_length) 32 )
#define MH_VL_MSG_CONTENT_CORRELATOR    ( (OM_value_length) 16 )
```

```
#define MH_VL_MTA_NAME                  ( (OM_value_length) 32 )
#define MH_VL_NUMERIC_USER_IDENTIFIER   ( (OM_value_length) 32 )
#define MH_VL_ORGANIZATION_NAME         ( (OM_value_length) 64 )
#define MH_VL_ORGANIZATIONAL_UNIT_NAMES ( (OM_value_length) 32 )
#define MH_VL_POSTAL_ADDRESS_DETAILS    ( (OM_value_length) 30 )
#define MH_VL_POSTAL_ADDRESS_IN_FULL    ( (OM_value_length) 185 )
#define MH_VL_POSTAL_CODE               ( (OM_value_length) 16 )
#define MH_VL_POSTAL_COUNTRY_NAME       ( (OM_value_length) 32 )
#define MH_VL_POSTAL_DELIV_POINT_NAME   ( (OM_value_length) 30 )
#define MH_VL_POSTAL_DELIV_SYSTEM_NAME  ( (OM_value_length) 16 )
#define MH_VL_POSTAL_GENERAL_DELIV_ADDR ( (OM_value_length) 30 )
#define MH_VL_POSTAL_LOCALE             ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_BOX_NUMBER  ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_NAME        ( (OM_value_length) 30 )
#define MH_VL_POSTAL_OFFICE_NUMBER      ( (OM_value_length) 30 )
#define MH_VL_POSTAL_ORGANIZATION_NAME  ( (OM_value_length) 30 )
#define MH_VL_POSTAL_PATRON_DETAILS     ( (OM_value_length) 30 )
#define MH_VL_POSTAL_PATRON_NAME        ( (OM_value_length) 30 )
#define MH_VL_POSTAL_STREET_ADDRESS     ( (OM_value_length) 30 )
#define MH_VL_PRIVACY_MARK              ( (OM_value_length) 128 )
#define MH_VL_PRIVATE_USE               ( (OM_value_length) 126 )
#define MH_VL_PRMD_IDENTIFIER           ( (OM_value_length) 16 )
#define MH_VL_PRMD_NAME                 ( (OM_value_length) 16 )
#define MH_VL_RECIP_NUMBER_FOR_ADVICE   ( (OM_value_length) 32 )
#define MH_VL_REDIRECTION_TIME          ( (OM_value_length) 7 )
#define MH_VL_REPORT_ADDITIONAL_INFO    ( (OM_value_length) 1024 )
#define MH_VL_SUPPLEMENTARY_INFO        ( (OM_value_length) 64 )
#define MH_VL_SURNAME                   ( (OM_value_length) 40 )
#define MH_VL_TERMINAL_IDENTIFIER       ( (OM_value_length) 24 )
#define MH_VL_TIME                      ( (OM_value_length) 17 )
#define MH_VL_X121_ADDRESS              ( (OM_value_length) 15 )

/* Value Number */

#define MH_VN_BILATERAL_INFORMATION      ( (OM_value_number) 8 )
#define MH_VN_ENCODED_INFORMATION_TYPES  ( (OM_value_number) 8 )
#define MH_VN_EXPANSION_HISTORY          ( (OM_value_number) 512 )
#define MH_VN_OTHER_RECIPIENT_NAMES      ( (OM_value_number) 32767 )
#define MH_VN_PREFERRED_DELIVERY_MODES   ( (OM_value_number) 10 )
#define MH_VN_RECIPIENT_DESCRIPTORS      ( (OM_value_number) 32767 )
#define MH_VN_REDIRECTION_HISTORY        ( (OM_value_number) 512 )
#define MH_VN_REPORT_SUBSTANCE           ( (OM_value_number) 32767 )
#define MH_VN_SECURITY_CATEGORY_DATA     ( (OM_value_number) 64 )
#define MH_VN_SECURITY_CATEGORY_IDS      ( (OM_value_number) 64 )
#define MH_VN_TRACE_INFO                 ( (OM_value_number) 512 )

/* END MH PORTION OF INTERFACE */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*, *X/Open CAE Specification (November 1991), API to Electronic Mail (X.400)*.

xmsga.h

Purpose

Definitions for the message store general attributes

Synopsis

```
#include  <xom.h>
#include  <xds.h>
#include  <xdsmdup.h>
#include  <xmhp.h>
#include  <xmsga.h>
```

Description

The **xmsga.h** header declares the object identifiers for the message store general attributes. They are used with the directory message store object. This header must be included when use of the MHS directory user package (MDUP) has been negotiated.

All application programs that include this header must first include the **xom.h** object management header, the **xds.h** header, the **xdsmdup.h** and **xmhp.h** headers.

```
#ifndef XMSGA_HEADER
#define XMSGA_HEADER

/* MS General Attributes Package object identifier */

#define OMP_O_MS_GENERAL_ATTRIBUTES_PACKAGE "\x56\x06\x01\x02\x06\x02"


/* MS General Attributes Types */
/*
 * Note: Every client program must explicitly import into
 * every compilation unit (C source program) the classes or
 * Object Identifiers that it uses. Each of these classes or
 * Object Identifier names must then be explicitly exported from
 * just one compilation unit.
 * Importing and exporting can be done using the OM_IMPORT and
 * OM_EXPORT macros respectively (see [OM API]).
 * For instance, the client program uses
 *        OM_IMPORT( MS_A_CHILD_SEQUENCE_NUMBERS)
 * which in turn will make use of
 *        OMP_O_MS_A_CHILD_SEQUENCE_NUMBERS
 * defined below.
 */


#define OMP_O_MS_A_CHILD_SEQUENCE_NUMBERS      "\x56\x04\x03\x00"
#define OMP_O_MS_A_CONTENT                     "\x56\x04\x03\x01"
#define OMP_O_MS_A_CONTENT_CONFIDENTL_ALGM_ID  "\x56\x04\x03\x02"
#define OMP_O_MS_A_CONTENT_CORRELATOR          "\x56\x04\x03\x03"
#define OMP_O_MS_A_CONTENT_IDENTIFIER          "\x56\x04\x03\x04"
#define OMP_O_MS_A_CONTENT_INTEGRITY_CHECK     "\x56\x04\x03\x05"
#define OMP_O_MS_A_CONTENT_LENGTH              "\x56\x04\x03\x06"
#define OMP_O_MS_A_CONTENT_RETURNED            "\x56\x04\x03\x07"
#define OMP_O_MS_A_CONTENT_TYPE                "\x56\x04\x03\x08"
#define OMP_O_MS_A_CONVERSION_LOSS_PROHIBITED  "\x56\x04\x03\x09"
#define OMP_O_MS_A_CONVERTED_EITS              "\x56\x04\x03\x0A"
```

```
#define OMP_O_MS_A_CREATION_TIME               "\x56\x04\x03\x0B"
#define OMP_O_MS_A_DELIVERED_EITS              "\x56\x04\x03\x0C"
#define OMP_O_MS_A_DELIVERY_FLAGS              "\x56\x04\x03\x0D"
#define OMP_O_MS_A_DL_EXPANSION_HISTORY        "\x56\x04\x03\x0E"
#define OMP_O_MS_A_ENTRY_STATUS                "\x56\x04\x03\x0F"
#define OMP_O_MS_A_ENTRY_TYPE                  "\x56\x04\x03\x10"
#define OMP_O_MS_A_INTENDED_RECIPIENT_NAME     "\x56\x04\x03\x11"
#define OMP_O_MS_A_MESSAGE_DELIVERY_ENVELOPE   "\x56\x04\x03\x12"
#define OMP_O_MS_A_MESSAGE_DELIVERY_ID         "\x56\x04\x03\x13"
#define OMP_O_MS_A_MESSAGE_DELIVERY_TIME       "\x56\x04\x03\x14"
#define OMP_O_MS_A_MESSAGE_ORIGIN_AUTHEN_CHK   "\x56\x04\x03\x15"
#define OMP_O_MS_A_MESSAGE_SECURITY_LABEL      "\x56\x04\x03\x16"
#define OMP_O_MS_A_MESSAGE_SUBMISSION_TIME     "\x56\x04\x03\x17"
#define OMP_O_MS_A_MESSAGE_TOKEN               "\x56\x04\x03\x18"
#define OMP_O_MS_A_ORIGINAL_EITS               "\x56\x04\x03\x19"
#define OMP_O_MS_A_ORIGINATOR_CERTIFICATE      "\x56\x04\x03\x1A"
#define OMP_O_MS_A_ORIGINATOR_NAME             "\x56\x04\x03\x1B"
#define OMP_O_MS_A_OTHER_RECIPIENT_NAMES       "\x56\x04\x03\x1C"
#define OMP_O_MS_A_PARENT_SEQUENCE_NUMBER      "\x56\x04\x03\x1D"
#define OMP_O_MS_A_PERRECIP_REPORT_DELIV_FLDS  "\x56\x04\x03\x1E"
#define OMP_O_MS_A_PRIORITY                    "\x56\x04\x03\x1F"
#define OMP_O_MS_A_PROOF_OF_DELIVERY_REQUEST   "\x56\x04\x03\x20"
#define OMP_O_MS_A_REDIRECTION_HISTORY         "\x56\x04\x03\x21"
#define OMP_O_MS_A_REPORT_DELIVERY_ENVELOPE    "\x56\x04\x03\x22"
#define OMP_O_MS_A_REPORT_ORIGIN_AUTHEN_CHK    "\x56\x04\x03\x23"
#define OMP_O_MS_A_REPORTING_DL_NAME           "\x56\x04\x03\x24"
#define OMP_O_MS_A_REPORTING_MTA_CERTIFICATE   "\x56\x04\x03\x25"
#define OMP_O_MS_A_SECURITY_CLASSIFICATION     "\x56\x04\x03\x26"
#define OMP_O_MS_A_SEQUENCE_NUMBER             "\x56\x04\x03\x27"
#define OMP_O_MS_A_SUBJECT_SUBMISSION_ID       "\x56\x04\x03\x28"
#define OMP_O_MS_A_THIS_RECIPIENT_NAME         "\x56\x04\x03\x29"


/* Enumeration Constants */

/* for MS_A_ENTRY_STATUS */

#define MS_ES_NEW               ((OM_enumeration) 0)
#define MS_ES_LISTED            ((OM_enumeration) 1)
#define MS_ES_PROCESSED         ((OM_enumeration) 2)


/* for MS_A_ENTRY_TYPE */

#define MS_ET_DELIVERED_MESSAGE    ((OM_enumeration) 0)
#define MS_ET_DELIVERED_REPORT     ((OM_enumeration) 1)
#define MS_ET_RETURNED_CONTENT     ((OM_enumeration) 2)


/* for MS_A_PRIORITY */

#define MS_PTY_NORMAL          ((OM_enumeration) 0)
#define MS_PTY_LOW             ((OM_enumeration) 1)
#define MS_PTY_URGENT          ((OM_enumeration) 2)


/* for MS_A_SECURITY_CLASSIFICATION */

#define MS_SC_UNMARKED           ((OM_enumeration) 0)
#define MS_SC_UNCLASSIFIED       ((OM_enumeration) 1)
#define MS_SC_RESTRICTED         ((OM_enumeration) 2)
#define MS_SC_CONFIDENTIAL       ((OM_enumeration) 3)
#define MS_SC_SECRET             ((OM_enumeration) 4)
#define MS_SC_TOP_SECRET         ((OM_enumeration) 5)


#endif /* XMSGA_HEADER */
```

Related Information

X/Open CAE Specification (November 1991), API to Directory Services (XDS), *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*, *X/Open CAE Specification (November 1991), API to Electronic Mail (X.400)*.

xom_intro

Purpose

Introduction to X/OPEN OSI-Abstract-Data Manipulation (XOM) functions

Synopsis

```
#include  <xom.h>
#include  <xomext.h>
```

Description

This **xom_intro** reference page defines the functions of the C interface. The following table lists the relevant functions.

Table 33. Service Interface Functions—xom_intro(3xom)

Function	Description
omX_extract()	Gets attribute values from specified object.
omX_fill()	Initializes an OM_descriptor structure.
omX_fill_oid()	Initializes an OM_descriptor with an OID value.
omX_object_to_string()	Converts an OM_object to string format.
omX_string_to_object()	Converts a string to OM_object.
om_copy()	Copies a private object.
om_copy_value()	Copies a string between private objects.
om_create()	Creates a private object.
om_decode()	This function is not supported by the DCE XOM interface, and returns with an **OM_FUNCTION_DECLINED** error.
om_delete()	Deletes a private or service-generated object.
om_encode()	This function is not supported by the DCE XOM interface, and returns with an **OM_FUNCTION_DECLINED** error.
om_get()	Gets copies of attribute values from a private object.
om_instance()	Tests an object's class.
om_put()	Puts attribute values into a private object.
om_read()	Reads a segment of a string in a private object.
om_remove()	Removes attribute values from a private object.
om_write()	Writes a segment of a string into a private object.

As indicated in the table, the service interface comprises a number of functions whose purpose and range of capabilities are summarized as follows:

omX_extract()
: Creates a new public object that is an exact but independent copy of an existing subobject in a private object. This function is similiar to the **om_get()** function but includes an additional parameter *navigation_path* that contains directions to the required object to be extracted.

omX_fill()
Initializes an OM descriptor structure with user supplied values for its type, syntax and value.

omX_fill_oid()
Initializes an OM descriptor structure with user supplied values for its type and value. The syntax of the descriptor is always set to **OM_S_OBJECT_IDENTIFIER_STRING**.

omX_object_to_string()
Converts an OM object into a string format.

omX_string_to_object()
Creates a new private object, which is build from the *string* and *class* input parameters.

om_copy()
Creates an independent copy of an existing private object and all its subobjects. The copy is placed in the original's workspace, or in another specified by the XOM application.

om_copy_value()
Replaces an existing attribute value or inserts a new value in one private object with a copy of an existing attribute value found in another. Both values must be strings.

om_create()
Creates a new private object that is an instance of a particular class. The object can be initialized with the attribute values specified as initial in the class definition.

The service does not permit the API user to explicitly create instances of all classes, but rather only those indicated by a package's definition as having this property.

om_delete()
Deletes a service-generated public object, or makes a private object inaccessible.

om_get()
Creates a new public object that is an exact but independent copy of an existing private object. The client can request certain exclusions, each of which reduces the copy to a part of the original. The client can also request that values be converted from one syntax to another before they are returned.

The copy can exclude: attributes of types other than those specified, values at positions other than those specified within an attribute, the values of multivalued attributes, copies of (not handles for) subobjects, or all attribute values (revealing only an attribute's presence).

om_instance()
Determines whether an object is an instance of a particular class. The client can determine an object's class simply by inspection. This function is useful since it reveals that an object is an instance of a particular class, even if the object is an instance of a subclass of that class.

om_put()
Places or replaces in one private object copies of the attribute values of another public or private object.

The source values can be inserted before any existing destination values, before the value at a specified position in the destination attribute, or after any existing destination values. Alternatively, the source values can be substituted for any existing destination values or for the values at specified positions in the destination attribute.

om_read()
Reads a segment of a value of an attribute of a private object. The value must be a string. The value can first be converted from one syntax to another. The function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

om_remove()
Removes and discards particular values of an attribute of a private object. The attribute itself is removed if no values remain.

om_write()
Writes a segment of a value of an attribute to a private object. The value must be a string. The segment can first be converted from one syntax to another. The written segment becomes the value's last segment since any elements beyond it are discarded. The function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

In the C interface, the functions are realized by macros. The function prototype in the synopsis of a function's specification simply shows the client's view of the function.

The intent of the interface definition is that each function be atomic; that is, either it carries out its assigned task in full and reports success, or it fails to carry out even a part of the task and reports an exception. However, the service does not guarantee that a task is always carried out in full.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages. The possible error return values are described in the function reference pages.

XOM functions check for NULL pointers and return an error, except for workspace pointers. Pointers are only checked at the function interface. The check is only for NULL and not for validity. If NULL or invalid pointers are passed this may result in an undetermined behaviour.

omX_extract

Purpose

Extracts the first occurrence of the requested OM type from an object

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code  omX_extract(
   OM_private_object object
   OM_type_list navigation_path
   OM_exclusions exclusions
   OM_type_list included_types
   OM_boolean local_strings
   OM_value_position initial_value
   OM_value_position limiting_value
   OM_public_object *values
   OM_value_position *total_number);
```

Parameters

Input

object The object from which data is to be extracted.

navigation_path
Contains a NULL-terminated list of OM types that lead to the target object to be extracted. It does not include the OM type of the target object.

exclusions
Explicit requests for zero or more exclusions, each of which reduces the copy to a prescribed portion of the original. The exclusions apply to the attributes of the target object, but not to those of its subobjects.

Apart from **OM_NO_EXCLUSIONS**, each value is chosen from the following list. When multiple exclusions are specified, each is applied in the order in which it is displayed in the list with lower-numbered exclusions having precedence over higher-numbered exclusions. If, after the application of an exclusion, that portion of the object is not returned, no further exclusions need be applied to that portion.

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**

  The copy includes descriptors comprising only attributes of specified types. Note that this exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

- **OM_EXCLUDE_MULTIPLES**

  The copy includes a single descriptor for each attribute that has two or more values, rather than one descriptor for each value. None of these descriptors contains an attribute value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  If the attribute has values of two or more syntaxes, the descriptor identifies one of those syntaxes; however, the syntax identified is not specified.

  Note that this exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**

  The copy includes descriptors comprising only values at specified positions within an attribute. Note that, when this exclusion is used in conjunction with the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion, it provides a means for determining the values of a specified attribute, as well as the syntaxes of those values, one or more but not all attributes at a time.

- **OM_EXCLUDE_VALUES**

  The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  Note that this exclusion provides a means for determining an object's composition; that is, the type and syntax of each of its attribute values.

- **OM_EXCLUDE_SUBOBJECTS**

  The copy includes, for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls.

  Note that this exclusion provides a means for examining an object one level at a time.

- **OM_EXCLUDE_DESCRIPTORS**

  When this exclusion is specified, no descriptors are returned and the copy result is not present. The *total_number* parameter reflects the number of descriptors that would be returned by applying the other inclusion and exclusion specifications.

  Note that this exclusion provides an attribute analysis capability. For instance, the total number of values in a multivalued attribute can be determined by specifying an inclusion of the specific attribute type, and exclusions of **OM_EXCLUDE_DESCRIPTORS**, **OM_EXCLUDE_SUBOBJECTS**, and **OM_EXCLUDE_ALL_BUT_THESE_TYPES**.

The **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion affects the choice of descriptors, while the **OM_EXCLUDE_VALUES** exclusion affects the composition of descriptors.

included_types
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion is requested; it identifies the types of the attributes to be included in the copy (provided that they are displayed in the original).

local_strings
: This Boolean parameter indicates whether conversion to local string format should be carried out or not.

initial_value
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute of the first value to be included in the copy.

  If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

limiting_value
This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute one beyond that of the last value to be included in the copy. If this parameter is not greater than the *initial_value* parameter, no values are included (and no descriptors are returned).

If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Output

values The *values* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS** and the **OM_EXCLUDE_DESCRIPTORS** exclusion is not specified. It contains the array of OM descriptors extracted.

The memory space for *values* is provided by **omX_extract()**. It is the responsibility of the calling function to subsequently release this space through a call to **om_delete()**.

total_number
The number of attribute descriptors returned in the public object, but not in any of its subobjects, based on the inclusion and exclusion parameters specified. If the **OM_EXCLUDE_DESCRIPTORS** exclusion is specified, no *values* result is returned and the *total_number* result reflects the actual number of attribute descriptors that would be returned based on the remaining inclusion and exclusion values.

Note that the total includes only the attribute descriptors in the *values* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **omX_extract()** function creates a new public object that is an exact, but independent, copy of an existing subobject in a private object. It is similiar to the **om_get()** function but includes an additional parameter, *navigation_path* which contains directions to the required object to be extracted. The client can request certain exclusions, each of which reduces the copy to a part of the original.

One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the *values* parameter includes a descriptor that omits the elements (but not the length) of the string. The *elements* component of the *string* component in the descriptor's *value* component is **OM_ELEMENTS_UNSPECIFIED**, and the **OM_S_LONG_STRING** bit of the *syntax* component is set to **OM_TRUE**.

The parameters *exclusions*, *included_types*, *local_strings*, *initial_value*, and *limiting_value* only apply to the target object being extracted.

Note that the client can access long values by means of **om_read()**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

omX_fill

Purpose

Initializes an OM_descriptor structure

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code  omX_fill(
   OM_type type
   OM_syntax syntax
   OM_uint32 length
   void *elements
   OM_descriptor *destination);
```

Parameters

Input

type The type of OM descriptor structure.

syntax The syntax value for this OM descriptor.

length The data length for values of string syntax. Zero is entered for values of type **OM_object**. When initializing an **OM_descriptor** with an **OM_type** that has an **OM_syntax** of either **OM_S_INTEGER**, **OM_S_BOOLEAN** or **OM_S_ENUMERATION**, then the associated value must be entered in the *length* parameter.

elements
The string contents.

Output

destination
Contains the filled descriptor.

Description

The **omX_fill()** function is used to initialize an OM descriptor structure with user supplied values for its type, syntax, and value.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages. Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*.

omX_fill_oid

Purpose

Initializes an OM_descriptor structure with an object identifier value

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code  omX_fill_oid(
   OM_type type
   OM_object_identifier object_id
   OM_descriptor *destination);
```

Parameters

Input

type The type of **OM_descriptor** structure.

object_id
The object identifier value.

Output

destination
Contains the filled descriptor.

Description

The **omX_fill_oid()** function is used to initialize an OM descriptor structure with user-supplied values for its type and value. The syntax of the descriptor is always set to **OM_S_OBJECT_IDENTIFIER_STRING**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

omX_object_to_string

Purpose

Converts an OM object from descriptor to string format

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code  omX_object_to_string(
   OM_object object
   OM_boolean local_strings
   OM_string *string);
```

Parameters

Input

object Contains the OM object to be converted.

local_strings
This Boolean value indicates if the *string* return value should be converted to a local string format. For further information on local strings please refer to the *OSF DCE Application Development Guide—Directory Services*.

Output

string Contains the converted object in string format.

The calling function should provide the memory for *string*. The string's contents are initially unspecified. The string's length becomes the number of octets required to contain the segment that the function is to read. The service modifies this parameter. The string's elements become the elements actually read. The string's length becomes the number of octets required to hold the segment actually read.

Description

The **omX_object_to_string()** function converts an OM object into a string format. The object can either be a client-generated or a service-generated public or private object.

The objects that can be handled by this function are restricted to those defined in the schema file, **xoischema**. Additionally, the OM objects **DS_C_ATTRIBUTE_ERROR** and **DS_C_ERROR** are also handled. For these, a message string containing the error message is returned.

For the syntax of the output strings, please refer to the *OSF DCE Application Development Guide—Directory Services*.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

Errors

Refer to **xom.h(4xom)** and **xomext.h** for a list of the possible error values that can be returned in *OM_return_code*. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

omX_string_to_object

Purpose

Converts an OM object specified in string format to descriptor format

Synopsis

```
#include <xom.h>
#include <xomext.h>

OM_return_code  omX_string_to_object(
   OM_workspace workspace
   OM_string *string
   OM_object_identifier class
   OM_boolean local_strings
   OM_private_object *object
   OM_integer *error_position
   OM_integer *error_type);
```

Parameters

Input

workspace
The workspace pointer obtained from a **ds_initialize()** call.

string The string to be converted. Refer to the *OSF DCE Application Development Guide—Directory Services* for details of the string syntaxes allowed.

class The OM class of the object to be created.

local_strings
Indicates if the attribute values are to be converted from their local string format.

Output

object The converted object.

error_position
If there is a syntax error in the input string, then *error_position* indicates the position in the string where the error was detected.

error_type
Indicates the type of error. Refer to the **xomext.h** header file for explanations of the error types.

Description

The **omX_string_to_object()** function creates a new private object, which is built from the *string* and *class* input parameters.

The objects that can be created by this function are restricted to those defined in the schema file, **xoischema**.

Notes

The memory space for the *object* return parameter is allocated by **omX_string_to_object()**. The calling application is responsible for releasing this memory with the **om_delete()** function call.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in the **xom.h(4xom)** reference page.

  If there is a syntax error in the input string, *OM_return_code* is set to **OM_WRONG_VALUE_MAKEUP** and the type of error is returned in *error_type*.

Errors

Refer to **xom.h(4xom)** and **xomext.h** for a list of the possible error values that can be returned in *OM_return_code* and *error_type*. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

om_copy

Purpose

Creates a new private object that is an exact, but independent, copy of an existing private object

Synopsis

```
#include <xom.h>

OM_return_code  om_copy(
   OM_private_object original
   OM_workspace workspace
   OM_private_object *copy);
```

Parameters

Input

original
The original that remains accessible.

workspace
The workspace in which the copy is to be created. The original's class must be in a package associated with this workspace.

Output

copy The new copy of the private object. This result is present if and only if the return value for *OM_return_code* is **OM_SUCCESS**.

Description

The **om_copy()** function creates a new private object (the copy) that is an exact but independent copy of an existing private object (the original). The function is recursive in that copying the original also copies its subobjects.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_WORKSPACE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_TOO_MANY_VALUES**

om_copy_value

Purpose

Places or replaces a string in one private object with a copy of a string in another private object

Synopsis

```
#include <xom.h>

OM_return_code  om_copy_value(
   OM_private_object source
   OM_type source_type
   OM_value_position source_value_position
   OM_private_object destination
   OM_type destination_type
   OM_value_position destination_value_position);
```

Parameters

Input

source The source that remains accessible.

source_type
Identifies the type of an attribute. One of the attribute values is copied.

source_value_position
The position within the *source* attribute of the value copied.

destination
The destination that remains accessible.

destination_type
Identifies the type of the attribute. One of the attribute values is placed or replaced.

destination_value_position
The position within the *destination* attribute of the value placed or replaced. If the value position exceeds the number of values present in the *destination* attribute, the parameter is taken to be equal to that number.

Description

The **om_copy_value()** function places or replaces an attribute value in one private object (the destination) with a copy of an attribute value in another private object (the source). The source value is a string. The copy's syntax is that of the original.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the

function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_create

Purpose

Creates a new private object that is an instance of a particular class

Synopsis

```
#include <xom.h>

OM_return_code  om_create(
   OM_object_identifier class
   OM_boolean initialize
   OM_workspace workspace
   OM_private_object *object);
```

Parameters

Input

class Identifies the class of the object to be created. The specified class shall be concrete.

initialize
Determines whether the object created is initialized as specified in the definition of its class. If this parameter is **OM_TRUE**, the object is made to comprise the attribute values specified as initial values in the tabular definitions of the object's class and its superclasses. If this parameter is **OM_FALSE**, the object is made to comprise the **OM_CLASS** attribute alone.

workspace
The workspace in which the object is created. The specified class is in a package associated with this workspace.

Output

object The created object. This result is present if and only if the return value for *OM_return_code* is **OM_SUCCESS**.

Description

The **om_create()** function creates a new private object that is an instance of a particular class.

Notes

By subsequently adding new values to the object and replacing and removing existing values, the client can create all conceivable instances of the object's class.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

  The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_WORKSPACE**
- **OM_NOT_CONCRETE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_delete

Purpose

Deletes a private or service-generated object

Synopsis

```
#include <xom.h>

OM_return_code  om_delete(
   OM_object subject);
```

Parameters

Input

subject
The object to be deleted.

Description

The **om_delete()** function deletes a service-generated public object or makes a private object inaccessible. It is not intended for use on client-generated public objects.

If applied to a service-generated public object, the function deletes the object and releases any resources associated with the object, including the space occupied by descriptors and attribute values. The function is applied recursively to any public subobjects. This does not affect any private subobjects.

If applied to a private object, the function makes the object inaccessible. Any existing object handles for the object are invalidated. The function is applied recursively to any private subobjects.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**

- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_THE_SERVICES**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_get

Purpose

Creates a public copy of all or particular parts of a private object

Synopsis

```
#include <xom.h>

OM_return_code  om_get(
   OM_private_object original
   OM_exclusions exclusions
   OM_type_list included_types
   OM_boolean local_strings
   OM_value_position initial_value
   OM_value_position limiting_value
   OM_public_object *copy
   OM_value_position *total_number);
```

Parameters

Input

original

The original that remains accessible.

exclusions

Explicit requests for zero or more exclusions, each of which reduces the copy to a prescribed portion of the original. The exclusions apply to the attributes of the object, but not to those of its subobjects.

Apart from **OM_NO_EXCLUSIONS**, each value is chosen from the following list. When multiple exclusions are specified, each is applied in the order in which it is displayed in the list with lower-numbered exclusions having precedence over higher-numbered exclusions. If, after the application of an exclusion, that portion of the object is not returned, no further exclusions need be applied to that portion.

- **OM_EXCLUDE_ALL_BUT_THESE_TYPES**

  The copy includes descriptors comprising only attributes of specified types. Note that this exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

- **OM_EXCLUDE_MULTIPLES**

  The copy includes a single descriptor for each attribute that has two or more values, rather than one descriptor for each value. None of these descriptors contains an attribute value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  If the attribute has values of two or more syntaxes, the descriptor identifies one of those syntaxes; however, the syntax identified is not specified.

  Note that this exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

- **OM_EXCLUDE_ALL_BUT_THESE_VALUES**

  The copy includes descriptors comprising only values at specified positions within an attribute. Note that, when this exclusion is used in conjunction with the **OM_EXCLUDE_ALL_BUT_THESE_TYPES**

exclusion, it provides a means for determining the values of a specified attribute, as well as the syntaxes of those values, one or more but not all attributes at a time.

- **OM_EXCLUDE_VALUES**

  The copy includes a single descriptor for each attribute value, but the descriptor does not contain the value, and the **OM_S_NO_VALUE** bit of the syntax component is set.

  Note that this exclusion provides a means for determining an object's composition; that is, the type and syntax of each of its attribute values.

- **OM_EXCLUDE_SUBOBJECTS**

  The copy includes, for each value whose syntax is **OM_S_OBJECT**, a descriptor containing an object handle for the original private subobject, rather than a public copy of it. This handle makes that subobject accessible for use in subsequent function calls.

  Note that this exclusion provides a means for examining an object one level at a time.

- **OM_EXCLUDE_DESCRIPTORS**

  When this exclusion is specified, no descriptors are returned and the copy result is not present. The *total_number* parameter reflects the number of descriptors that would be returned by applying the other inclusion and exclusion specifications.

  Note that this exclusion provides an attribute analysis capability. For instance, the total number of values in a multivalued attribute can be determined by specifying an inclusion of the specific attribute type, and exclusions of **OM_EXCLUDE_DESCRIPTORS**, **OM_EXCLUDE_SUBOBJECTS**, and **OM_EXCLUDE_ALL_BUT_THESE_TYPES**.

The **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion affects the choice of descriptors, while the **OM_EXCLUDE_VALUES** exclusion affects the composition of descriptors.

included_types
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_TYPES** exclusion is requested; it identifies the types of the attributes to be included in the copy (provided that they are displayed in the original).

local_strings
: This Boolean parameter indicates whether conversion to local string format should be carried out or not. For further information on local strings please refer the *OSF DCE Application Development Guide—Directory Services*.

initial_value
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute of the first value to be included in the copy.

  If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

limiting_value
: This parameter is present if and only if the **OM_EXCLUDE_ALL_BUT_THESE_VALUES** exclusion is requested; it specifies the position within each attribute one beyond that of the last value

to be included in the copy. If this parameter is not greater than the *initial_value* parameter, no values are included (and no descriptors are returned).

If it is **OM_ALL_VALUES** or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Output

copy The *copy* parameter is only present if the return value from *OM_return_code* is **OM_SUCCESS** and the **OM_EXCLUDE_DESCRIPTORS** exclusion is not specified.

The space occupied by the public object and every attribute value that is a string is service provided. If the client alters any part of that space, the effect upon the service's subsequent behavior is unspecified.

total_number
The number of attribute descriptors returned in the public object, but not in any of its subobjects, based on the inclusion and exclusion parameters specified. If the **OM_EXCLUDE_DESCRIPTORS** exclusion is specified, no *copy* result is returned and the *total_number* result reflects the actual number of attribute descriptors that would be returned based on the remaining inclusion and exclusion values.

Note that the total includes only the attribute descriptors in the *copy* parameter. It excludes the special descriptor signaling the end of a public object.

Description

The **om_get()** function creates a new public object (the *copy*) that is an exact, but independent, copy of an existing private object, the *original* parameter. The client can request certain exclusions, each of which reduces the copy to a part of the original.

One exclusion is always requested implicitly. For each attribute value in the original that is a string whose length exceeds an implementation-defined number, the *copy* parameter includes a descriptor that omits the elements (but not the length) of the string. The *elements* component of the *string* component in the descriptor's *value* component is **OM_ELEMENTS_UNSPECIFIED**, and the **OM_S_LONG_STRING** bit of the *syntax* component is set to **OM_TRUE**.

Note that the client can access long values by means of **om_read()**.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_EXCLUSION**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_instance

Purpose

Determines whether an object is an instance of a particular class or any of its subclasses

Synopsis

```
#include <xom.h>

OM_return_code  om_instance(
   OM_object subject
   OM_object_identifier class
   OM_boolean *instance);
```

Parameters

Input

subject
The subject that remains accessible.

class Identifies the class in question.

Output

instance
Indicates whether the subject is an instance of the specified class or any of its subclasses. This result is present if and only if the value of the *OM_return_code* is set to **OM_SUCCESS**.

Description

The **om_instance()** function determines whether a service-generated public or private object (the subject) is an instance of a particular class or any of its subclasses.

Notes

The client can determine an object's class (**C**) by simply inspecting the object, using programming language constructs if the object is public or **om_get()** if it is private. This function is useful in that it reveals that an object is an instance of the specified class, even if **C** is a subclass of that class.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NOT_THE_SERVICES**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_put

Purpose

Places or replaces in one private object copies of the attribute values of another public or private object

Synopsis

```
#include <xom.h>

OM_return_code  om_put(
   OM_private_object destination
   OM_modification modification
   OM_object source
   OM_type_list included_types
   OM_value_position initial_value
   OM_value_position limiting_value);
```

Parameters

Input

destination
The destination that remains accessible. The destination's class is unaffected.

modification
The nature of the requested modification. The modification determines how **om_put()** uses the attribute values in the source to modify the object. In all cases, for each attribute present in the source, copies of its values are placed in the object's destination attribute of the same type. The data value is chosen from among the following:

- **OM_INSERT_AT_BEGINNING**

  The source values are inserted before any existing destination values. (The latter are retained.)

- **OM_INSERT_AT_CERTAIN_POINT**

  The source values are inserted before the value at a specified position in the destination attribute. (The latter are retained.)

- **OM_INSERT_AT_END**

  The source values are inserted after any existing destination values. (The latter are retained.)

- **OM_REPLACE_ALL**

  The source values are placed in the *destination* attribute. The existing destination values, if any, are discarded.

- **OM_REPLACE_CERTAIN_VALUES**

  The source values are substituted for the values at specified positions in the destination attribute. (The latter are discarded.)

source The source that remains accessible. The source's class is ignored. However, the attributes being copied from the source must be compatible with the destination's class definition.

included_types
If present, this parameter identifies the types of the attributes to be included

in the destination (provided that they are displayed in the source); otherwise, all attributes are to be included.

initial_value
This parameter is present if and only if the *modification* parameter is **OM_INSERT_AT_CERTAIN_POINT** or **OM_REPLACE_CERTAIN_VALUES**. It specifies the position within each destination attribute at which source values are inserted, or of the first value replaced, respectively.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in a destination attribute, the parameter is taken to be equal to that number.

limiting_value
Present if and only if the *modification* parameter is **OM_REPLACE_CERTAIN_VALUES**. It specifies the position within each destination attribute one beyond that of the last value replaced. If this parameter is present, it must be greater than the *initial_value* parameter.

If the *limiting_value* parameter is **OM_ALL_VALUES** or exceeds the number of values present in a destination attribute, the parameter is taken to be equal to that number.

Description

The **om_put()** function places or replaces in one private object (that is, the destination) copies of the attribute values of another public or private object (that is, the source). The client can specify that the source's values replace all or particular values in the destination, or are inserted at a particular position within each attribute. All string values being copied that are in the local representation are first converted into the nonlocal representation for that syntax (which may entail the loss of some information).

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**

- **OM_NO_SUCH_CLASS**
- **OM_NO_SUCH_MODIFICATION**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_CONCRETE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_TOO_MANY_VALUES**
- **OM_VALUES_NOT_ADJACENT**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_MAKEUP**
- **OM_WRONG_VALUE_NUMBER**
- **OM_WRONG_VALUE_POSITION**
- **OM_WRONG_VALUE_SYNTAX**
- **OM_WRONG_VALUE_TYPE**

om_read

Purpose

Reads a segment of a string in a private object

Synopsis

```
#include <xom.h>

OM_return_code  om_read(
   OM_private_object subject
   OM_type type
   OM_value_position value_position
   OM_boolean local_string
   OM_string_length *string_offset
   OM_string *elements);
```

Parameters

Input

subject
The subject that remains accessible.

type Identifies the type of the attribute, one of whose values is read.

value_position
The position within the attribute of the value read.

local_string
This Boolean parameter indicates whether conversion to local string format should be carried out or not. For further information on local strings please refer to the *OSF DCE Application Development Guide—Directory Services*.

Input/Output

string_offset
On input this parameter contains the offset, in octets, of the start of the string segment to be read. If it exceeds the total length of the string, the parameter is equal to the string length.

On output it contains the offset, in octets, of the start of the next string segment to be read, or 0 (zero) if the value's final segment is read. The result is present if, and only if, the *OM_return_code* is **OM_SUCCESS** . The value returned can be used as the input *string_offset* parameter in the next call of this function. This enables sequential reading of a value of a long string.

elements
On input, the space the client provides for the segment to be read. The string's contents are initially unspecified. The string's length is initially the number of octets required to contain the segment that the function is to read.

On output, the string's elements become the elements actually read. The string's length becomes the number of octets required to hold the segment actually read. This can be less than the initial length if the segment is the last in a long string.

Description

The **om_read()** function reads a segment of an attribute value in a private object, namely the subject.

The segment returned is a segment of the string value that is returned if the complete value is read in a single call.

Note that this function enables the client to read an arbitrarily long value without requiring that the service place a copy of the entire value in memory.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

  The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_SYNTAX**

om_remove

Purpose

Removes and discards values of an attribute of a private object

Synopsis

```
#include <xom.h>

OM_return_code  om_remove(
   OM_private_object subject
   OM_type type
   OM_value_position initial_value
   OM_value_position limiting_value);
```

Parameters

Input

subject
The subject that remains accessible. The subject's class is unaffected.

type Identifies the type of the attribute, some of whose values are removed. The type is not **OM_CLASS**.

initial_value
The position within the attribute of the first value removed.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in the attribute, the parameter is taken to be equal to that number.

limiting_value
The position within the attribute one beyond that of the last value removed. If this parameter is not greater than the *initial_value* parameter, no values are removed.

If it is **OM_ALL_VALUES**, or exceeds the number of values present in an attribute, the parameter is taken to be equal to that number.

Description

The **om_remove()** function removes and discards particular values of an attribute of a private object, the subject. If no values remain, the attribute itself is also removed. If the value is a subobject, the value is first removed and then **om_delete()** is applied to it, thus destroying the object.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed in this reference page.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**

om_write

Purpose

Writes a segment of a string into a private object

Synopsis

```
#include <xom.h>

OM_return_code  om_write(
   OM_private_object subject
   OM_type type
   OM_value_position value_position
   OM_syntax syntax
   OM_string_length *string_offset
   OM_string elements);
```

Parameters

Input

subject
The subject that remains accessible.

type Identifies the type of the attribute, one of whose values is written.

value_position
The position within the above attribute of the value to be written. The value position can neither be negative nor exceed the number of values present. If it equals the number of values present, the segment is inserted into the attribute as a new value.

syntax If the value being written is not already present in the subject, this identifies the syntax that the value has. It must be a permissible syntax for the attribute of which this is a value. If the value being written is already present in the subject, then that value's syntax is preserved and this parameter is ignored.

elements
The string segment to be written. A copy of this segment occupies a position within the string value being written, starting at the offset given by the *string_offset* input parameter. Any values already at or beyond this offset are discarded.

Input/Output

string_offset
On input this parameter contains the offset, in octets, of the start of the string segment to be written. If it exceeds the current length of the string value being written, the parameter is taken to be equal to that current length.

On output it contains the offset, in octets, after the last string segment written. This result is present if, and only if, the *OM_return_code* result is **OM_SUCCESS**. The value returned in *string_offset* can be used as the input *string_offset* parameter the next time this function is called. This enables sequential writing of the value of a long string.

Description

The **om_write()** function writes a segment of an attribute value in a private object, the *subject* parameter.

The segment supplied is a segment of the string value that is supplied if the complete value is written in a single call.

The written segment is made the value's last. The function discards any values whose offset equals or exceeds the *string_offset* result. If the value being written is in the local representation, it is converted to the nonlocal representation (which may entail the loss of information and which may yield a different number of elements than that provided).

Note that this function enables the client to write an arbitrarily long value without having to place a copy of the entire value in memory.

Return Values

The following describes a partial list of messages (or errors) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

OM_return_code
: Indicates whether the function succeeded and, if not, why not. If the function is successful, the value of *OM_return_code* is set to **OM_SUCCESS**; whereas, if the function fails, it has one of the values listed under **ERRORS**.

  The exact constants for *OM_return_code* are defined in the **xom.h** header file (see the **xom.h(4xom)** reference page later in this chapter).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

- **OM_FUNCTION_DECLINED**
- **OM_FUNCTION_INTERRUPTED**
- **OM_MEMORY_INSUFFICIENT**
- **OM_NETWORK_ERROR**
- **OM_NO_SUCH_OBJECT**
- **OM_NO_SUCH_SYNTAX**
- **OM_NO_SUCH_TYPE**
- **OM_NOT_PRESENT**
- **OM_NOT_PRIVATE**
- **OM_PERMANENT_ERROR**
- **OM_POINTER_INVALID**
- **OM_SYSTEM_ERROR**
- **OM_TEMPORARY_ERROR**
- **OM_WRONG_VALUE_LENGTH**
- **OM_WRONG_VALUE_MAKEUP**

- **OM_WRONG_VALUE_POSITION**
- **OM_WRONG_VALUE_SYNTAX**

xom.h

Purpose

Header file for XOM

Synopsis

#include <xom.h>

Description

The declarations, as assembled here, constitute the contents of a header file made accessible to client programmers. The header file includes by reference a second header file (**xomi.h**) comprising the declarations defining the C workspace interface. The **xomi.h** header file and the workspace interface are only used internally by the service interface, and are not visible to the client programmer.

```
#ifndef XOM_HEADER
#define XOM_HEADER

/* BEGIN SERVICE INTERFACE */

/* INTERMEDIATE DATA TYPES */

typedef int      OM_sint;
typedef short     OM_sint16;
typedef long int    OM_sint32;
typedef unsigned    OM_uint;
typedef unsigned short OM_uint16;
typedef long unsigned OM_uint32;

/* PRIMARY DATA TYPES */

/* Boolean */

typedef OM_uint32 OM_boolean;

/* String Length */

typedef OM_uint32 OM_string_length;

/* Enumeration */

typedef OM_sint32 OM_enumeration;

/* Exclusions */

typedef OM_uint OM_exclusions;

/* Integer */

typedef OM_sint32 OM_integer;

/* Modification */

typedef OM_uint OM_modification;

/* Object */

typedef struct OM_descriptor_struct *OM_object;
```

```
/* String */

typedef struct {
    OM_string_length   length;
    void  *elements;
} OM_string;

#define OM_STRING(string)    \
     { (OM_string_length)(sizeof(string)-1), string }

/* Workspace */

typedef void *OM_workspace;

/* SECONDARY DATA TYPES */

/* Object Identifier */

typedef OM_string OM_object_identifier;

/* Private Object */

typedef OM_object OM_private_object;

/* Public Object */

typedef OM_object OM_public_object;

/* Return Code */

typedef OM_uint OM_return_code;

/* Syntax */

typedef OM_uint16 OM_syntax;

/* Type */

typedef OM_uint16 OM_type;

/* Type List */

typedef OM_type *OM_type_list;

/* Value */

typedef struct {
    OM_uint32    padding;
    OM_object    object;
} OM_padded_object;

typedef union OM_value_union {
    OM_string    string;
    OM_boolean   boolean;
    OM_enumeration enumeration;
    OM_integer   integer;
    OM_padded_object    object;
} OM_value;

/* Value Length */

typedef OM_uint32 OM_value_length;

/* Value Position */

typedef OM_uint32 OM_value_position;
```

```
/* TERTIARY DATA TYPES */

/* Descriptor */

typedef struct OM_descriptor_struct {
    OM_type         type;
    OM_syntax        syntax;
    union OM_value_union  value;
} OM_descriptor;


/* SYMBOLIC CONSTANTS */

/* Boolean */

#define OM_FALSE    ((OM_boolean) 0)
#define OM_TRUE     ((OM_boolean) 1)

/* Element Position */

#define OM_LENGTH_UNSPECIFIED  ((OM_string_length) 0xFFFFFFFF)


/* Exclusions */

#define OM_NO_EXCLUSIONS          ((OM_exclusions) 0)
#define OM_EXCLUDE_ALL_BUT_THESE_TYPES   ((OM_exclusions) 1)
#define OM_EXCLUDE_ALL_BUT_THESE_VALUES  ((OM_exclusions) 2)
#define OM_EXCLUDE_MULTIPLES        ((OM_exclusions) 4)
#define OM_EXCLUDE_SUBOBJECTS       ((OM_exclusions) 8)
#define OM_EXCLUDE_VALUES         ((OM_exclusions) 16)
#define OM_EXCLUDE_DESCRIPTORS       ((OM_exclusions) 32)

/* Modification */

#define OM_INSERT_AT_BEGINNING       ((OM_modification) 1)
#define OM_INSERT_AT_CERTAIN_POINT     ((OM_modification) 2)
#define OM_INSERT_AT_END          ((OM_modification) 3)
#define OM_REPLACE_ALL           ((OM_modification) 4)
#define OM_REPLACE_CERTAIN_VALUES     ((OM_modification) 5)

/* Object Identifiers */

/* NOTE: These macros rely on the ## token-pasting operator of
 * ANSI C. On many pre-ANSI compilers the same effect can be
 * obtained by replacing ## with /**/

/* Private macro to calculate length of an object identifier
 */
#define OMP_LENGTH(oid_string) (sizeof(OMP_O_##oid_string)-1)


/* Macro to initialize the syntax and value of an object identifier
 */
#define OM_OID_DESC(type, oid_name)            \
    { (type), OM_S_OBJECT_IDENTIFIER_STRING,     \
    { { OMP_LENGTH(oid_name) , OMP_D_##oid_name } } }


/* Macro to mark the end of a client-allocated public object
 */
#define OM_NULL_DESCRIPTOR   \
 { OM_NO_MORE_TYPES, OM_S_NO_MORE_SYNTAXES, \
 { { 0, OM_ELEMENTS_UNSPECIFIED } } }
```

```
/* Macro to make class constants available
/* within a compilation unit
 */
#define OM_IMPORT(class_name)                \
        extern char OMP_D_##class_name [];     \
        extern OM_string class_name;


/* Macro to allocate memory for class constants
/* within a compilation unit
 */
#define OM_EXPORT(class_name)                \
    char OMP_D_##class_name[] = OMP_O_##class_name ;  \
    OM_string class_name =                \
      { OMP_LENGTH(class_name), OMP_D_##class_name } ;


/* Constant for the OM package
 */
/* { joint-iso-ccitt(2) mhs-motis(6) group(6) white(1)
   api(2) om(4) } */

#define OMP_O_OM_OM       "\x56\x06\x01\x02\x04"

/* Constant for the Encoding class
 */
#define OMP_O_OM_C_ENCODING   "\x56\x06\x01\x02\x04\x01"

/* Constant for the External class
 */
#define OMP_O_OM_C_EXTERNAL   "\x56\x06\x01\x02\x04\x02"

/* Constant for the Object class
 */
#define OMP_O_OM_C_OBJECT    "\x56\x06\x01\x02\x04\x03"

/* Constant for the BER Object Identifier
 */
#define OMP_O_OM_BER      "\x51\x01"

/* Constant for the Canonical-BER Object Identifier
 */
#define OMP_O_OM_CANONICAL_BER "\x56\x06\x01\x02\x04\x04"

/* Return Code */

#define OM_SUCCESS          ((OM_return_code) 0)
#define OM_ENCODING_INVALID     ((OM_return_code) 1)
#define OM_FUNCTION_DECLINED     ((OM_return_code) 2)
#define OM_FUNCTION_INTERRUPTED   ((OM_return_code) 3)
#define OM_MEMORY_INSUFFICIENT    ((OM_return_code) 4)
#define OM_NETWORK_ERROR       ((OM_return_code) 5)
#define OM_NO_SUCH_CLASS       ((OM_return_code) 6)
#define OM_NO_SUCH_EXCLUSION     ((OM_return_code) 7)
#define OM_NO_SUCH_MODIFICATION   ((OM_return_code) 8)
#define OM_NO_SUCH_OBJECT      ((OM_return_code) 9)
#define OM_NO_SUCH_RULES       ((OM_return_code) 10)
#define OM_NO_SUCH_SYNTAX      ((OM_return_code) 11)
#define OM_NO_SUCH_TYPE       ((OM_return_code) 12)
#define OM_NO_SUCH_WORKSPACE     ((OM_return_code) 13)
#define OM_NOT_AN_ENCODING      ((OM_return_code) 14)
#define OM_NOT_CONCRETE       ((OM_return_code) 15)
#define OM_NOT_PRESENT        ((OM_return_code) 16)
#define OM_NOT_PRIVATE        ((OM_return_code) 17)
#define OM_NOT_THE_SERVICES     ((OM_return_code) 18)
#define OM_PERMANENT_ERROR      ((OM_return_code) 19)
#define OM_POINTER_INVALID      ((OM_return_code) 20)
```

```
#define OM_SYSTEM_ERROR       ((OM_return_code) 21)
#define OM_TEMPORARY_ERROR      ((OM_return_code) 22)
#define OM_TOO_MANY_VALUES      ((OM_return_code) 23)
#define OM_VALUES_NOT_ADJACENT    ((OM_return_code) 24)
#define OM_WRONG_VALUE_LENGTH    ((OM_return_code) 25)
#define OM_WRONG_VALUE_MAKEUP    ((OM_return_code) 26)
#define OM_WRONG_VALUE_NUMBER    ((OM_return_code) 27)
#define OM_WRONG_VALUE_POSITION   ((OM_return_code) 28)
#define OM_WRONG_VALUE_SYNTAX    ((OM_return_code) 29)
#define OM_WRONG_VALUE_TYPE     ((OM_return_code) 30)

/* String (Elements component) */

#define OM_ELEMENTS_UNSPECIFIED   ((void *) 0)

/* Syntax */

#define OM_S_NO_MORE_SYNTAXES     ((OM_syntax) 0)
#define OM_S_BIT_STRING        ((OM_syntax) 3)
#define OM_S_BOOLEAN         ((OM_syntax) 1)
#define OM_S_ENCODING_STRING     ((OM_syntax) 8)
#define OM_S_ENUMERATION       ((OM_syntax) 10)
#define OM_S_GENERAL_STRING      ((OM_syntax) 27)
#define OM_S_GENERALISED_TIME_STRING ((OM_syntax) 24)
#define OM_S_GRAPHIC_STRING      ((OM_syntax) 25)
#define OM_S_IA5_STRING        ((OM_syntax) 22)
#define OM_S_INTEGER         ((OM_syntax) 2)
#define OM_S_NULL           ((OM_syntax) 5)
#define OM_S_NUMERIC_STRING      ((OM_syntax) 18)
#define OM_S_OBJECT          ((OM_syntax) 127)
#define OM_S_OBJECT_DESCRIPTOR_STRING ((OM_syntax) 7)
#define OM_S_OBJECT_IDENTIFIER_STRING ((OM_syntax) 6)
#define OM_S_OCTET_STRING       ((OM_syntax) 4)
#define OM_S_PRINTABLE_STRING     ((OM_syntax) 19)
#define OM_S_TELETEX_STRING      ((OM_syntax) 20)
#define OM_S_UTC_TIME_STRING     ((OM_syntax) 23)
#define OM_S_VIDEOTEX_STRING     ((OM_syntax) 21)
#define OM_S_VISIBLE_STRING      ((OM_syntax) 26)


#define OM_S_LONG_STRING       ((OM_syntax) 0x8000)
#define OM_S_NO_VALUE         ((OM_syntax) 0x4000)
#define OM_S_LOCAL_STRING       ((OM_syntax) 0x2000)
#define OM_S_SERVICE_GENERATED    ((OM_syntax) 0x1000)
#define OM_S_PRIVATE         ((OM_syntax) 0x0800)
#define OM_S_SYNTAX          ((OM_syntax) 0x03FF)

/* Type */

#define OM_NO_MORE_TYPES       ((OM_type) 0)
#define OM_ARBITRARY_ENCODING     ((OM_type) 1)
#define OM_ASN1_ENCODING       ((OM_type) 2)
#define OM_CLASS           ((OM_type) 3)
#define OM_DATA_VALUE_DESCRIPTOR   ((OM_type) 4)
#define OM_DIRECT_REFERENCE      ((OM_type) 5)
#define OM_INDIRECT_REFERENCE     ((OM_type) 6)
#define OM_OBJECT_CLASS        ((OM_type) 7)
#define OM_OBJECT_ENCODING      ((OM_type) 8)
#define OM_OCTET_ALIGNED_ENCODING   ((OM_type) 9)
#define OM_PRIVATE_OBJECT       ((OM_type) 10)
#define OM_RULES           ((OM_type) 11)

/* Value Position */

#define OM_ALL_VALUES     ((OM_value_position) 0xFFFFFFFF)

/* WORKSPACE INTERFACE */
```

```
#include <xomi.h>     /* Only for internal use by interface */

/* END SERVICE INTERFACE */
#endif  /* XOM_HEADER */
```

Related Information

Books: *X/Open CAE Specification (November 1991), API to Directory Services (XDS)*, *X/Open CAE Specification (November 1991), OSI-Abstract-Data Manipulation API (XOM)*, *OSF DCE Application Development Guide—Directory Services*.

Chapter 5. DCE Distributed Time Service

dts_intro

Purpose

Introduction to DCE Distributed Time Service (DTS)

Description

The DCE Distributed Time Service programming routines can obtain timestamps that are based on Coordinated Universal Time (UTC), translate between different timestamp formats, and perform calculations on timestamps. Applications can call the DTS routines from server or clerk systems and use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling.

The DTS routines can perform the following basic functions:

- Retrieve the current (UTC-based) time from DTS.
- Convert binary timestamps expressed in the **utc** time structure to or from **tm** structure components.
- Convert the binary timestamps expressed in the **utc** time structure to or from **timespec** structure components.
- Convert the binary timestamps expressed in the **utc** time structure to or from ASCII strings.
- Compare two binary time values.
- Calculate binary time values.
- Obtain time zone information.

DTS can convert between several types of binary time structures that are based on different calendars and time unit measurements. DTS uses UTC-based time structures, and can convert other types of time structures to its own presentation of UTC-based time.

Absolute time is an interval on a time scale; absolute time measurements are derived from system clocks or external time-providers. For DTS, absolute times reference the UTC standard and include the inaccuracy and other information. When you display an absolute time, DTS converts the time to ASCII text, as shown in the following display:

```
1992-11-21-13:30:25.785-04:00I000.082
```

Relative time is a discrete time interval that is often added to or subtracted from an absolute time. A TDF associated with an absolute time is one example of a relative time. Note that a relative time does not use the calendar date fields, since these fields concern absolute time.

UTC is the international time standard that DTS uses. The zero hour of UTC is based on the zero hour of Greenwich Mean Time (GMT). The documentation consistently refers to the time zone of the Greenwich Meridian as GMT. However, this time zone is also sometimes referred to as UTC.

The Time Differential Factor (TDF) is the difference between UTC and the time in a particular time zone.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime( )** system call, which is described in the **ctime(3)** reference page, provides additional information.)

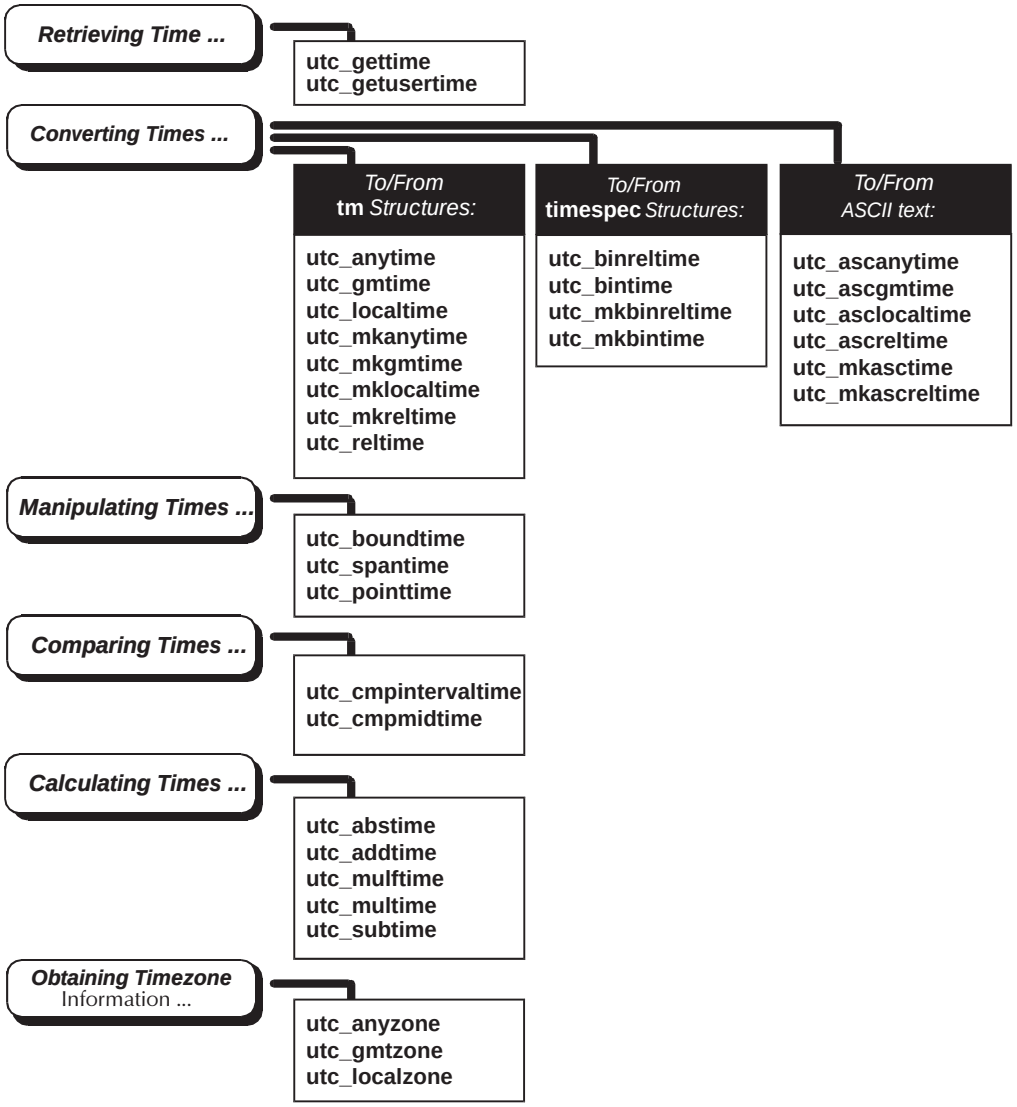
If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

The *OSF DCE Application Development Guide* provides additional information about UTC and GMT, TDF and time zones, and relative and absolute times.

Unless otherwise specified, the default input and output parameters are as follows:

- If NULL is specified for a *utc* input parameter, the current time is used.
- If NULL is specified for any output parameter, no result is returned.

The following illustration categorizes the DTS portable interface routines by function.



An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

utc_abstime( )
Computes the absolute value of a relative binary timestamp.

utc_addtime( )
Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time.

utc_anytime( )
Converts a binary timestamp to a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure.

utc_anyzone( )
Gets the time zone label and offset from GMT by using the TDF contained in the *utc* input parameter.

utc_ascanytime( )
Converts a binary timestamp to an ASCII string that represents an arbitrary time zone.

utc_ascgmtime( )
Converts a binary timestamp to an ASCII string that expresses a GMT time.

utc_asclocaltime( )
Converts a binary timestamp to an ASCII string that represents a local time.

utc_ascreltime( )
Converts a relative binary timestamp to an ASCII string that represents the time.

utc_binreltime( )
Converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy.

utc_bintime( )
Converts a binary timestamp to a **timespec** structure.

utc_boundtime( )
Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.

utc_cmpintervaltime( )
Compares two binary timestamps or two relative binary timestamps.

utc_cmpmidtime( )
Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.

utc_gettime( )
Returns the current system time and inaccuracy as a binary timestamp.

utc_getusertime( )
Returns the time and process-specific TDF, rather than the system-specific TDF.

utc_gmtime( )
Converts a binary timestamp to a **tm** structure that expresses GMT or the equivalent UTC.

utc_gmtzone( )
Gets the time zone label for GMT.

utc_localtime( )
Converts a binary timestamp to a **tm** structure that expresses local time.

utc_localzone( )
Gets the local time zone label and offset from GMT, given **utc**.

utc_mkanytime( )
Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.

utc_mkascreltime( )
Converts a NULL-terminated character string that represents a relative timestamp to a binary timestamp.

utc_mkasctime( )
Converts a NULL-terminated character string that represents an absolute timestamp to a binary timestamp.

utc_mkbinreltime( )
Converts a **timespec** structure expressing a relative time to a binary timestamp.

utc_mkbintime( )
Converts a **timespec** structure to a binary timestamp.

utc_mkgmtime( )
Converts a **tm** structure that expresses GMT or UTC to a binary timestamp.

utc_mklocaltime( )
Converts a **tm** structure that expresses local time to a binary timestamp.

utc_mkreltime( )
Converts a **tm** structure that expresses relative time to a relative binary timestamp.

utc_mulftime( )
Multiplies a relative binary timestamp by a floating-point value.

utc_multime( )
Multiplies a relative binary timestamp by an integer factor.

utc_pointtime( )
Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.

utc_reltime( )
Converts a relative binary timestamp to a **tm** structure.

utc_spantime( )
Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps.

utc_subtime( )
Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times.

Related Information

Books: *OSF DCE Application Development Guide—Core Components*.

utc_abstime

Purpose

Computes the absolute value of a relative binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_abstime(
   utc_t* result
   utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Absolute value of the input relative binary timestamp.

Description

The **utc_abstime()** routine computes the absolute value of a relative binary timestamp. The input timestamp represents a relative (delta) time.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

The following example scales a relative time, computes its absolute value, and prints the result.

```
utc_t    relutc, scaledutc;
char    timstr[UTC_MAX_STR_LEN];

/*
 *  Make sure relative timestamp represents a positive interval...
 */

utc_abstime(&relutc,    /* Out: Abs-value of rel time */
      &relutc);    /* In: Relative time to scale */

/*
 *  Scale it by a factor of 17...
 */

utc_multime(&scaledutc,   /* Out: Scaled relative time  */
      &relutc,    /* In: Relative time to scale */
      17L);        /* In: Scale factor      */
```

```
utc_ascreltime(timstr,     /* Out: ASCII relative time  */
        UTC_MAX_STR_LEN, /* In: Length of input string */
        &scaledutc); /* In: Relative time to    */
                /* convert           */

printf("%s\n",timstr);

/*
 *  Scale it by a factor of 17.65...
 */

utc_mulftime(&scaledutc,  /* Out: Scaled relative time  */
       &relutc,    /* In: Relative time to scale */
       17.65);      /* In: Scale factor      */

utc_ascreltime(timstr,     /* Out: ASCII relative time  */
        UTC_MAX_STR_LEN, /* In: Length of input string */
        &scaledutc); /* In: Relative time to    */
                /* convert           */

printf("%s\n",timstr);
```

utc_addtime

Purpose

Computes the sum of two binary timestamps

Synopsis

```
#include <dce/utc.h>

int  utc_addtime(
   utc_t* result
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Resulting binary timestamp or relative binary timestamp, depending upon the operation performed:

- *relative time*+*relative time*=*relative time*
- *absolute time*+*relative time*=*absolute time*
- *relative time*+*absolute time*=*absolute time*
- *absolute time*+*absolute time* is undefined. (See the note later in this reference page.)

Description

The **utc_addtime()** routine adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies. One or both of the input timestamps typically represents a relative (delta) time. The TDF in the first input timestamp is copied to the output. The timestamps can be two relative times or a relative time and an absolute time.

Notes

Although no error is returned, the combination *absolute time*+*absolute time* should *not* be used.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

The following example shows how to compute a timestamp that represents a time at least 5 seconds in the future.

```
utc_t        now, future, fivesec;
reltimespec_t    tfivesec;
timespec_t     tzero;

/*  Construct a timestamp that represents 5 seconds...
 */
tfivesec.tv_sec = 5;
tfivesec.tv_nsec = 0;
tzero.tv_sec = 0;
tzero.tv_nsec = 0;
utc_mkbinreltime(&fivesec, /* Out: 5 secs in binary timestamp  */
         &tfivesec, /* In: 5 secs in timespec      */
         &tzero);  /* In: 0 secs inaccuracy in timespec */

/*  Get the maximum possible current time...
 *  (The NULL input parameter is used to specify the current time.)
 */
utc_pointtime((utc_t *)0,   /* Out: Earliest possible current time */
       (utc_t *)0,   /* Out: Midpoint of current time    */
       &now,    /* Out: Latest possible current time  */
       (utc_t *)0);  /* In: Use current time        */

/*  Add 5 seconds to get future timestamp...
 */
utc_addtime(&future,   /* Out: Future binary timestamp   */
      &now,     /* In: Latest possible time now   */
      &fivesec);  /* In: 5 secs            */
```

Related Information

Functions: **utc_subtime(3dts)**.

utc_anytime

Purpose

Converts a binary timestamp to a tm structure

Synopsis

```
#include <dce/utc.h>

int  utc_anytime(
   struct tm *timetm
   long *tns
   struct tm *inacctm
   long *ins
   long *tdf
   utc_t *utc);
```

Parameters

Input

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp expressed in the timestamp's local time.

tns Nanoseconds since the time component of the binary timestamp.

inacctm
Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp.

tdf TDF component of the binary timestamp in units of seconds east of GMT.

Description

The **utc_anytime()** routine converts a binary timestamp to a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure. The TDF information contained in the timestamp is returned with the time and inaccuracy components; the TDF component determines the offset from GMT and the local time value of the **tm** structure. Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

The following example converts a timestamp by using the TDF information in the timestamp, and then prints the result.

```
utc_t        evnt;
struct tm      tmevnt;
timespec_t     tevnt, ievnt;
char        tznam[80];

/*  Assume evnt contains the timestamp to convert...
 *
 *  Get time as a tm structure, using the time zone information in
 *  the timestamp...
 */
utc_anytime(&tmevnt,    /* Out: tm struct of time of evnt  */
      (long *)0,     /* Out: nanosec of time of evnt   */
      (struct tm *)0,   /* Out: tm struct of inacc of evnt  */
      (long *)0,     /* Out: nanosec of inacc of evnt   */
      (int *)0,      /* Out: tdf of evnt         */
      &evnt);     /* In: binary timestamp of evnt   */

/*  Get the time and inaccuracy as timespec structures...
 */
utc_bintime(&tevnt,     /* Out: timespec of time of evnt   */
      &ievnt,     /* Out: timespec of inacc of evnt  */
      (int *)0,      /* Out: tdf of evnt         */
      &evnt);     /* In: Binary timestamp of evnt   */

/*  Construct the time zone name from time zone information in the
 *  timestamp...
 */

utc_anyzone(tznam, /* Out: Time zone name        */
      80,         /* In: Size of time zone name    */
      (long *)0,     /* Out: tdf of event         */
      (long *)0,     /* Out: Daylight saving flag     */
      &evnt);     /* In: Binary timestamp of evnt   */

/*  Print timestamp in the format:
 *
 *      1991-03-05-21:27:50.023I0.140 (GMT-5:00)
 *      1992-04-02-12:37:24.003Iinf (GMT+7:00)
 */

printf("%d-%02d-%02d-%02d:%02d:%02d.%03d",
    tmevnt.tm_year+1900, tmevnt.tm_mon+1, tmevnt.tm_mday,
    tmevnt.tm_hour, tmevnt.tm_min, tmevnt.tm_sec,
    (tevnt.tv_nsec/1000000));

if ((long)ievnt.tv_sec == -1)
  printf("Iinf");
else
  printf("I%d.%03d", ievnt.tv_sec, (ievnt.tv_nsec/1000000));

printf(" (%s)\n", tznam);
```

Related Information

Functions: **utc_anyzone(3dts), utc_gettime(3dts), utc_getusertime(3dts), utc_gmtime(3dts), utc_localtime(3dts), utc_mkanytime(3dts).**

utc_anyzone

Purpose

Gets the time zone label and offset from GMT

Synopsis

```
#include <dce/utc.h>

int  utc_anyzone(
   char *tzname
   size_t tzlen
   long *tdf
   int *isdst
   const utc_t *utc);
```

Parameters

Input

tzlen Length of the *tzname* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

tzname
Character string that is long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT.

isdst Integer with a value of −1, indicating that no information is supplied as to whether it is standard time or daylight saving time. A value of −1 is always returned.

Description

The **utc_anyzone()** routine gets the time zone label and offset from GMT by using the TDF contained in the *utc* input parameter. The label returned is always of the form GMT+*n* or GMT-*n* where *n* is the *tdf* expressed in *hours***:** *minutes*. (The label associated with an arbitrary time zone is not known; only the offset is known.)

Notes

All of the output parameters are optional. No value is returned and no error occurs if the pointer is NULL.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or an insufficient buffer.

Examples

See the sample program in the **utc_anytime(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts)**, **utc_gmtzone(3dts)**, **utc_localzone(3dts)**.

utc_ascanytime

Purpose

Converts a binary timestamp to an ASCII string that represents an arbitrary time zone

Synopsis

```
#include <dce/utc.h>

int  utc_ascanytime(
   char *cp
   size_t stringlen
   utc_t *utc);
```

Parameters

Input

stringlen
The length of the *cp* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

cp ASCII string that represents the time.

Description

The **utc_ascanytime()** routine converts a binary timestamp to an ASCII string that expresses a time. The TDF component in the timestamp determines the local time used in the conversion.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts a time to an ASCII string that expresses the time in the time zone where the timestamp was generated.

```
utc_t   evnt;
char    localTime[UTC_MAX_STR_LEN];

/*
 * Assuming that evnt contains the timestamp to convert, convert
 * the time to ASCII in the following format:
 *
 *     1991-04-01-12:27:38.37-8:00I2.00
 */
```

```
utc_ascanytime(localtime,     /* Out: Converted time  */
        UTC_MAX_STR_LEN,  /* In: Length of string */
        &evnt);    /* In: Time to convert  */
```

Related Information

Functions: **utc_ascgmtime(3dts)**, **utc_asclocaltime(3dts)**.

utc_ascgmtime

Purpose

Converts a binary timestamp to an ASCII string that expresses a GMT time

Synopsis

```
#include <dce/utc.h>

int  utc_ascgmtime(
   char *cp
   size_t stringlen
   utc_t *utc);
```

Parameters

Input

stringlen
Length of the *cp* buffer.

utc Binary timestamp.

Output

cp ASCII string that represents the time.

Description

The **utc_ascgmtime()** routine converts a binary timestamp to an ASCII string that expresses a time in GMT.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts the current time to GMT format.

```
char  gmTime[UTC_MAX_STR_LEN];

/*  Convert the current time to ASCII in the following format:
 *     1991-04-01-12:27:38.37I2.00
 */
utc_ascgmtime(gmTime,      /* Out: Converted time   */
       UTC_MAX_STR_LEN,  /* In: Length of string  */
       (utc_t*) NULL);  /* In: Time to convert   */
                 /* Default is current time */
```

Related Information

Functions: **utc_ascanytime(3dts),utc_asclocaltime(3dts)**.

utc_asclocaltime

Purpose

Converts a binary timestamp to an ASCII string that represents a local time

Synopsis

```
#include <dce/utc.h>

int  utc_asclocaltime(
   char *cp
   size_t stringlen
   utc_t *utc);
```

Parameters

Input

stringlen
: Length of the *cp* buffer.

utc
: Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

cp
: ASCII string that represents the time.

Description

The **utc_asclocaltime()** routine converts a binary timestamp to an ASCII string that expresses local time.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime** () system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Return Values

0
: Indicates that the routine executed successfully.

−1
: Indicates an invalid time parameter or invalid results.

Examples

The following example converts the current time to local time.

```
char  localTime[UTC_MAX_STR_LEN];

/* Convert the current time...
 */
```

```
utc_asclocaltime(localTime,    /* Out: Converted time   */
         UTC_MAX_STR_LEN, /* In:  Length of string  */
         (utc_t*) NULL);  /* In:  Time to convert   */
                  /* Default is current time  */
```

Related Information

Functions: **utc_ascanytime(3dts)**, **utc_ascgmtime(3dts)**.

utc_ascreltime

Purpose

Converts a relative binary timestamp to an ASCII string that represents the time

Synopsis

```
#include <dce/utc.h>

int  utc_ascreltime(
   char *cp
   const size_t stringlen
   utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp.

stringlen
Length of the *cp* buffer.

Output

cp ASCII string that represents the time.

Description

The **utc_ascreltime()** routine converts a relative binary timestamp to an ASCII string that represents the time.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid results.

Examples

See the sample program in the **utc_abstime(3dts)** reference page.

Related Information

Functions: **utc_mkascreltime(3dts)**.

utc_binreltime

Purpose

Converts a relative binary timestamp to two timespec structures that express relative time and inaccuracy

Synopsis

```
#include <dce/utc.h>

int  utc_binreltime(
   reltimespec_t *timesp
   timespec_t *inaccsp
   utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timesp Time component of the relative binary timestamp, in the form of seconds and nanoseconds since the base time (1970−01−01:00:00:00.0+00:00I0).

inaccsp
Inaccuracy component of the relative binary timestamp, in the form of seconds and nanoseconds.

Description

The **utc_binreltime()** routine converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy. These **timespec** structures describe a time interval.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

The following example measures the duration of a process, then prints the resulting relative time and inaccuracy.

```
utc_t        before, duration;
reltimespec_t    tduration;
timespec_t     iduration;

/*  Get the time before the start of the operation...
 */
utc_gettime(&before);      /* Out: Before binary timestamp   */

/*  ...Later...
 *  Subtract, getting the duration as a relative time.
```

```
 *
 *     NOTE: The NULL argument is used to obtain the current time.
 */

utc_subtime(&duration,     /* Out: Duration rel bin timestamp */
      (utc_t *)0,     /* In: After binary timestamp   */
      &before);      /* In: Before binary timestamp   */

/*  Convert the relative times to timespec structures...
 */

utc_binreltime(&tduration,   /* Out: Duration time timespec   */
        &iduration,   /* Out: Duration inacc timespec   */
        &duration);   /* In: Duration rel bin timestamp */

/*  Print the duration...
 */
printf("%d.%04d", tduration.tv_sec, (tduration.tv_nsec/10000));

if ((long)iduration.tv_sec == -1)
  printf("Iinf\n");
else
  printf("I%d.%04d\n", iduration.tv_sec, (iduration.tv_nsec/100000));
```

Related Information

Functions: **utc_mkbinreltime(3dts)**.

utc_bintime

Purpose

Converts a binary timestamp to a timespec structure

Synopsis

```
#include <dce/utc.h>

int  utc_bintime(
   timespec_t *timesp
   timespec_t *inaccsp
   long *tdf
   utc_t *utc);
```

Parameters

Input

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timesp Time component of the binary timestamp, in the form of seconds and nanoseconds since the base time.

inaccsp
Inaccuracy component of the binary timestamp, in the form of seconds and nanoseconds.

tdf TDF component of the binary timestamp in the form of signed number of seconds east of GMT.

Description

The **utc_bintime()** routine converts a binary timestamp to a **timespec** structure. The TDF information contained in the timestamp is returned.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_anytime(3dts)** reference page.

Related Information

Functions: **utc_binreltime(3dts)**, **utc_mkbintime(3dts)**.

utc_boundtime

Purpose

Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event

Synopsis

```
#include <dce/utc.h>

int  utc_boundtime(
   utc_t *result
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Before binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 After binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Spanning timestamp.

Description

Given two UTC times, the **utc_boundtime()** routine returns a single UTC time whose inaccuracy bounds the two input times. This is useful for timestamping events: the routine gets the **utc** values before and after the event, then calls **utc_boundtime()** to build a timestamp that includes the event.

Notes

The TDF in the output UTC value is copied from the *utc2* input parameter. If one or both input values have unspecified inaccuracies, the returned time value also has an unspecified inaccuracy and is the average of the two input values.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time parameter or invalid parameter order.

Examples

The following example records the time of an event and constructs a single timestamp, which includes the time of the event. Note that the **utc_getusertime()** routine is called so the time zone information that is included in the timestamp references the user's environment rather than the system's default time zone.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime** () system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

```
utc_t        before, after, evnt;

/* Get the time before the event...
 */
utc_getusertime(&before);    /* Out: Before binary timestamp   */

/* Get the time after the event...
 */
utc_getusertime(&after);    /* Out: After binary timestamp   */

/* Construct a single timestamp that describes the time of the
 *  event...
 */
utc_boundtime(&evnt,      /* Out: Timestamp that bounds event */
       &before,     /* In: Before binary timestamp   */
       &after);     /* In: After binary timestamp
*/
```

Related Information

Functions: **utc_gettime(3dts)**, **utc_pointtime(3dts)**, **utc_spantime(3dts)**.

utc_cmpintervaltime

Purpose

Compares two binary timestamps or two relative binary timestamps

Synopsis

```
#include <dce/utc.h>

int  utc_cmpintervaltime(
   enum utc_cmptype *relation
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

relation

Receives the result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

- **utc_equalTo**
- **utc_lessThan**
- **utc_greaterThan**
- **utc_indeterminate**

Description

The **utc_cmpintervaltime()** routine compares two binary timestamps and returns a flag indicating that the first time is greater than, less than, equal to, or overlapping with the second time. Two times overlap if the intervals (*time* − *inaccuracy*, *time* + *inaccuracy*) of the two times intersect.

The input binary timestamps express two absolute or two relative times. Do *not* compare relative binary timestamps to absolute binary timestamps. If you do, no meaningful results and no errors are returned.

The following routine does a temporal ordering of the time intervals.

```
utc1 is utc_lessThan utc2 iff
    utc1.time + utc1.inacc < utc2.time - utc2.inacc

utc1 is utc_greaterThan utc2 iff
    utc1.time - utc1.inacc > utc2.time + utc2.inacc
```

```
utc1 utc_equalTo utc2 iff
    utc1.time == utc2.time and
    utc1.inacc == 0 and
    utc2.inacc == 0
```

utc1 is **utc_indeterminate** with respect to **utc2** if the intervals overlap.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument.

Examples

The following example checks to see if the current time is definitely after 13:00 local time.

```
struct tm      tmtime, tmzero;
enum utc_cmptype  relation;
utc_t        testtime;

/*  Zero the tm structure for inaccuracy...
 */
memset(&tmzero, 0, sizeof(tmzero));

/* Get the current time, mapped to a tm structure...
 *
 *    NOTE: The NULL argument is used to get the current time.
 */
utc_gmtime(&tmtime,     /* Out: Current GMT time in tm struct  */
      (long *)0,    /* Out: Nanoseconds of time       */
      (struct tm *)0, /* Out: Current inaccuracy in tm struct */
      (long *)0,    /* Out: Nanoseconds of inaccuracy    */
      (utc_t *)0);   /* In: Current timestamp        */

/*  Alter the tm structure to correspond to 13:00 local time    */
 */

tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

/* Convert to a binary timestamp...
 */
utc_mkgmtime(&testtime,   /* Out: Binary timestamp of 13:00    */
       &tmtime,    /* In: 1:00 PM in tm struct      */
       0,       /* In: Nanoseconds of time       */
       &tmzero,    /* In: Zero inaccuracy in tm struct  */
       0);      /* In: Nanoseconds of inaccuracy    */

/*  Compare to the current time. Note the use of the NULL argument */
 */
utc_cmpintervaltime(&relation,  /* Out: Comparison relation    */
          (utc_t *)0,  /* In: Current timestamp     */
          &testtime);  /* In: 13:00 PM timestamp    */

/*  If it is not later - wait, print a message, etc.
 */

if (relation != utc_greaterThan) {

/*
 *   Note: It could be earlier than 13:00 local time or it could be
```

```
 *      indeterminate. If indeterminate, for some applications
 *      it might be worth waiting.
 */
}
```

Related Information

Functions: **utc_cmpmidtime(3dts)**.

utc_cmpmidtime

Purpose

Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies

Synopsis

```
#include <dce/utc.h>

int  utc_cmpmidtime(
   enum utc_cmptype *relation
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

relation

Result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

- **utc_equalTo**
- **utc_lessThan**
- **utc_greaterThan**

Description

The **utc_cmpmidtime()** routine compares two binary timestamps and returns a flag indicating that the first timestamp is greater than, less than, or equal to the second timestamp. Inaccuracy information is ignored for this comparison; the input values are therefore equivalent to the midpoints of the time intervals described by the input binary timestamps.

The input binary timestamps express two absolute or two relative times. Do *not* compare relative binary timestamps to absolute binary timestamps. If you do, no meaningful results and no errors are returned.

The following routine does a lexical ordering on the time interval midpoints.

```
utc1 is utc_lessThan utc2 iff
    utc1.time < utc2.time

utc1 is utc_greaterThan utc2 iff
    utc1.time > utc2.time

utc1 is utc_equalTo utc2 iff
    utc1.time == utc2.time
```

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument.

Examples

The following example checks if the current time (ignoring inaccuracies) is after 13:00 local time.

```
struct tm      tmtime, tmzero;
enum utc_cmptype  relation;
utc_t        testtime;

/*  Zero the tm structure for inaccuracy...
 */
memset(&tmzero, 0, sizeof(tmzero));

/*  Get the current time, mapped to a tm structure...
 *
 *    NOTE: The NULL argument is used to get the current time.
 */
utc_localtime(&tmtime,    /* Out: Current local time in tm struct */
       (long *)0,    /* Out: Nanoseconds of time       */
       (struct tm *)0, /* Out: Current inacc in tm struct   */
       (long *)0,    /* Out: Nanoseconds of inaccuracy    */
       (utc_t *)0);   /* In: Current timestamp        */


 /*  Alter the tm structure to correspond to 13:00 local time.
 */
tmtime.tm_hour = 13;
tmtime.tm_min = 0;
tmtime.tm_sec = 0;

 /*  Convert to a binary timestamp...
 */
utc_mklocaltime(&testtime,  /* Out: Binary timestamp of 13:00    */
        &tmtime,   /* In: 13:00 in tm struct       */
        0,      /* In: Nanoseconds of time       */
        &tmzero,   /* In: Zero inaccuracy in tm struct  */
        0);      /* In: Nanoseconds of inaccuracy    */

 /*  Compare to the current time. Note the use of the NULL argument
 */
utc_cmpmidtime(&relation,   /* Out: Comparison relation       */
        (utc_t *)0,  /* In: Current timestamp        */
        &testtime);  /* In: 13:00 local time timestamp   */

 /*  If the time is not later - wait, print a message, etc.
 */
if (relation != utc_greaterThan) {

 /*     It is not later then 13:00 local time. Note that
 *     this depends on the setting of the user's environment.
 */
}
```

Related Information

Functions: **utc_cmpintervaltime(3dts)**.

utc_gettime

Purpose

Returns the current system time and inaccuracy as a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_gettime(
   utc_t *utc);
```

Parameters

Input

None.

Output

utc System time as a binary timestamp.

Description

The **utc_gettime()** routine returns the current system time and inaccuracy in a binary timestamp. The routine takes the TDF from the operating system's kernel; the TDF is specified in a system-dependent manner.

Return Values

0 Indicates that the routine executed successfully.

−1 Generic error that indicates the time service cannot be accessed.

Examples

See the sample program in the **utc_binreltime(3dts)** reference page.

utc_getusertime

Purpose

Returns the time and process-specific TDF, rather than the system-specific TDF

Synopsis

```
#include <dce/utc.h>

int  utc_getusertime(
   utc_t *utc);
```

Parameters

Input

None.

Output

utc System time as a binary timestamp.

Description

The **utc_getusertime()** routine returns the system time and inaccuracy in a binary timestamp. The routine takes the TDF from the user's environment, which determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime()** system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user environment does not specify a TDF, the system's TDF is used. The system's time zone rule is applied (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Return Values

0 Indicates that the routine executed successfully.

−1 Generic error that indicates the time service cannot be accessed.

Examples

See the sample program in the **utc_boundtime(3dts)** reference page.

Related Information

Functions: **utc_gettime(3dts)**.

utc_gmtime

Purpose

Converts a binary timestamp to a tm structure that expresses GMT or the equivalent UTC

Synopsis

```
#include <dce/utc.h>

int  utc_gmtime(
   struct tm *timetm
   long *tns
   struct tm *inacctm
   long *ins
   utc_t *utc);
```

Parameters

Input

utc Binary timestamp to be converted to **tm** structure components. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp.

tns Nanoseconds since the time component of the binary timestamp.

inacctm
Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp. If the inaccuracy is unspecified, *ins* returns a value of −1.

Description

The **utc_gmtime()** routine converts a binary timestamp to a **tm** structure that expresses GMT (or the equivalent UTC). Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpintervaltime(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts),utc_gmtzone(3dts)**, **utc_localtime(3dts)**, **utc_mkgmtime(3dts)**.

utc_gmtzone

Purpose

Gets the time zone label for GMT

Synopsis

```
#include <dce/utc.h>

int  utc_gmtzone(
   char *tzname
   size_t tzlen
   long *tdf
   int *isdst
   utc_t *utc);
```

Parameters

Input

tzlen Length of buffer *tzname*.

utc Binary timestamp. This parameter is ignored.

Output

tzname
Character string long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT. A value of 0 (zero) is always returned.

isdst Integer with a value of 0 (zero), indicating that daylight saving time is not in effect. A value of 0 (zero) is always returned.

Description

The **utc_gmtzone()** routine gets the time zone label and zero offset from GMT. Outputs are always *tdf*=**0** and *tzname*=**GMT**. This routine exists for symmetry with the **utc_anyzone()** and the **utc_localzone()** routines. Use NULL if you want this routine to use the current time for this parameter.

Notes

All of the output parameters are optional. No value is returned and no error occurs if the *tzname* pointer is NULL.

Return Values

0 Indicates that the routine executed successfully (always returned).

Examples

The following example prints out the current time in both local time and GMT time.

```
utc_t    now;
struct tm  tmlocal, tmgmt;
```

```
long    tzoffset;
int     tzdaylight;
char    tzlocal[80], tzgmt[80];

/* Get the current time once, so both conversions use the same
 *  time...
 */
utc_gettime(&now);

/* Convert to local time, using the process TZ environment
 *  variable...
 */
utc_localtime(&tmlocal,    /* Out: Local time tm structure    */
       (long *)0,   /* Out: Nanosec of time        */
       (struct tm *)0, /* Out: Inaccuracy tm structure    */
       (long *)0,   /* Out: Nanosec of inaccuracy     */
       (int *)0,    /* Out: TDF of local time       */
       &now);     /* In: Current timestamp (ignore)  */



/*  Get the local time zone name, offset from GMT, and current
 *  daylight savings flag...
 */
utc_localzone(tzlocal,    /* Out: Local time zone name     */
       80,       /* In: Length of loc time zone name */
       &tzoffset,   /* Out: Loc time zone offset in secs */
       &tzdaylight,  /* Out: Local time zone daylight flag */
       &now);     /* In: Current binary timestamp   */


/*  Convert to GMT...
 */
utc_gmtime(&tmgmt,      /* Out: GMT tm structure       */
      (long *)0,     /* Out: Nanoseconds of time      */
      (struct tm *)0,  /* Out: Inaccuracy tm structure    */
      (long *)0,     /* Out: Nanoseconds of inaccuracy   */
      &now);       /* In: Current binary timestamp   */

/*  Get the GMT time zone name...
 */
utc_gmtzone(tzgmt,      /* Out: GMT time zone name      */
      80,        /* In: Size of GMT time zone name  */
      (long *)0,    /* Out: GMT time zone offset in secs */
      (int *)0,     /* Out: GMT time zone daylight flag  */
      &now);      /* In: Current binary timestamp   */
   /*   (ignore)    */

/*  Print out times and time zone information in the following
 *  format:
 *
 *     12:00:37 (EDT) = 16:00:37 (GMT)
 *     EDT is -240 minutes ahead of Greenwich Mean Time.
 *     Daylight savings time is in effect.
 */
printf("%d:%02d:%02d (%s) = %d:%02d:%02d (%s)\n",
    tmlocal.tm_hour, tmlocal.tm_min, tmlocal.tm_sec, tzlocal,
    tmgmt.tm_hour, tmgmt.tm_min, tmgmt.tm_sec, tzgmt);
printf("%s is %d minutes ahead of Greenwich Mean Time\n", tzlocal,
    tzoffset/60);
if (tzdaylight != 0)
  printf("Daylight savings time is in effect\n");
```

Related Information

Functions: **utc_anyzone(3dts)**, **utc_gmtime(3dts)**, **utc_localzone(3dts)**.

utc_localtime

Purpose

Converts a binary timestamp to a tm structure that expresses local time

Synopsis

```
#include <dce/utc.h>

int  utc_localtime(
   struct tm *timetm
   long *tns
   struct tm *inacctm
   long *ins
   utc_t *utc);
```

Parameters

Input

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

timetm Time component of the binary timestamp, expressing local time.

tns Nanoseconds since the time component of the binary timestamp.

inacctm
Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the binary timestamp. If the inaccuracy is unspecified, *ins* returns a value of −1.

Description

The **utc_localtime()** routine converts a binary timestamp to a **tm** structure that expresses local time.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime** () system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_gmtzone(3dts)** reference page.

Related Information

Functions: **utc_anytime(3dts)**, **utc_gmtime(3dts)**, **utc_localzone(3dts)**, **utc_mklocaltime(3dts)**.

utc_localzone

Purpose

Gets the local time zone label and offset from GMT, given utc

Synopsis

```
#include <dce/utc.h>

int  utc_localzone(
   char *tzname
   size_t tzlen
   long *tdf
   int *isdst
   utc_t *utc);
```

Parameters

Input

tzlen Length of the *tzname* buffer.

utc Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

tzname
Character string long enough to hold the time zone label.

tdf Long word with differential in seconds east of GMT.

isdst Integer with a value of 0 (zero) if standard time is in effect or a value of 1 if daylight saving time is in effect.

Description

The **utc_localzone()** routine gets the local time zone label and offset from GMT, given **utc**.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime** () system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Notes

All of the output parameters are optional. No value is returned and no error occurs if the pointer is NULL.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or an insufficient buffer.

Examples

See the sample program in the **utc_gmtzone(3dts)** reference page.

Related Information

Functions: **utc_anyzone(3dts)**, **utc_gmtzone(3dts)**, **utc_localtime(3dts)**.

utc_mkanytime

Purpose

Converts a tm structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkanytime(
   utc_t *utc
   struct tm *timetm
   long tns
   struct tm *inacctm
   long ins
   long tdf);
```

Parameters

Input

timetm A **tm** structure that expresses the local time; **tm_wday** and **tm_yday** are ignored on input; the value of **tm_isdt** should be −1.

tns Nanoseconds since the time component.

inacctm
A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified; **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored on input.

ins Nanoseconds of the inaccuracy component.

tdf Time differential factor to use in conversion.

Output

utc Resulting binary timestamp.

Description

The **utc_mkanytime()** routine converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp. Required inputs include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

The following example converts a string ISO format time in an arbitrary time zone to a binary timestamp. This may be part of an input timestamp routine, although a real implementation will include range checking.

```
utc_t    utc;
struct tm  tmtime, tminacc;
float    tsec, isec;
double   tmp;
long    tnsec, insec;
int     i, offset, tzhour, tzmin, year, mon;
char    *string;

/* Try to convert the string...                */

if(sscanf(string, "%d-%d-%d-%d:%d:%e+%d:%dI%e",
     &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
     &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {

/* Try again with a negative TDF...              */

if (sscanf(string, "%d-%d-%d-%d:%d:%e-%d:%dI%e",
      &year, &mon, &tmtime.tm_mday, &tmtime.tm_hour,
      &tmtime.tm_min, &tsec, &tzhour, &tzmin, &isec) != 9) {

/* ERROR                           */

    exit(1);
  }


/* TDF is negative                         */

  tzhour = -tzhour;
  tzmin = -tzmin;

}

/* Fill in the fields...                     */

tmtime.tm_year = year - 1900;
tmtime.tm_mon = --mon;
tmtime.tm_sec = tsec;
tnsec = (modf(tsec, &tmp)*1.0E9);
offset = tzhour*3600 + tzmin*60;
tminacc.tm_sec = isec;
insec = (modf(isec, &tmp)*1.0E9);

/* Convert to a binary timestamp...                */

utc_mkanytime(&utc,    /* Out: Resultant binary timestamp    */
       &tmtime,  /* In: tm struct that represents input */
       tnsec,   /* In: Nanoseconds from input      */
       &tminacc,  /* In: tm struct that represents inacc */
       insec,   /* In: Nanoseconds from input      */
       offset);  /* In: TDF from input          */
```

Related Information

Functions: **utc_anytime(3dts)**, **utc_anyzone(3dts)**.

utc_mkascreltime

Purpose

Converts a NULL-terminated character string that represents a relative timestamp to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkascreltime(
   utc_t *utc
   char *string);
```

Parameters

Input

string A NULL-terminated string that expresses a relative timestamp in its ISO format.

Output

utc Resulting binary timestamp.

Description

The **utc_mkascreltime()** routine converts a NULL-terminated string, which represents a relative timestamp, to a binary timestamp.

Notes

The ASCII string must be NULL-terminated.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts an ASCII relative time string to its binary equivalent.

```
utc_t   utc;
char    str[UTC_MAX_STR_LEN];

/*  Relative time of -333 days, 12 hours, 1 minute, 37.223 seconds
 *  Inaccuracy of 50.22 seconds in the format: -333-12:01:37.223I50.22
 */
(void)strcpy((void *)str,
       "-333-12:01:37.223I50.22");

utc_mkascreltime(&utc,  /* Out: Binary utc           */
         str);  /* In: String             */
```

Related Information

Functions: **utc_ascreltime(3dts)**.

utc_mkasctime

Purpose

Converts a NULL-terminated character string that represents an absolute timestamp to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkasctime(
   utc_t *utc
   char *string);
```

Parameters

Input

string A NULL-terminated string that expresses an absolute time.

Output

utc Resulting binary timestamp.

Description

The **utc_mkasctime()** routine converts a NULL-terminated string that represents an absolute time to a binary timestamp.

Notes

The ASCII string must be NULL-terminated.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time parameter or invalid results.

Examples

The following example converts an ASCII time string to its binary equivalent.

```
utc_t   utc;
char   str[UTC_MAX_STR_LEN];

/*  July 4, 1776, 12:01:37.223 local time
 *  TDF of -5:00 hours
 *  Inaccuracy of 3600.32 seconds
 */
(void)strcpy((void *)str,
       "1776-07-04-12:01:37.223-5:00I3600.32");

utc_mkasctime(&utc,  /* Out: Binary utc      */
       str);  /* In: String        */
```

Related Information

Functions: **utc_ascanytime(3dts)**, **utc_ascgmtime(3dts)**, **utc_asclocaltime(3dts)**.

utc_mkbinreltime

Purpose

Converts a timespec structure expressing a relative time to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkbinreltime(
   utc_t *utc
   reltimespec_t *timesp
   timespec_t *inaccsp);
```

Parameters

Input

timesp A **reltimespec** structure that expresses a relative time.

inaccsp
A **timespec** structure that expresses inaccuracy. If a null pointer is passed, or if **tv_sec** is set to a value of −1, the inaccuracy is considered to be unspecified.

Output

utc Resulting relative binary timestamp.

Description

The **utc_mkbinreltime()** routine converts a **timespec** structure that expresses relative time to a binary timestamp.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_addtime(3dts)** reference page.

Related Information

Functions: **utc_binreltime(3dts)**, **utc_mkbintime(3dts)**.

utc_mkbintime

Purpose

Converts a timespec structure to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkbintime(
   utc_t *utc
   timespec_t *timesp
   timespec_t *inaccsp
   long tdf);
```

Parameters

Input

timesp A **timespec** structure that expresses time since 1970−01−01:00:00:00.0+00:00I0.

inaccsp
A **timespec** structure that expresses inaccuracy. If a null pointer is passed, or if **tv_sec** is set to a value of −1, the inaccuracy is considered to be unspecified.

tdf TDF component of the binary timestamp.

Output

utc Resulting binary timestamp.

Description

The **utc_mkbintime()** routine converts a **timespec** structure time to a binary timestamp. The TDF input is used as the TDF of the binary timestamp.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

The following example obtains the current time from **time(3)**, converts it to a binary timestamp with an inaccuracy of 5.2 seconds, and specifies GMT.

```
timespec_t  ttime, tinacc;
utc_t    utc;

/*  Obtain the current time (without the inaccuracy)...
 */

ttime.tv_sec = time((time_t *)0);
ttime.tv_nsec = 0;
```

```
/*  Specify the inaccuracy...
 */

tinacc.tv_sec = 5;
tinacc.tv_nsec = 200000000;

/*  Convert to a binary timestamp...
 */

utc_mkbintime(&utc,    /* Out: Binary timestamp     */
       &ttime,   /* In: Current time in timespec */
       &tinacc,  /* In: 5.2 seconds in timespec  */
       0);     /* In: TDF of GMT        */
```

Related Information

Functions: **utc_bintime(3dts)**, **utc_mkbinreltime(3dts)**.

utc_mkgmtime

Purpose

Converts a tm structure that expresses GMT or UTC to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkgmtime(
   utc_t *utc
   struct tm *timetm
   long tns
   struct tm *inacctm
   long ins);
```

Parameters

Input

timetm A **tm** structure that expresses GMT. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdt** should be −1.

tns Nanoseconds since the time component.

inacctm
A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored.

ins Nanoseconds of the inaccuracy component.

Output

utc Resulting binary timestamp.

Description

The **utc_mkgmtime()** routine converts a **tm** structure that expresses GMT or UTC to a binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpintervaltime(3dts)** reference page.

Related Information

Functions: **utc_gmtime(3dts)**.

utc_mklocaltime

Purpose

Converts a tm structure that expresses local time to a binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mklocaltime(
   utc_t *utc
   struct tm *timetm
   long tns
   struct tm *inacctm
   long ins);
```

Parameters

Input

timetm A **tm** structure that expresses the local time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** should be −1.

tns Nanoseconds since the time component.

inacctm
A **tm** structure that expresses days, hours, minutes, and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday**, and **tm_isdst** are ignored.

ins Nanoseconds of the inaccuracy component.

Output

utc Resulting binary timestamp.

Description

The **utc_mklocaltime()** routine converts a **tm** structure that expresses local time to a binary timestamp.

The user's environment determines the time zone rule (details are system dependent). For example, on OSF/1 systems, the user selects a time zone by specifying the **TZ** environment variable. (The reference information for the **localtime** () system call, which is described in the **ctime(3)** reference page, provides additional information.)

If the user's environment does not specify a time zone rule, the system's rule is used (details of the rule are system dependent). For example, on OSF/1 systems, the rule in **/etc/zoneinfo/localtime** applies.

Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_cmpmidtime(3dts)** reference page.

Related Information

Functions: **utc_localtime(3dts)**.

utc_mkreltime

Purpose

Converts a tm structure that expresses relative time to a relative binary timestamp

Synopsis

```
#include <dce/utc.h>

int  utc_mkreltime(
   utc_t *utc
   struct tm *timetm
   long tns
   struct tm *inacctm
   long ins);
```

Parameters

Input

timetm A **tm** structure that expresses a relative time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** should be −1.

tns Nanoseconds since the time component.

inacctm
A **tm** structure that expresses seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_year**, **tm_wday**, **tm_isdst**, and **tm_zone** are ignored.

ins Nanoseconds of the inaccuracy component.

Output

utc Resulting relative binary timestamp.

Description

The **utc_mkreltime()** routine converts a **tm** structure that expresses relative time to a relative binary timestamp. Additional inputs include nanoseconds since the last second of time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

The following example converts the relative time **125-03:12:30.1I120.25** to a relative binary timestamp.

```
utc_t    utc;
struct tm  tmtime,tminacc;
long    tnsec,insec;
```

```
  /* Fill in the fields
   */
  memset((void *)&tmtime,0,sizeof(tmtime));
  tmtime.tm_mday = 125;
  tmtime.tm_hour = 3;
  tmtime.tm_min = 12;
  tmtime.tm_sec = 30;
  tnsec = 100000000;   /* .1 * 1.0E9 */

  memset((void *)&tminacc,0,sizeof(tminacc));
  tminacc.tm_sec = 120;
  tnsec = 250000000;   /* .25 * 1.0E9 */

/* Convert to a relative binary timestamp...
 */
utc_mkreltime(&utc,    /* Out: Resultant relative binary timestamp */
       &tmtime,  /* In: tm struct that represents input   */
       tnsec,   /* In: Nanoseconds from input       */
       &tminacc,  /* In: tm struct that represents inacc   */
       insec);   /* In: Nanoseconds from input
*/
```

utc_mulftime

Purpose

Multiplies a relative binary timestamp by a floating-point value

Synopsis

```
#include <dce/utc.h>

int  utc_mulftime(
   utc_t *result
   utc_t *utc1
   double factor);
```

Parameters

Input

utc1 Relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

factor Real scale factor (double-precision, floating-point value).

Output

result Resulting relative binary timestamp.

Description

The **utc_mulftime()** routine multiplies a relative binary timestamp by a floating-point value. Either or both may be negative; the resulting relative binary timestamp has the appropriate sign. The unsigned inaccuracy in the relative binary timestamp is also multiplied by the absolute value of the floating-point value.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

Examples

The following example scales a relative time by a floating-point factor and prints the result.

```
utc_t    relutc, scaledutc;
struct tm  scaledreltm;
char    timstr[UTC_MAX_STR_LEN];

/*  Assume relutc contains the time to scale.
 */
utc_mulftime(&scaledutc,      /* Out: Scaled rel time   */
       &relutc,       /* In: Rel time to scale  */
       17.65);          /* In: Scale factor    */

utc_ascreltime(timstr,         /* Out: ASCII rel time   */
        UTC_MAX_STR_LEN,    /* In: Input buffer length */
        &scaledutc);    /* In: Rel time to convert */
```

```
printf("%s\n",timstr);

/*  Convert it to a tm structure and print it.
 */
utc_reltime(&scaledreltm,     /* Out: Scaled rel tm    */
      (long *)0,         /* Out: Scaled rel nano-sec */
      (struct tm *)0,      /* Out: Scaled rel inacc tm */
      (long *)0,         /* Out: Scd rel inacc nanos */
      &scaledutc);      /* In: Rel time to convert */

printf("Approximately %d days, %d hours and %d minutes\n",
    scaledreltm.tm_yday, scaledreltm.tm_hour, scaledreltm.tm_min);
```

Related Information

Functions: **utc_multime(3dts)**.

utc_multime

Purpose

Multiplies a relative binary timestamp by an integer factor

Synopsis

```
#include <dce/utc.h>

int  utc_multime(
   utc_t *result
   utc_t *utc1
   long factor);
```

Parameters

Input

utc1 Relative binary timestamp.

factor Integer scale factor.

Output

result Resulting relative binary timestamp.

Description

The **utc_multime()** routine multiplies a relative binary timestamp by an integer. Either or both may be negative; the resulting binary timestamp has the appropriate sign. The unsigned inaccuracy in the binary timestamp is also multiplied by the absolute value of the integer.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

Examples

The following example scales a relative time by an integral value and prints the result.

```
utc_t    relutc, scaledutc;

char    timstr[UTC_MAX_STR_LEN];

/*  Assume relutc contains the time to scale.
 *  Scale it by a factor of 17 ...
 */
utc_multime(&scaledutc,   /* Out: Scaled rel time    */
    &relutc,   /* In: Rel time to scale   */
      17L);    /* In: Scale factor     */

utc_ascreltime(timstr,     /* Out: ASCII rel time    */
```

```
   UTC_MAX_STR_LEN,     /* In: Input buffer length  */
      &scakedutc);   /* In: Rel time to convert  */

printf("Scaled result is %s, timstr);
```

Related Information

Functions: **utc_mulftime(3dts)**.

utc_pointtime

Purpose

Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time

Synopsis

```
#include <dce/utc.h>

int  utc_pointtime(
   utc_t *utclp
   utc_t *utcmp
   utc_t *utchp
   utc_t *utc);
```

Parameters

Input

utc Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

utclp Lowest (earliest) possible absolute time or shortest possible relative time that the input timestamp can represent.

utcmp Midpoint of the input timestamp.

utchp Highest (latest) possible absolute time or longest possible relative time that the input timestamp can represent.

Description

The **utc_pointtime()** routine converts a binary timestamp to three binary timestamps that represent the earliest, latest, and most likely (midpoint) times. If the input is a relative binary time, the outputs represent relative binary times.

Notes

All outputs have zero inaccuracy. An error is returned if the input binary timestamp has an unspecified inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument.

Examples

See the sample program in the **utc_addtime(3dts)** reference page.

Related Information

Functions: **utc_boundtime(3dts)**, **utc_spantime(3dts)**.

utc_reltime

Purpose

Converts a relative binary timestamp to a tm structure

Synopsis

```
#include <dce/utc.h>

int  utc_reltime(
   struct tm *timetm
   long *tns
   struct tm *inacctm
   long *ins
   utc_t *utc);
```

Parameters

Input

utc Relative binary timestamp.

Output

timetm Relative time component of the relative binary timestamp. The field **tm_mday** returns a value of −1 and the fields **tm_year** and **tm_mon** return values of 0 (zero). The field **tm_yday** contains the number of days of relative time.

tns Nanoseconds since the time component of the relative binary timestamp.

inacctm
Seconds of the inaccuracy component of the relative binary timestamp. If the inaccuracy is finite, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Nanoseconds of the inaccuracy component of the relative binary timestamp.

Description

The **utc_reltime()** routine converts a relative binary timestamp to a **tm** structure. Additional returns include nanoseconds since time and nanoseconds of inaccuracy.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_mulftime(3dts)** reference page.

Related Information

Functions: **utc_mkreltime(3dts)**.

utc_spantime

Purpose

Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps

Synopsis

```
#include <dce/utc.h>

int  utc_spantime(
   utc_t *result
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Spanning timestamp.

Description

Given two binary timestamps, the **utc_spantime()** routine returns a single UTC time interval whose inaccuracy spans the two input timestamps (that is, the interval resulting from the earliest possible time of either timestamp to the latest possible time of either timestamp).

Notes

The *tdf* parameter in the output UTC value is copied from the *utc2* input. If either input binary timestamp has an unspecified inaccuracy, an error is returned.

Return Values

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument.

Examples

The following example computes the earliest and latest times for an array of 10 timestamps.

```
utc_t        time_array[10], testtime, earliest, latest;
int         i;

/*  Set the running timestamp to the first entry...
```

```
 */
testtime = time_array[0];

for (i=1; i<10; i++) {

  /*  Compute the minimum and the maximum against the next
  *  element...
  */
utc_spantime(&testtime,    /* Out: Resultant interval     */
       &testtime,    /* In: Largest previous interval  */
       &time_array[i]); /* In: Element under test     */
}

/*  Compute the earliest and latest possible times
 */

utc_pointtime(&earliest,    /* Out: Earliest poss time in array */
       (utc_t *)0,      /* Out: Midpoint          */
       &latest,     /* Out: Latest poss time in array  */
       &testtime);    /* In: Spanning interval
*/
```

Related Information

Functions: **utc_boundtime(3dts)**, **utc_gettime(3dts)**, **utc_pointtime(3dts)**.

utc_subtime

Purpose

Computes the difference between two binary timestamps

Synopsis

```
#include <dce/utc.h>

int  utc_subtime(
   utc_t *result
   utc_t *utc1
   utc_t *utc2);
```

Parameters

Input

utc1 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

utc2 Binary timestamp or relative binary timestamp. Use NULL if you want this routine to use the current time for this parameter.

Output

result Resulting binary timestamp or relative binary timestamp, depending upon the operation performed:

- *absolute time* − *absolute time* = *relative time*
- *relative time* − *relative time* = *relative time*
- *absolute time* − *relative time* = *absolute time*
- *relative time* − *absolute time* is undefined. (See the note later in this reference page.)

Description

The **utc_subtime()** routine subtracts one binary timestamp from another. The two binary timestamps express either an absolute time and a relative time, two relative times, or two absolute times. The resulting timestamp is *utc1* minus *utc2*. The inaccuracies of the two input timestamps are combined and included in the output timestamp. The TDF in the first timestamp is copied to the output.

Notes

Although no error is returned, the combination *relative time*−*absolute time* should *not* be used.

Return Values

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

Examples

See the sample program in the **utc_binreltime(3dts)** reference page.

Related Information

Functions: **utc_addtime(3dts)**.

Chapter 6. DCE Security Service

sec_intro

Purpose

Application program interface to the DCE Security Service

Description

The DCE Security Service application program interface (API) allows developers to create network services with complete access to all the authentication and authorization capabilities of DCE Security Service and facilities.

The transaction of a network service generally consists of a client process requesting some action from a server process. The client may itself be a server, or a user, and the server may also be a client of other servers. Before the targeted server executes the specified action, it must be sure of the client's identity, and it must know whether the client is authorized to request the service.

The security service API consists of the following sets of remote procedure calls (RPCs) used to communicate with various security-related services and facilities:

rgy Maintains the network registry of principal identities.

era Maintains extended registry attributes.

login Validates a principal's network identity and establish delegated identities.

epa Extracts privilege attributes from an opaque binding handle.

acl Implements an access control list (ACL) protocol for the authorization of a principal to network access and services.

key Provides facilities for the maintenance of account keys for daemon principals.

id Maps file system names to universal unique IDs (UUIDs).

pwd_mgmt
Provides facilities for password management.

pk Provides facilities for public key authentication.

All the calls in this API have names beginning with the **sec_** prefix. These are the same calls used by various user-level tools provided as part of the DCE. For example, the **sec_create_db(1)** tool is written with **sec_rgy** calls, **acl_edit(1)** is written with **sec_acl** calls, and the **login(1)** program, with which a user logs in to a DCE system, is written using **sec_login** calls. Most sites will find the user-level tools adequate for their needs, and only must use the security service API to customize or replace the functionality of these tools.

Though most of the calls in the security service API represent RPC transactions, code has been provided on the client side to handle much of the overhead involved with making remote calls. These *stubs* handle binding to the requested security server site, the marshalling of data into whatever form is needed for transmission, and other bookkeeping involved with these remote calls. An application programmer can use the security service interfaces as if they were composed of simple C functions.

This reference page introduces each of the following APIs:

- Registry APIs
- Login APIs
- Extended privilege attributes APIs
- Extended registry attributes APIs
- ACL APIs
- Key management APIs
- ID mapping APIs
- Password management APIs
- Public Key APIs

The section for each API is organized as follows:

- Synopsis
- Data Types
- Constants
- Files

Registry API Data Types

Synopsis

```
#include  <dce/rgybase.h>
```

Data Types

The following data types are used in **sec_rgy_** * calls:

sec_rgy_handle_t
A pointer to the registry server handle. The registry server is bound to a handle with the **sec_rgy_site_open()** routine.

sec_rgy_bind_auth_info_type_t
A enumeration that defines whether or not the binding is authenticated. This data type is used in conjunction with the **sec_rgy_bind_auth_info_t** data type to set up the authorization method and parameters for a binding. The **sec_rgy_bind_auth_info_type_t** type consists of the following elements:

sec_rgy_bind_auth_none
The binding is not authenticated.

sec_rgy_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_rgy_bind_auth_info_t
A discriminated union that defines authorization and authentication parameters for a binding. This data type is used in conjunction with the **sec_rgy_bind_auth_info_type_t** data type to set up the authorization method and parameters for a binding. The **sec_rgy_bind_auth_info_t** data type consists of the following elements:

info_type
A **sec_rgy_bind_auth_info_type_t** data type that specifies whether or not the binding is authenticated. The contents of the union depend on the value of **sec_rgy_bind_auth_info_type_t**.

For unauthenticated bindings (**sec_rgy_bind_auth_info_type_t** = **sec_rgy_bind_auth_none**), no parameters are supplied.

For authenticated bindings (**sec_rgy_bind_auth_info_type_t** = **sec_rgy_bind_auth_dce**), the **dce_info** structure is supplied.

dce_info
A structure that consists of the following elements:

authn_level
An unsigned 32-bit integer indicating the protection level for RPC calls made using the server binding handle. The protection level determines the degree to which authenticated communications between the client and the server are protected by the authentication service specified by **authn_svc**.

If the RPC runtime or the RPC protocol in the bound protocol sequence does not support a specified level, the level is automatically upgraded to the next higher supported level. The possible protection levels are as follows:

Protection Level	Description
rpc_c_protect_level_default	Uses the default protection level for the specified authentication service. The default protection level for DCE shared-secret key authentication is **rpc_c_protect_level_pkt_value**.
rpc_c_protect_level_none	Performs no authentication: tickets are not exchanged, session keys are not established, client PACs or names are not certified, and transmissions are in the clear. Note that although uncertified PACs should not be trusted, they may be useful for debugging, tracing, and measurement purposes.
rpc_c_protect_level_connect	Authenticates only when the client establishes a relationship with the server.
rpc_c_protect_level_call	Authenticates only at the beginning of each remote procedure call when the server receives the request. This level does not apply to remote procedure calls made over a connection-based protocol sequence (that is, **ncacn_ip_tcp**). If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.
rpc_c_protect_level_pkt	Ensures that all data received is from the expected client.

Protection Level	Description
rpc_c_protect_level_pkt_integ	Ensures and verifies that none of the data transferred between client and server has been modified. This is the highest protection level that is guaranteed to be present in the RPC runtime.
rpc_c_protect_level_pkt_privacy	Authenticates as specified by all of the previous levels and also encrypts each RPC argument value. This is the highest protection level, but is not guaranteed to be present in the RPC runtime.

authn_svc

Specifies the authentication service to use. The exact level of protection provided by the authentication service is specified by *protect_level*. The supported authentication services are as follows:

Authentication Service	Description
rpc_c_authn_none	No authentication: no tickets are exchanged, no session keys established, client PACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for remote procedure calls made using this binding.
rpc_c_authn_dce_secret	DCE shared-secret key authentication.
rpc_c_authn_default	Default authentication service. The current default authentication service is DCE shared-secret key; therefore, specifying **rpc_c_authn_default** is equivalent to specifying **rpc_c_authn_dce_secret**.
rpc_c_authn_dce_public	DCE public key authentication (reserved for future use).

authz_svc
Specifies the authorization service implemented by the server for the interface. The validity and trustworthiness of authorization data, like any application data, is dependent on the authentication service and protection level specified. The supported authorization services are as follows:

Authentication Service	Description
rpc_c_authz_none	Server performs no authorization. This is valid only if **authn_svc** is set to **rpc_c_authn_none**, specifying that no authentication is being performed.
rpc_c_authz_name	Server performs authorization based on the client principal name. This value cannot be used if **authn_svc** is **rpc_c_authn_none**.
rpc_c_authz_dce	Server performs authorization using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with this binding. Generally, access is checked against DCE access control lists (ACLs).

identity
A value of type **sec_login_handle_t** that represents a complete login context.

sec_timeval_sec_t
A 32-bit integer containing the seconds portion of a UNIX **timeval_t**, to be used when expressing absolute dates.

sec_timeval_t
A structure containing the full UNIX time. The structure contains two 32-bit integers that indicate seconds (**sec**) and microseconds (**usec**) since 0:00, January 1, 1970.

sec_timeval_period_t
A 32-bit integer expressing seconds relative to some well-known time.

sec_rgy_acct_key_t
Specifies how many parts (person, group, organization) of an account login name will be enough to specify a unique abbreviation for that account.

sec_rgy_cursor_t
A structure providing a pointer into a registry database. This type is used for iterative operations on the registry information. For example, a call to **sec_rgy_pgo_get_members()** might return the 10 account names following the input **sec_rgy_cursor_t** position. Upon return, the cursor position will have been updated, so the next call to that routine will return the next 10 names. The components of this structure are not used by application programs.

sec_rgy_pname_t
A character string of length **sec_rgy_pname_t_size**.

sec_rgy_name_t
A character string of length **sec_rgy_name_t_size**.

sec_rgy_login_name_t
A structure representing an account login name. It contains three strings of type **sec_rgy_name_t**:

pname
The person name for the account.

gname
The group name for the account.

oname
The organization name for the account.

sec_rgy_member_t
A character string of length **sec_rgy_name_t_size**.

sec_rgy_foreign_id_t
The representation of a foreign ID. This structure contains two components:

cell A string of type **uuid_t** representing the UUID of the foreign cell.

principal
A string of type **uuid_t** representing the UUID of the principal.

sec_rgy_sid_t
A structure identifying an account. It contains three fields:

person
The UUID of the person part of the account.

group The UUID of the group part of the account.

org The UUID of the organization part of the account.

sec_rgy_unix_sid_t
A structure identifying an account with UNIX ID numbers. It contains three fields:

person
The UNIX ID of the person part of the account.

group The UNIX ID of the group part of the account.

org The UNIX ID of the organization part of the account.

sec_rgy_domain_t
This 32-bit integer specifies which naming domain a character string refers to: person, group, or organization.

sec_rgy_pgo_flags_t
A 32-bit bitset containing flags pertaining to registry entries. This type contains the following three flags:

sec_rgy_pgo_is_an_alias
If set, indicates the registry entry is an alias of another entry.

sec_rgy_pgo_is_required
If set, the registry item is required and cannot be deleted. An example of a required account is the one for the registry server itself.

sec_rgy_pgo_projlist_ok
If the accompanying item is a person entry, this flag indicates the person may have concurrent group sets. If the item is a group entry, the flag means this group can appear in a concurrent group set. The flag is undefined for organization items.

sec_rgy_pgo_item_t
The structure identifying a registry item. It contains five components:

id The UUID of the registry item, in **uuid_t** form.

unix_num
A 32-bit integer containing the UNIX ID number of the registry item.

quota A 32-bit integer representing the maximum number of user-defined groups the account owner can create.

flags A **sec_rgy_pgo_flags_t** bitset containing information about the entry.

fullname
A **sec_rgy_pname_t** character string containing a full name for the registry entry. For a person entry, this field might contain the real name of the account owner. For a group, it might contain a description of the group. This is just a data field, and registry queries cannot search on the **fullname** entry.

sec_rgy_acct_admin_flags_t
A 32-bit bitset containing administration flags used as part of the administrator's information for any registry account. The set contains three flags:

sec_rgy_acct_admin_valid
Specifies that the account is valid for login.

sec_rgy_acct_admin_server
If set, the account's name can be used as a server name in a ticket-granting ticket.

sec_rgy_acct_admin_client
If set, the account's name can be used as a client name in a ticket-granting ticket.

Note that you can prevent the principal from being authenticated, by turning off both the **sec_rgy_acct_admin_server** and the **sec_rgy_acct_admin_client flags**.

sec_rgy_acct_auth_flags_t
A 32-bit bitset containing account authorization flags used to implement authentication policy as defined by the Kerberos Version 5 protocol. The set contains the following flags:

sec_rgy_acct_auth_user_to_user
Forces the use of user-to-user server authentication on a server principal.

sec_rgy_acct_auth_post_dated
Allows issuance of post-dated certificates.

sec_rgy_acct_auth_forwardable
Allows issuance of forwardable certificates.

sec_rgy_acct_auth_tgt
Allows issuance of certificates based on ticket-granting ticket (TGT) authentication. If this flag is not set, a client requesting a service may have to supply a password directly to the server.

sec_rgy_acct_auth_renewable
Allows issuance of renewable certificates.

sec_rgy_acct_auth_proxiable
Allows issuance of proxiable certificates.

sec_rgy_acct_auth_dup_session_key
Allows issuance of duplicate session keys.

sec_rgy_acct_admin_t
The portion of a registry account item containing components relevant to administrators. This structure consists of the fields listed below. Note that only *expiration_date*, *good_since_date*, *flags*, and *authentication_flags* can be modified by an administrator; the remaining fields are set by the security server.

creator
This field, in **foreign_id_t format**, identifies the administrator who created the registry account.

creation_date
Specifies the creation date of the account, in **sec_timeval_sec_t** format.

last_changer
Identifies the last person to change any of the account information, in **foreign_id_t** format.

change_date
Specifies the date of the last modification of the account information, in **sec_timeval_sec_t** format.

expiration_date
The date after which the account will no longer be valid. In **sec_timeval_sec_t** format.

good_since_date
The Kerberos Version 5 TGT revocation date. TGTs issued before this date will not be honored. In **sec_timeval_sec_t** format.

flags Administrative flags in **sec_rgy_acct_admin_flags_t** format.

authentication_flags
Authentication flags in **sec_rgy_acct_auth_flags_t** format.

sec_rgy_acct_user_flags_t
A 32-bit bitset containing flags controlling user-modifiable information. There is only one flag currently implemented. If **sec_rgy_acct_user_passwd_valid** is set, it indicates the user password is valid. If it is not set, this flag prompts the user to change the password on the next login attempt.

sec_rgy_acct_user_t
A structure containing registry account information. The structure consists of the fields listed below. Note that only the **gecos**, **homedir**, **shell**, and **flags** fields can be modified by the account owner or other authorized user; the remaining fields are set by the security server.

gecos This is a character string (in **sec_rgy_pname_t** format) containing information about the account user. It generally consists of everything after the full name in the UNIX **gecos** format.

homedir
The login directory for the account user, in **sec_rgy_pname_t** format.

shell The default shell for the account user, in **sec_rgy_pname_t** format.

passwd_version_number
An unsigned 32-bit integer, indicating the password version number. This value is used as output only.

passwd
The UNIX encrypted account password, in **sec_rgy_unix_passwd_buf_t** format. This value is used as output only.

passwd_dtm
The date the password was established, in **sec_timeval_sec_t** format.

flags Account user flags, in **sec_rgy_acct_user_flags_t** format.

sec_rgy_plcy_pwd_flags_t
A 32-bit bitset containing two flags about password policy:

sec_rgy_plcy_pwd_no_spaces
If set, will not allow spaces in a password.

sec_rgy_plcy_pwd_non_alpha
If set, requires at least one nonalphanumeric character in the password.

sec_rgy_plcy_t
A structure defining aspects of registry account policy. It contains five components:

passwd_min_len
A 32-bit integer describing the minimum number of characters in the account password.

passwd_lifetime
The number of seconds after a password's creation until it expires, in **sec_timeval_period_t** format.

passwd_exp_date
The expiration date of the account password, in **sec_timeval_sec_t** format.

acct_lifespan
The number of seconds after the creation of an account before it expires, in **sec_timeval_period_t** format.

passwd_flags
Account password policy flags, in **sec_rgy_plcy_pwd_flags_t** format.

sec_rgy_plcy_auth_t
This type describes authentication policy. It is a structure containing two time periods, in **sec_timeval_period_t** format. One, **max_ticket_lifetime**, specifies the maximum length of the period during which a ticket-granting ticket (TGT) will be valid. The other, **max_renewable_lifetime**, specifies the maximum length of time for which such a ticket may be renewed. This authentication policy applies both to the registry as a whole as well as individual accounts. The effective policy for a given account is defined to be the more restrictive of the site and principal authentication policy.

sec_rgy_properties_t
A structure describing some registry properties. It contains the following:

read_version
A 32-bit integer describing the earliest version of the **secd** software that can read this registry.

write_version
A 32-bit integer describing the version of the **secd** software that wrote this registry.

minimum_ticket_lifetime
The minimum lifetime of an authentication certificate, in **sec_timeval_period_t** format.

default_certificate_lifetime
The normal lifetime of an an authentication certificate (ticket-granting ticket in Kerberos parlance), in **sec_timeval_period_t** format. Processes may request authentication certificates with longer lifetimes up to, but not in excess of, the maximum allowable lifetime as determined by the effective policy for the account.

low_unix_id_person
The lowest UNIX number permissible for a person item in the registry.

low_unix_id_group
The lowest UNIX number permissible for a group item in the registry.

low_unix_id_org
The lowest UNIX number permissible for an organization item in the registry.

max_unix_id
The largest UNIX number permissible for any registry entry.

flags Property flags, in **sec_rgy_properties_flags_t** format.

realm The name of the cell, in **sec_rgy_name_t** form, for which this registry is the authentication service.

realm_uuid
The UUID of the same cell.

sec_rgy_properties_flags_t
A 32-bit bitset, containing flags concerning registry properties:

sec_rgy_prop_readonly
If set (TRUE), indicates that this registry is a query site.

sec_rgy_prop_auth_cert_unbound
If set (TRUE), the registry server will accept requests from any site.

sec_rgy_prop_shadow_passwd
If the shadow password flag is set (TRUE), the registry server will not include the account password when responding to a request for the user data from a specified account. This helps minimize the risk of an account password being intercepted while traveling over the network.

sec_rgy_prop_embedded_unix_id
Indicates that all UUIDs in this registry contain a UNIX number embedded. This implies that the UNIX numbers of objects in the registry cannot be changed, since UUIDs are immutable.

sec_rgy_override_t
A 32-bit integer used as a flag for registry override mode. Possible values are the constants **sec_rgy_no_override** and **sec_rgy_override**. When this mode is enabled, override data supplied by the node administrator will replace some of the data gotten from the registry for a given person/account under certain conditions. These conditions are as follows:

1. The registry permits the requested overrides to be set for this machine.
2. The override data is intended for person/account at hand.

When the mode is override off, data from the registry is returned to the end user or the application remains untouched.

sec_rgy_mode_resolve_t
A 32-bit integer used as a flag for resolve mode. Possible values are the constants **sec_rgy_no_resolve_pname** and **sec_rgy_resolve_pname**. When the mode is enabled, pathnames containing leading **//** (slashes) will be translated into a form understandable by the local machine's NFS.

sec_rgy_unix_passwd_buf_t
A character array of UNIX password strings.

Constants

The following constants are used in **sec_rgy_** calls:

sec_rgy_default_handle
The value of an unbound registry server handle.

sec_rgy_acct_key_t
The following 32-bit integer constants are used with the **sec_rgy_acct_key_t** data type:

sec_rgy_acct_key_none
Invalid key.

sec_rgy_acct_key_person
The person name alone is enough.

sec_rgy_acct_key_group
The person and group names are both necessary for the account abbreviation.

sec_rgy_acct_key_org
The person, group, and organization names are all necessary.

sec_rgy_acct_key_last
Key values must be less than this constant.

sec_rgy_pname_t_size
The maximum number of characters in a **sec_rgy_pname_t**.

sec_rgy_name_t_size
The maximum number of characters in a **sec_rgy_name_t**.

sec_rgy_domain_t
The following 32-bit integer constants are the possible values of the **sec_rgy_domain_t** data type:

sec_rgy_domain_person
The name in question refers to a person.

sec_rgy_domain_group
The name in question refers to a group.

sec_rgy_domain_org
The name in question refers to an organization.

sec_rgy_pgo_flags_t
A 32-bit constant equal to a variable of type **sec_rgy_pgo_flags_t** with no flags set.

sec_rgy_quota_unlimited
A 32-bit integer. Set the *quota* field of the **sec_rgy_pgo_item_t** type to this constant to override the registry quota limitation.

sec_rgy_acct_admin_flags_t
A 32-bit integer. This is the value of the **sec_rgy_acct_admin_flags_t** bitset when none of its flags are set.

sec_rgy_acct_auth_flags_none
A 32-bit integer. This is the value of the **sec_rgy_acct_auth_flags_t** bitset when none of its flags are set.

sec_rgy_acct_user_flags_t
A 16-bit integer. This is the value of the **sec_rgy_acct_user_flags_t** bitset when none of its flags are set.

sec_rgy_plcy_pwd_flags_t
A 16-bit integer. This is the value of the **sec_rgy_policy_pwd_flags_t** bitset when none of its flags are set.

sec_rgy_properties_flags_t
A 16-bit integer. This is the value of the **sec_rgy_properties_flags_t** bitset when none of its flags are set.

sec_rgy_override
A 32-bit integer, which turns registry override mode on. When this mode is enabled, override data supplied by the node administrator will replace some of the data gotten from the registry for a given person/account under certain conditions.

sec_rgy_no_override
A 32-bit integer, which turns off registry override mode.

sec_rgy_resolve_pname
A 32-bit integer, which turns on registry resolve mode. When the mode is enabled, pathnames containing leading ***//*** (slashes) will be translated into a form understandable by the local machine's NFS.

sec_rgy_no_resolve_pname
A 32-bit integer, which turns off registry resolve mode.

Files

/usr/include/dce/rgybase.idl
The **idl** file from which **rgybase.h** was derived.

Extended Registry Attribute Data Types

Synopsis

```
#include  <dce/sec_attr_base.h>
```

Data Types

The following data types are used in **sec_rgy_attr** calls:

sec_attr_twr_ref_t
A pointer to a tower. This data type is used with the **sec_attr_twr_set_t** data type to allow a client to pass an unallocated array of towers, which the server must allocate. Both data types are used in conjunction with the **sec_attr_bind_type_t** data type.

sec_attr_twr_set_t
A structure that defines an array of towers. This data type is used with the **sec_attr_twr_ref_t** data type to allow a client to pass an unallocated array of towers, which the server must allocate. Both data types are used in conjunction with the **sec_attr_bind_type_t** data type. The **sec_attr_twr_set_t** structure consists of the following elements:

count An unsigned 32-bit integer specifying the number of towers in the array.

towers [ ]
An array of pointers (of type **sec_attr_twr_ref_t**) to towers.

sec_attr_bind_type_t
A 32-bit integer that specifies the type of binding used by an attribute interface. The data type (which is used in conjunction with the **sec_attr_binding_t** data type) uses the following constants:

sec_attr_bind_type_string
An RPC string binding.

sec_attr_bind_type_twrs
A DCE protocol tower representation of a bindings.

sec_attr_bind_type_svrname
A name in **rpc_c_ns_syntax** format that identifies a CDS entry containing the server's binding information. This constant has the following structure:

name_syntax
Must be **rpc_c_ns_syntax_dce** to specify that DCE naming rules are used to specify **name**.

name A pointer to a name of a CDS entry in **rpc_c_ns_syntax_dce** syntax.

sec_attr_binding_t
A discriminated union that supplies information to generate a binding handle for a attribute trigger. This data type, which is used in conjunction with the **sec_attr_bind_info_t** data type, is composed of the following elements:

bind_type
A value of type **sec_attr_bind_type_t** that defines the type of binding used by an attribute interface. The contents of **tagged union** (see table) depend on the value of **sec_attr_bind_type_t**.

tagged_union
A tagged union specifying the binding handle. The contents of the tagged union depend on the value of **bind_type** as follows:

If bind_type is...	Then tagged_union is...
sec_attr_bind_type_string	A pointer to an unsigned 32-bit character string specifying an attribute's RPC string binding.
sec_attr_bind type_twrs	An attribute's tower binding representation of type **sec_attr_twr_set_t**.
sec_attr_bind_svrname	A pointer to a name of type **sec_attr_bind_type_t** that specifies a Cell Directory Service entry containing a attribute trigger's binding information.

sec_attr_binding_p_t
A pointer to a **sec_attr_binding_t** union.

sec_attr_bind_auth_info_type_t
An enumeration that defines whether or not the binding is authenticated. This data type is used in conjunction with the **sec_attr_bind_auth_info_t** data type to set up the authorization method and parameters for an RPC binding. The **sec_attr_bind_auth_info_type_t** type consists of the following elements:

sec_attr_bind_auth_none
The binding is not authenticated.

sec_attr_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_attr_bind_auth_info_t
A discriminated union that defines authorization and authentication parameters for a binding. This data type is used in conjunction with the **sec_attr_bind_auth_info_type_t** data type to set up the authorization method and parameters for an RPC binding. The **sec_attr_bind_auth_info_t** data type consists of the following elements:

info_type
A **sec_attr_bind_auth_info_type_t** data type that specifies whether or not the binding is authenticated. The contents of **tagged union** (below) depend on the value of **sec_attr_bind_auth_info_type_t**.

tagged_union
A tagged union specifying the method of authorization and the authorization parameters. For unauthenticated bindings (**sec_attr_bind_auth_info_type_t** = **sec_attr_bind_auth_none**), no parameters are supplied. For authenticated bindings (**sec_attr_bind_auth_info_type_t** = **sec_attr_bind_auth_dce**), the following union is supplied:

svr_princ_name
A pointer to a character string that specifies the principal name of the server referenced by the binding handle.

protect_level
An unsigned 32-bit integer indicating the protection level for RPC calls made using the server binding handle. The protection level determines the degree to which

authenticated communications between the client and the server are protected by the authentication service specified by **authn_svc**.

If the RPC runtime or the RPC protocol in the bound protocol sequence does not support a specified level, the level is automatically upgraded to the next higher supported level. The possible protection levels are as follows:

Protection Level	Description
rpc_c_protect_level_default	Uses the default protection level for the specified authentication service. The default protection level for DCE shared-secret key authentication is **rpc_c_protect_level_pkt_value**
rpc_c_protect_level_none	Performs no authentication: tickets are not exchanged, session keys are not established, client PACs or names are not certified, and transmissions are in the clear. Note that although uncertified PACs should not be trusted, they may be useful for debugging, tracing, and measurement purposes.
rpc_c_protect_level_connect	Authenticates only when the client establishes a relationship with the server.
rpc_c_protect_level_call	Authenticates only at the beginning of each remote procedure call when the server receives the request. This level does not apply to remote procedure calls made over a connection-based protocol sequence (that is, **ncacn_ip_tcp**). If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.
rpc_c_protect_level_pkt	Ensures that all data received is from the expected client.
rpc_c_protect_level_pkt_integ	Ensures and verifies that none of the data transferred between client and server has been modified. This is the highest protection level that is guaranteed to be present in the RPC runtime.
rpc_c_protect_level_pkt_privacy	Authenticates as specified by all of the previous levels and also encrypts each RPC argument value. This is the highest protection level, but is not guaranteed to be present in the RPC runtime.

authn_svc

Specifies the authentication service to use. The exact level of protection provided by the authentication service is specified by *protect_level*. The supported authentication services are as follows:

Authentication Service	Description
rpc_c_authn_none	No authentication: no tickets are exchanged, no session keys established, client PACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for remote procedure calls made using this binding.
rpc_c_authn_dce_secret	DCE shared-secret key authentication.

Authentication Service	Description
rpc_c_authn_default	Default authentication service. The current default authentication service is DCE shared-secret key; therefore, specifying **rpc_c_authn_default** is equivalent to specifying **rpc_c_authn_dce_secret**.
rpc_c_authn_dce_public	DCE public key authentication (reserved for future use).

authz_svc
Specifies the authorization service implemented by the server for the interface. The validity and trustworthiness of authorization data, like any application data, is dependent on the authentication service and protection level specified. The supported authorization services are as follows:

Authentication Service	Description
rpc_c_authz_none	Server performs no authorization. This is valid only if **authn_svc** is set to **rpc_c_authn_none**, specifying that no authentication is being performed.
rpc_c_authz_name	Server performs authorization based on the client principal name. This value cannot be used if **authn_svc** is **rpc_c_authn_none**.
rpc_c_authz_dce	Server performs authorization using the client's DCE privilege attribute certificate (PAC) sent to the server with each remote procedure call made with this binding. Generally, access is checked against DCE ACLs.

sec_attr_bind_info_t
A structure that specifies attribute trigger binding information. This data type, which is used in conjunction with the **sec_attr_schema_entry_t** data type, contains of the following elements:

auth_info
The binding authorization information of type **sec_attr_bind_auth_info_t**.

num_bindings
An unsigned 32-bit integer specifying the number of binding handles in **bindings**.

bindings
An array of **sec_attr_binding_t** data types that specify binding handles.

sec_attr_bind_info_p_t
A pointer to a **sec_attr_bind_info_t** union.

sec_attr_encoding_t
An enumerator that contains attribute encoding tags used to define the legal encodings for attribute values. The data type, which is used in conjunction with the **sec_attr_value_t** and **sec_attr_schema_entry_t** data types, consists of the following elements:

sec_attr_enc_any
The attribute value can be of any legal encoding type. This encoding tag is legal only in a schema entry. An attribute entry must contain a concrete encoding type.

sec_attr_enc_void
The attribute has no value. It is simple a marker that is either present or absent.

sec_attr_enc_printstring
The attribute value is a printable IDL string in DCE portable character set.

sec_attr_enc_printstring_array
The attribute value is an array of printstrings.

sec_attr_enc_integer
The attribute value is a signed 32-bit integer.

sec_attr_enc_bytes
The attribute value is a string of bytes. The string is assumed to be a pickle or some other self describing type. (See also the **sec_attr_enc_bytes_t** data type.)

sec_attr_enc_confidential_bytes
The attribute value is a string of bytes that have been encrypted in the key of the principal object to which the attribute is attached. The string is assumed to be a pickle or some other self describing type. This encoding type is useful only when attached to a principal object, where it is decrypted and encrypted each time the principal's password changes. (See also the **sec_attr_enc_bytes_t** data type.)

sec_attr_enc_i18n_data
The attribute value is an internationalized string of bytes with a tag identifying the OSF registered codeset used to encode the data. (See also the **sec_attr_i18n_data_t** data type.)

sec_attr_enc_uuid
The attribute is a value of type **uuid_t**, a DCE UUID.

sec_attr_enc_attr_set
The attribute value is an attribute set, a vector of attribute UUIDs used to associate multiple related attribute instances which are members of the set. (See also the **sec_attr_enc_attr_set_t** data type.)

sec_attr_enc_binding
The attribute value is a **sec_attr_bind_info_t** data type that specifies DCE server binding information.

sec_attr_enc_trig_binding
This encoding type is returned by **rs_attr_lookup** call. It informs the client agent of the trigger binding information of an attribute with a query trigger.

Unless **sec_attr_enc_void** or **sec_attr_enc_any** is specified, the attribute values must conform to the attribute's encoding type.

sec_attr_enc_bytes_t
A structure that defines the length of attribute encoding values for attributes encoded as **sec_attr_enc_bytes** and **sec_attr_enc_confidential_bytes**. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

length An unsigned 32-bit integer that defines the data length.

data [ ]
An array of bytes specifying the length of attribute encoding data.

sec_attr_i18n_data_t
A structure that defines the codeset used for attributes encoded as **sec_attr_enc_il8n_data** and the length of the attribute encoding values. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

codeset
An unsigned 32-bit identifier of a codeset registered with the Open Software Foundation.

length An unsigned 32-bit integer that defines the data length.

data [ ]
An array of bytes specifying the length of attribute encoding data.

sec_attr_enc_attr_set_t
A structure that that supplies the UUIDs of each member of an attribute set. The structure, which is used in conjunction with the **sec_attr_value_t** data type, consists of

num_members
An unsigned 32-bit integer specifying the total number of attribute's in the set.

members [ ]
An array containing values of type **uuid_t**, the UUID of each member in the set.

sec_attr_enc_printstring_t
A structure that contains a printstring.

sec_attr_enc_printstring_p_t
A pointer to a **sec_attr_enc_printstring_t** structure.

sec_attr_enc_str_array_t
A structure that defines a printstring array. It consists of

num_strings
An unsigned 32-bit integer specifying the number of strings in the array.

strings [ ]
An array of pointers (of type **sec_attr_enc_print_string_p_t**) to printstrings.

sec_attr_value_t
A discriminated union that defines attribute values. The union, which is used in conjunction with the **sec_attr_t** data type, consists of the following elements:

attr_encoding
A **sec_attr_encoding_t** data type that defines attribute encoding. The contents of **tagged union** depend on the value of **sec_attr_encoding_t**.

tagged_union
A tagged union whose contents depend on **attr_encoding** as follows:

If attr_encoding is...	Then tagged_union is...
sec_attr_enc_void	NULL
sec_attr_enc_printstring	A pointer to **printstring**
sec_attr_enc_printstring_array	A pointer to an array of **printstring** s
sec_attr_enc_integer	**signed_int**, a 32-bit signed integer
sec_attr_enc_bytes	**bytes**, a pointer to a structure of type **sec_attr_enc_bytes_t**
sec_attr_enc_confidential_bytes	**bytes**, a pointer to a structure of type **sec_attr_enc_bytes_t**
sec_attr_enc_i18n_data	**idata**, a pointer to a structure of type **sec_attr_i18n_data_t**
sec_attr_end_uuid	**uuid**, a value of type **uuid_t**
sec_attr_enc_attr_set	**attr_set**, a pointer to a structure of type **sec_attr_enc_attr_set_t**
sec_attr_enc_binding	**binding**, a pointer to a structure of type **sec_attr_binding_info_t**

sec_attr_t
A structure that defines an attribute. The structure consists of

attr_id
A value of type **uuid_t**, the UUID of the attribute.

attr_value
A value of type **sec_attr_value_t**.

sec_attr_acl_mgr_info_t
A structure that contains the access control information defined in a schema entry for an attribute. The structure, which is used in conjunction with the **sec_attr_schema_entry_t** data type, consists of the following elements:

acl_mgr_type
The value of type **uuid_t** that specifies the UUID of the ACL manager type that supports the object type to which the attribute can be attached. This field provides a well-defined context for evaluating the permission bits needed to operate on the attribute. The following table lists the ACL manager types for registry objects.

Registry Object Type	ACL Manager Type	Valid Permissions
principal	06ab9320-0191-11ca-a9e8-08001e039d7d	rcDnfmaug
group	06ab9640-0191-11ca-a9e8-08001e039d7d	rctDnfmM
organization	06ab9960-0191-11ca-a9e8-08001e039d7d	rctDnfmM
directory	06ab9c80-0191-11ca-a9e8-08001e039d7d	rcidDn
policy	06ab8f10-0191-11ca-a9e8-08001e039d7d	rcma
replist	2ac24970-60c3-11cb-b261-08001e039d7d	cidmAI

query_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to access the attribute's value.

update_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to update the attribute's value.

test_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to test the attribute's value.

delete_permset
Data of type **sec_acl_permset_t** that defines the permission bits needed to delete an attribute instance.

sec_attr_acl_mgr_info_p_t
A pointer to a **sec_attr_acl_mgr_info_t** structure.

sec_attr_acl_mgr_info_set_t
A structure that defines an attribute's ACL manager set. The structure consists of the following elements:

num_acl_mgrs
An unsigned 32-bit integer that specifies the number of ACL managers in the ACL manager set.

mgr_info [ ]
An array of pointers of type **sec_attr_mgr_info_p_t** that define the ACL manager types in the ACL manager set and the permission sets associated with the ACL manager type.

sec_attr_intercell_action_t
An enumerator that specifies the action that should be taken by the privilege service when it reads acceptable attributes from a foreign cell. A foreign attribute is acceptable only if there is either a schema entry for the foreign cell or if **sec_attr_intercell_act_accept** is set to **true**.

This enumerator, which is used in conjunction with the **sec_attr_schema_entry_t** data type, is composed of the following elements:

sec_attr_intercell_act_accept
If the **unique** flag in the **sec_attr_schema_entry_t** data type is not set on, retain the attribute. If the **unique** flag is set on, retain the attribute only if its value is unique among all attribute instances of the same attribute type within the cell.

sec_attr_intercell_act_reject
Discard the input attribute.

sec_attr_intercell_act_evaluate
Use the binding information in the *trig_binding* field of this **sec_attr_schema_entry_t** data type to make a **sec_attr_trig_query** call to a trigger server. That server determines whether to retain the attribute value, discard the attribute value, or map the attribute to another value.

sec_attr_trig_type_t
Specifies the trigger type, a flag that determines whether an attribute trigger should be invoked for query operations. The data type, which is used in conjunction with the **sec_attr_schema_entry_t** data type, uses the following constants:

sec_attr_trig_type_query
The attribute trigger server is invoked for query operations.

sec_attr_trig_type_query
The attribute trigger server is invoked for update operations.

sec_attr_schema_entry_t

A structure that defines a complete attribute entry for the schema catalog. The entry is identified by both a unique string name and a unique attribute UUID. Although either can either can be used as a retrieval key, the string name should be used for interactive access to the attribute and the UUID for programmatic access. The attribute UUID is used to identify the semantics defined for the attribute type in the schema.

The **sec_attr_schema_entry_t** data type consists of the following elements:

attr_name
A pointer to the attribute name.

attr_id
A value of type **uuid_t** that identifies the attribute type.

attr_encoding
An enumerator of type **sec_attr_encoding_t** that specifies the attribute's encoding.

acl_mgr_set
A structure of type **sec_attr_acl_mgr_info_set_t** that specifies the ACL manager types that support the objects on which attributes of this type can be created and the permission bits supported by that ACL manager type.

schema_entry_flags
An unsigned integer of type **sec_attr_sch_entry_flags_t** that defines bitsets for the following flags:

unique
When set on, this flag indicates that each instance of this attribute type must have a unique value within the cell for the object type implied by the ACL manager type. If this flag is not set on, uniqueness checks are not performed for attribute writes.

multi_valued
When set on, this flag indicates that this attribute type may be multivalued; in other words, multiple instances of the same attribute type can be attached to a single registry object. If this flag is not set on, only one instance of this attribute type can be attached to an object.

reserved
When set on, this flag prevents the schema entry from being deleted through any interface or by any user. If this flag is not set on, the entry can be deleted by any authorized principal.

use_defaults
When set on, the system-defined default attribute value will be returned on a client query if an instance of this attribute does not exist on the queried object. If this flag is not set on, system defaults are not used.

intercell_action
An enumerator of type **sec_attr_intercell_action_t** that specifies how the privilege service will handle attributes from a foreign cell.

trig_types
A flag of type **sec_attr_trig_type_t** that specifies whether whether a trigger can perform update or query operations.

trig_binding
A pointer to a structure of type **sec_attr_bind_info_t** that supplies the attribute trigger binding handle.

scope A pointer to a string that defines the objects to which the attribute can be attached.

comment
A pointer to a string that contains general comments about the attribute.

sec_attr_schema_entry_parts_t
A 32-bit bitset containing flags that specify the schema entry fields that can be modified on a schema entry update operation. This data type contains the following flags:

sec_attr_schema_part_name
If set, indicates that the attribute name (**attr_name**) can be changed.

sec_attr_schema_part_reserved
If set, indicates that the setting of the flag that determines whether or not the schema entry can be deleted (**reserved**) can be changed.

sec_attr_schema_part_defaults
If set, indicates that the flag that determines whether or not a query for a nonexistent attribute will not result in a search for a system default (**apply_default**) can be changed.

sec_attr_schema_part_trig_bind
If set, indicates that the trigger's binding information (**trig_binding**) can be changed.

sec_attr_schema_part_comment
If set, indicates whether or not comments associated with the schema entry (**comment**) can be changed.

sec_attr_component_name_t
A pointer to a character string used to further specify the object to which the attribute is attached. (Note that this data type is analogous to the **sec_acl_component_name_t** data type in the ACL interface.)

sec_attr_cursor_t
A structure that provides a pointer into a registry database and is used for multiple database operations.

This cursor must minimally represent the object indicated by **xattrschema** in the schema interfaces, or *component_name* in the attribute interfaces. The cursor may additionally represent an entry within that schema or an attribute instance on that component.

sec_attr_srch_cursor_t
A structure that provides a pointer into a registry database and is used for multiple database operations. The cursor must minimally represent the list of all objects managed by this server that possess the search attributes

specified in the **sec_attr_srch_cursor_init** routine. It may additionally represent a given object within this list as well as attribute instance(s) possessed by that object.

sec_attr_trig_cursor_t
A structure that provides an attribute trigger cursor for interactive operations. The structure consists of the following elements:

source
A value of type **uuid_t** that provides a UUID to identify the server that initialized the cursor.

object_handle
A signed 32-bit integer that identifies the object (specified by **xattrschema** in the schema interface or *component_name* in the attribute interface) upon which the operation is being performed.

entry_handle
A signed 32-bit integer that identifies the current entry (*schema_entry* in the schema interface or *attribute instance* in the attribute interface) for the operation.

valid A Boolean field with the following values:

true (1)
Indicates an initialized cursor.

false (0)
Indicates an uninitialized cursor.

sec_attr_trig_timeval_sec_t
A 32-bit integer containing the seconds portion of a UNIX **timeval_t**, to be used when expressing absolute dates.

Files

/usr/include/dce/sec_attr_base.idl
The **idl** file from which **sec_attr_base.h** was derived.

Constants

The following constants are used in **sec_attr** calls:

sec_attr_bind_auth_dce
The binding uses DCE shared-secret key authentication.

sec_attr_bind_auth_none
The binding is not authenticated.

sec_attr_bind_type_string
The attribute uses an RPC string binding.

sec_attr_bind_type_svrname
The attribute uses a name in **rpc_c_ns_syntax** format that identifies a CDS entry containing the server's binding information. This constant has the following structure:

name_syntax
Must be **rpc_c_ns_syntax_dce** to specify that DCE naming rules are used to specify **name**.

name A pointer to a name of a CDS entry in **rpc_c_ns_syntax_dce** syntax.

sec_attr_bind_type_twr
The attribute uses a DCE protocol tower binding representation.

sec_attr_trig_type_t
The following 32-bit constants are used with the **sec_attr_trig_type_t** data type:

sec_attr_trig_type_query The trigger server can perform only query operations.

sec_attr_trig_type_update The trigger server can perform only update operations.

sec_attr_intercell_action_t
The following constants are used with the **sec_attr_intercell_action_t** data type:

sec_attr_intercell_act_accept
If the **unique** flag in the **sec_attr_schema_entry_t** data type is not set on, retain attributes from a foreign cell. If the **unique** flag is set on, retain the foreign attribute only if its value is unique among all attribute instances of the same attribute type within the cell.

sec_attr_intercell_act_reject
Discard attributes from a foreign cell.

sec_attr_intercell_act_evaluate
A trigger server determines whether to retain foreign attributes, discard foreign attributes, or map foreign attribute to another value(s).

sec_attr_schema_entry_parts_t
The following constants are used with the **sec_attr_schema_entry_parts_t** data type:

sec_attr_schema_part_name
Indicates that the attribute name can be changed in an schema update operation.

sec_attr_schema_part_reserved
Indicates that the setting of the **reserved** flag can be changed in a schema entry update.

sec_attr_schema_part_defaults
Indicates that the **apply_default** flag can be changed in a schema entry update operation.

sec_attr_schema_part_trig_bind
Indicates that trigger binding information can be changed in a schema entry update operation.

sec_attr_schema_part_comment
Indicates that comments associated with the schema entry can be changed in a schema entry update.

Login API Data Types

Synopsis

```
#include  <dce/sec_login.h>
```

Data Types

The following data types are used in **sec_login_** calls:

sec_login_handle_t
This is an opaque pointer to a data structure representing a complete login context. The context includes a principal's network credentials, as well as other account information. The network credentials are also referred to as the principal's ticket-granting ticket.

sec_login_flags_t
A 32-bit set of flags describing restrictions on the use of a principal's validated network credentials. Currently, only one flag is implemented. Possible values are:

sec_login_no_flags
No special flags are set.

sec_login_credentials_private
Restricts the validated network credentials to the current process. If this flag is not set, it is permissible to share credentials with descendents of current process.

sec_login_auth_src_t
An enumerated set describing how the login context was authorized. The possible values are:

sec_login_auth_src_network
Authentication accomplished through the normal network authority. A login context authenticated this way will have all the network credentials it ought to have.

sec_login_auth_src_local
Authentication accomplished via local data. Authentication occurs locally if a principal's account is tailored for the local machine, or if the network authority is unavailable. Since login contexts authenticated locally have no network credentials, they may not be used for network operations.

sec_login_auth_src_overridden
Authentication accomplished via the override facility.

sec_login_passwd_t
The **sec_login_get_pwent()** call will return a pointer to a password structure, which depends on the underlying registry structure.

In most cases, the structure will look like that supported by Berkeley 4.4BSD and OSF/1, which looks like this:

```
struct passwd {
 char  *pw_name;              * user name *
 char  *pw_passwd;            * encrypted password *
 int    pw_uid;               * user uid *
 int    pw_gid;               * user gid *
 time_t pw_change;            * password change time *
```

```
 char  *pw_class;            * user access class *
 char  *pw_gecos;            * Honeywell login info *
 char  *pw_dir;              * home directory *
 char  *pw_shell;            * default shell *
 time_t pw_expire;           * account expiration *
};
```

sec_passwd_rec_t
A structure containing either a plaintext password or a preencrypted buffer of password data. The **sec_passwd_rec_t** structure consists of three components:

version_number
The version number of the password.

pepper
A character string combined with the password before an encryption key is derived from the password.

key A structure consists of the following components:

key_type
The key type can be the following:

sec_passwd_plain
Indicates that a printable string of data is stored in **plain**.

sec_passwd_des
Indicates that an array of data is stored in **des_key**.

tagged_union
A structure specifying the password. The value of the structure depends on **key_type**. If **key_type** is **sec_passwd_plain**, structure contains **plain**, a character string. If **key_type** is **sec_passwd_des**, the structure contains **des_key**, a DES key of type **sec_passwd_des_key_t**.

Constants

The following constants are used in **sec_login_** calls:

sec_login_default_handle
The value of a login context handle before setup or validation.

sec_login_flags_t
The following two constants are used with the **sec_login_flags_t** type:

sec_login_no_flags
No special flags are set.

sec_login_credentials_private
Restricts the validated network credentials to the current process. If this flag is not set, it is permissible to share credentials with descendents of current process.

sec_login_remote_uid
Used in the **sec_login_passwd_t** structure for users from remote cells.

sec_login_remote_gid
Used in the **sec_login_passwd_t** structure for users from remote cells.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **sec_login.h** was derived.

Extended Privilege Attribute API Data Types

Synopsis

```
#include <dce/id_epac.h>
#include <dce/nbase.h>
```

Data Types

The following data types are used in extended privilege attribute calls and in the **sec_login_cred** calls that implement extended privilege attributes.

sec_cred_cursor_t
A structure that provides an input/output cursor used to iterate through a set of delegates in the **sec_cred_get_delegate()** or **sec_login_cred_get_delegate()** calls. This cursor is initialized by the **sec_cred_initialize_cursor()** or **sec_login_cred_init_cursor()** call.

sec_cred_attr_cursor_t
A structure that provides an input/output cursor used to iterate through a set of extended attributes in the **sec_cred_get_extended_attributes()** call. This cursor is initialized by the **sec_cred_initialize_attr_cursor()** call.

sec_id_opt_req_t
A structure that specifies application-defined optional restrictions. The **sec_id_opt_req_t** data type is composed of the following elements:

restriction_len
An unsigned 16-bit integer that defines the size of the restriction data.

restrictions
A pointer to a **byte_t** that contains the restriction data.

sec_rstr_entry_type_t
An enumerator that specifies the entry types for delegate and target restrictions. This data type is used in conjunction with the **sec_id_restriction_t** data type where the specific UUID(s), if appropriate, are supplied. It consists of the following components:

sec_rstr_e_type_user
The target is a local principal identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_group
The target is a local group identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_foreign_user
The target is a foreign principal identified by principal and cell UUID.

sec_rstr_e_type_foreign_group
The target is a foreign group identified by group and cell UUID.

sec_rstr_e_type_foreign_other
The target is any principal that can authenticate to the foreign cell identified by UUID.

sec_rstr_e_type_any_other
The target is any principal that can authenticate to any cell, but is not identified in any other type entry.

sec_rstr_e_type_no_other
No principal can act as a target or delegate.

sec_id_restriction_t
A discriminated union that defines delegate and target restrictions. The union, which is used in conjunction with the **sec_restriction_set_t** data type, consists of the following elements:

entry_type
A **sec_rstr_entry_type_t** that defines the ACL entry types for delegate and target restrictions. The value of **tagged_union** depends on the value of **entry_type**.

tagged_union
A tagged union whose contents depend on **entry_type** as follows:

If entry_type is...	Then tagged_union is...
sec_rstr_e_type_any_other	NULL
sec_rstr_e_type_foreign_other	**foreign_id** that identifies the foreign cell.
sec_rstr_e_type_user **Sec_rstr_e_type_group**	**id**, a **sec_id_t** that identifies the user or group.
sec_rstr_e_type_foreign_user **sec_rstr_e_type_foreign_group**	**foreign_id**, a **sec_id_foreign_t** that identifies the foreign user or group.

sec_id_restriction_set_t
A structure that that supplies delegate and target restrictions. The structure consists of

num_restrictions
A 16-bit unsigned integer that defines the number of restrictions in **restrictions**.

restrictions
A pointer to a **sec_id_restriction_t** that contains the restrictions.

sec_id_compatibility_mode_t
A unsigned 16 bit integer that defines the compatibility between current and pre-1.1 servers. The data type uses the following constants:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The 1.0 PAC data extracted from the EPAC of the chain initiator.

sec_id_compat_mode_caller
Compatibility mode is on. The 1.0 PAC data extracted from the last delegate in the delegation chain.

sec_id_delegation_type_t
An unsigned 16 bit integer that defines the delegation type. The data type uses the following constants:

sec_id_deleg_type_none
Delegation is not allowed.

sec_id_deleg_type_traced
Traced delegation is allowed.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation is allowed.

sec_id_pa_t
An structure that contains pre-1.1 PAC data extracted from an EPAC of a current version server. This data type, which is used for compatibility with pre-1.1 servers, consists of the following elements:

realm A value of type **sec_id_t** that contains the UUID that identifies the cell in which the principal associated with the PAC exists.

principal
A value of type **sec_id_t** that contains the UUID of the principal.

group A value of type **sec_id_t** that contains the UUID of the principal's primary group.

num_groups
An unsigned 16-bit integer that specifies the number of groups in the principal's groupset.

groups
An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

num_foreign_groupsets
An unsigned 16-bit integer that specifies the number of foreign groups for the principal's groupset.

foreign_groupsets
An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

sec_id_pac_t
An structure that contains a pre-1.1 PAC. This data type, which is used as output of the **sec_cred_get_v1_pac** call, consists of the following elements:

pac_type
A value of type **sec_id_pac_format_t** that can be used to describe the PAC format.

authenticated
A boolean field that indicates whether or not the PAC is authenticated (obtained from an authenticated source). FALSE indicates that the PAC is not authenticated. No authentication protocol was used in the rpc that transmitted the identity of the caller. TRUE indicates that the PAC is authenticated.

realm A value of type **sec_id_t** that contains the UUID that identifies the cell in which the principal associated with the PAC exists.

principal
A value of type **sec_id_t** that contains the UUID of the principal.

group For local principals, a value of type **sec_id_t** that contains the UUID of the principal's primary group.

num_groups
An unsigned 16-bit integer that specifies the number of groups in the principal's groupset.

groups
An array of pointers to **sec_id_t**s that contain the UUIDs of the each group in the principal's groupset.

num_foreign_groups
An unsigned 16-bit integer that specifies the number of foreign groups in the principal's groupset.

foreign_groups
An array of pointers to **sec_id_t**s that contain the UUIDs of the each foreign group in the principal's groupset.

sec_id_pac_format_t
An enumerator that can be used to describe the PAC format.

sec_id_t
A structure that contains UUIDs for principals, groups, or organizations and an optional printstring name. Since a UUID is an handle for the object's identity, the **sec_id_t** data type is the basic unit for identifying principals, groups, and organizations.

Because the printstring name is dynamically allocated, this datatype requires a destructor function. Generally, however, the **sec_id_t** is embedded in other data types (ACLs, for example), and these datatypes have a destructor function to release the printstring storage.

The **sec_id_t** data type is composed of the following elements:

uuid A value of type **uuid_t**, the UUID of the principal, group, or organization.

name A pointer to a character string containing the name of the principal, group, or organization.

sec_id_foreign_t
A structure that contains UUIDs for principals, groups, or organizations for objects in a foreign cell and the UUID that identifies the foreign cell. The **sec_id_foreign_t** data type is composed of the following elements:

id A value of type **sec_id_t** that contains the UUIDs of the objects from the foreign cell.

realm A value of type **sec_id_t** that contains the UUID of the foreign cell.

sec_id_foreign_groupset_t
A structure that contains UUIDs for set of groups in a foreign cell and the UUID that identifies the foreign cell. The **sec_id_foreign_groupset_t** data type is composed of the following elements:

realm A value of type **sec_id_t** that contain the UUID of the foreign cell.

num_groups
An unsigned 16-bit integer specifying the number of group UUIDs in **groups**.

groups
A printer to a **sec_id_t** that contains the UUIDs of the groupset from the foreign cell.

Constants

The following constants are used in the extended privilege attribute calls and in the the **sec_login** calls that implement extended privilege attributes:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The 1.0 PAC data extracted from the EPAC of the chain initiator.

sec_id_compat_mode_caller
Compatibility mode is on. The 1.0 PAC data extracted from the last delegate in the delegation chain.

sec_id_deleg_type_none
Delegation is not allowed.

sec_id_deleg_type_traced
Traced delegation is allowed.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation is allowed.

sec_rstr_e_type_user
The delegation target is a local principal identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_group
The delegation target is a local group identified by UUID. This type conforms with the POSIX 1003.6 standard.

sec_rstr_e_type_foreign_user
The delegation target is a foreign principal identified by principal and cell UUID.

sec_rstr_e_type_foreign_group
The delegation target is a foreign group identified by group and cell UUID.

sec_rstr_e_type_foreign_other
The delegation target is any principal that can authenticate to the foreign cell identified by UUID.

sec_rstr_e_type_any_other
The delegation target is any principal that can authenticate to any cell, but is not identified in any other type entry.

sec_rstr_e_type_no_other
No pincipal can act as a target or delegate.

Files

/usr/include/dce/sec_cred.idl
The **idl** file from which **sec_cred.h** was derived.

/usr/include/dce/sec_epac.idl
The **idl** file from which **sec_epac.h** was derived.

/usr/include/dce/sec_nbase.idl
The **idl** file from which **sec_nbase.h** was derived.

ACL API Data Types

Synopsis

```
#include  <dce/aclbase.h>
```

Data Types

The following data types are used in **sec_acl_** calls:

sec_acl_handle_t
A pointer to an opaque handle bound to an ACL that is the subject of a test or examination. The handle is bound to the ACL with **sec_acl_bind()**. An unbound handle has the value **sec_acl_default_handle**.

sec_acl_posix_semantics_t
A flag that indicates which, if any, POSIX ACL semantics an ACL manager supports. The following constants are defined for use with the **sec_acl_posix_semantics_t** data type:

sec_acl_posix_no_semantics
The manager type does not support POSIX semantics.

sec_acl_posix_mask_obj
The manager type supports the **mask_obj** entry type and POSIX 1003.6 Draft 12 ACL mask entry semantics.

sec_acl_t
This data type is the fundamental type for the ACL manager interfaces. The **sec_acl_t** type contains a complete access control list, made up of a list of entry fields (type **sec_acl_entry_t**). The default cell identifies the authentication authority for simple ACL entries (foreign entries identify their own foreign cells). The **sec_acl_manager_type** identifies the manager to interpret this ACL.

The **sec_acl_t** type is a structure containing the following fields:

default_realm
A structure of type **sec_acl_id_t**, this identifies the UUID and (optionally) the name of the default cell.

sec_acl_manager_type
Contains the UUID of the ACL manager type.

num_entries
An unsigned 32-bit integer containing the number of ACL entries in this ACL.

sec_acl_entries
An array containing **num_entries** pointers to different ACL entries, each of type **sec_acl_entry_t**.

sec_acl_p_t
This data type, simply a pointer to a **sec_acl_t**, is for use with the **sec_acl_list_t** data type.

sec_acl_list_t
This data type is a structure containing an unsigned 32-bit integer **num_acls** that describes the number of ACLs indicated by its companion array of pointers, **sec_acls**, of type **sec_acl_p_t**.

sec_acl_entry_t

The **sec_acl_entry_t** type is a structure made up of the following components:

perms A set of flags of type **sec_acl_permset_t** that describe the permissions granted for the principals identified by this ACL entry. Note that if a principal matches more than one ACL entry, the effective permissions will be the most restrictive combination of all the entries.

entry_info
A structure containing two members:

entry_type
A flag of type **sec_acl_entry_type_t**, indicating the type of ACL entry.

tagged_union
A tagged union whose contents depend on the type of the entry.

The types of entries indicated by **entry_type** can be the following:

sec_acl_e_type_user_obj
The entry contains permissions for the implied user object. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_group_obj
The entry contains permissions for the implied group object. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_other_obj
The entry contains permissions for principals not otherwise named through user or group entries. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_user
The entry contains a key that identifies a user. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_group
The entry contains a key that identifies a group. This type is described in the POSIX 1003.6 standard.

sec_acl_e_type_mask_obj
The entry contains the maximum permissions for all entries other than **mask_obj**, **unauthenticated**, **user_obj**, **other_obj**.

sec_acl_e_type_foreign_user
The entry contains a key that identifies a user and the foreign realm.

sec_acl_e_type_foreign_group
The entry contains a key that identifies a group and the foreign realm.

sec_acl_e_type_foreign_other
The entry contains a key that identifies a foreign realm. Any user that can authenticate to the foreign realm will be allowed access.

sec_acl_e_type_any_other
The entry contains permissions to be applied to any accessor who

can authenticate to any realm, but is not identified in any other entry (except **sec_acl_e_type_unauthenticated**).

sec_acl_e_type_unauthenticated
The entry contains permissions to be applied when the accessor does not pass authentication procedures. A privilege attribute certificate will indicate that the caller's identity is not authenticated. The identity is used to match against the standard entries, but the access rights are masked by this mask. If this mask does not exist in an ACL, the ACL is assumed to grant no access and all unauthenticated access attempts will be denied.

Great care should be exercised when allowing unauthenticated access to an object. Almost by definition, unauthenticated access is very easy to spoof. The presence of this mask on an ACL essentially means that anyone can get at least as much access as allowed by the mask.

sec_acl_e_type_extended
The entry contains additional pickled data. This kind of entry cannot be interpreted, but can be used by an out-of-date client when copying an ACL from one manager to another (assuming that the two managers each understand the data).

The contents of the tagged union depend on the entry type.

For the following entry types, the union contains a UUID and an optional print string (called **entry_info.tagged_union.id** with type **sec_id_t**) for an identified local principal, or for an identified foreign realm.

- **sec_acl_e_type_user**
- **sec_acl_e_type_group**
- **sec_acl_type_foreign_other**

For the following entry types, the union contains two UUIDs and optional print strings (called **entry_info.tagged_union.foreign_id** with type **sec_id_foreign_t**) for an identified foreign principal and its realm.

- **sec_acl_e_type_foreign_user**
- **sec_acl_e_type_foreign_group**

For an extended entry (**sec_acl_e_type_extended**), the union contains **entry_info.tagged_union.extended_info**, a pointer to an information block of type **sec_acl_extend_info_t**.

sec_acl_permset_t
A 32-bit set of permission flags. The flags currently represent the conventional file system permissions (read, write, execute) and the extended DFS permissions (owner, insert, delete).

The unused flags represent permissions that can only be interpreted by the manager for the object. For example, **sec_acl_perm_unused_00000080** may mean to one ACL manager that withdrawals are allowed, and to another ACL manager that rebooting is allowed.

The following constants are defined for use with the **sec_acl_permset_t** data type:

sec_acl_perm_read
The ACL allows read access to the protected object.

sec_acl_perm_write
The ACL allows write access to the protected object.

sec_acl_perm_execute
The ACL allows execute access to the protected object.

sec_acl_perm_control
The ACL allows the ACL itself to be modified.

sec_acl_perm_insert
The ACL allows insert access to the protected object.

sec_acl_perm_delete
The ACL allows delete access to the protected object.

sec_acl_perm_test
The ACL allows access to the protected object only to the extent of being able to test for existence.

The bits from 0x00000080 to 0x80000000 are not used by the conventional ACL permission set. Constants of the form **sec_acl_perm_unused_00000080** have been defined so application programs can easily use these bits for extended ACLs.

sec_acl_extend_info_t
This is an extended information block, provided for future extensibility. Primarily, this allows an out-of-date client to read an ACL from a newer manager and apply it to another (up-to-date) manager. The data cannot be interpreted by the out-of-date client without access to the appropriate pickling routines (that presumably are unavailable to such a client).

In general, ACL managers should not accept ACLs that contain entries the manager does not understand. The manager clearly cannot perform the security service requested by an uninterpretable entry, and it is considered a security breach to lead a client to believe that the manager is performing a particular class of service if the manager cannot do so.

The data structure is made up of the following components:

extension_type
The UUID of the extension type.

format_label
The format of the label, in **ndr_format_t** form.

num_bytes
An unsigned 32-bit integer indicating the number of bytes containing the pickled data.

pickled_data
The byte array containing the pickled data.

sec_acl_type_t
The **sec_acl_type_t** type differentiates among the various types of ACLs an object can possess. Most file system objects will only have one ACL controlling the access to that object, but objects that control the creation of other objects (sometimes referred to as *containers*) may have more. For example, a directory can have three different ACLs: the directory ACL, controlling access to the directory; the initial object (or default object) ACL, which serves as a mask when creating new objects in the directory; and the initial directory (or default directory) ACL, which serves as a mask when creating new directories (containers).

The **sec_acl_type_t** is an enumerated set containing one of the following values:

sec_acl_type_object
The ACL refers to the specified object.

sec_acl_type_default_object
The ACL is to be used when creating objects in the container.

sec_acl_type_default_container
The ACL is to be used when creating nested containers.

The following values are defined but not currently used. They are available for application programs that may create an application-specific ACL definition.

- **sec_acl_type_unspecified_3**
- **sec_acl_type_unspecified_4**
- **sec_acl_type_unspecified_5**
- **sec_acl_type_unspecified_6**
- **sec_acl_type_unspecified_7**

sec_acl_printstring_t
A **sec_acl_printstring_t** structure contains a printable representation for a permission in a **sec_acl_permset_t** permission set. This allows a generic ACL editing tool to be used for application-specific ACLs. The tool need not know the printable representation for each permission bit in a given permission set. The **sec_acl_get_printstring()** function will query an ACL manager for the print strings of the permissions it supports. The structure consists of three components:

printstring
A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.

helpstring
A character string of maximum length **sec_acl_printstring_help_len** containing some text that may be used to describe the specified permission.

permissions
A **sec_acl_permset_t** permission set describing the permissions that will be represented with the specified print string.

sec_acl_component_name_t
This type is a pointer to a character string, to be used to specify the entity a given ACL is protecting.

Constants

The following constants are used in **sec_acl_** calls:

sec_acl_default_handle
The value of an unbound ACL manager handle.

sec_rgy_acct_key_t
The following 32-bit integer constants are used with the **sec_rgy_acct_key_t** data type:

sec_rgy_acct_key_none
Invalid key.

sec_rgy_acct_key_person
The person name alone is enough.

sec_rgy_acct_key_group
The person and group names are both necessary for the account abbreviation.

sec_rgy_acct_key_org
The person, group, and organization names are all necessary.

sec_rgy_acct_key_last
Key values must be less than this constant.

sec_rgy_pname_t_size
The maximum number of characters in a **sec_rgy_pname_t**.

sec_acl_permset_t
The following constants are defined for use with the **sec_acl_permset_t** data type:

sec_acl_perm_read
The ACL allows read access to the protected object.

sec_acl_perm_write
The ACL allows write access to the protected object.

sec_acl_perm_execute
The ACL allows execute access to the protected object.

sec_acl_perm_owner
The ACL allows owner-level access to the protected object.

sec_acl_perm_insert
The ACL allows insert access to the protected object.

sec_acl_perm_delete
The ACL allows delete access to the protected object.

sec_acl_perm_test
The ACL allows access to the protected object only to the extent of being able to test for existence.

sec_acl_perm_unused_00000080 – sec_acl_perm_unused_0x80000000
The bits from 0x00000080 to 0x80000000 are not used by the conventional ACL permission set. Constants have been defined so application programs can easily use these bits for extended ACLs.

sec_acl_printstring_len
The maximum length of the printable representation of an ACL permission. (See **sec_acl_printstring_t**.)

sec_acl_printstring_help_len
The maximum length of a help message to be associated with a supported ACL permission. (See **sec_acl_printstring_t**.)

Files

/usr/include/dce/aclbase.idl
The **idl** file from which **aclbase.h** was derived.

Key Management API Data Types

Notes

Key management operations that take a keydata argument expect a pointer to a **sec_passwd_rec_t** structure, and those that take a keytype argument (**void ***) expect a pointer to a **sec_passwd_type_t**. Key management operations that yield a keydata argument as output set the pointer to an array of **sec_passwd_rec_t**. (The array is terminated by an element with a key type of **sec_passwd_none**.)

Operations that take a keydata argument expect a pointer to a **sec_passwd_rec_t** structure. Operations that yield a keydata argument as output set the pointer to an array of **sec_passwd_rec_t**. (The array is terminated by an element with key type **sec_passwd_none**.) Operations that take a keytype argument (**void ***) expect a pointer to a **sec_passwd_type_t**.

Synopsis

```
#include  <dce/keymgmt.h>
```

Data Types

sec_passwd_type_t
An enumerated set describing the currently supported key types. The possible values are as follows:

sec_passwd_none
Indicates no key types are supported.

sec_passwd_plain
Indicates that the key is a printable string of data.

sec_passwd_des
Indicates that the key is DES encrypted data.

sec_passwd_privkey
Indicates that the key is a private or public key of a public key pair used in public key authentication.

sec_passwd_genprivkey
Indicates the modulus bit size of the private key to be generated for a public key pair used in public key authentication.

sec_passwd_rec_t
A structure containing any of the following: a plaintext password, a preencrypted buffer of password data, a public-key-pair generation request, or a public or private key. The **sec_passwd_rec_t** structure consists of three components:

version_number
The version number of the password.

pepper
A character string combined with the password before an encryption key is derived from the password.

key A structure consists of the following components:

key_type
The key type can be the following:

sec_passwd_plain
Indicates that a printable string of data is stored in **plain**.

sec_passwd_des
Indicates that an array of data is stored in **des_key**.

sec_passwd_privkey
Indicates that X.509 ASN.1 DER-encoded data is stored in **priv_key**.

sec_passwd_genprivkey
Indicates that unsigned 32-bit data is stored in **modulus_size**.

tagged_union
A structure specifying the password. The value of the structure depends on **key_type**.

If **key_type** is **sec_passwd_plain**, the structure contains **plain**, a character string.

If **key_type** is **sec_passwd_des**, the structure contains **des_key**, a DES key of type **sec_passwd_des_key_t**.

If **key_type** is **sec_passwd_privkey**, the structure contains **priv_key**, a public or private key of type **sec_pk_data_t**.

If **key_type** is **sec_passwd_genprivkey**, the structure contains **modulus_size**, unsigned 32-bit data.

sec_passwd_version_t
An unsigned 32-bit integer that defines the password version number. You can supply a version number or a 0 for no version number. If you supply the constant **sec_passwd_c_version_none**, the security service supplies a system-generated version number.

sec_key_mgmt_authn_service
A 32-bit unsigned integer whose purpose is to indicate the authentication service in use, since a server may have different keys for different levels of security. The possible values of this data type and their meanings are as follows:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_private
DCE private key authentication (an implementation of the Kerberos system).

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

Constants

There are no constants specially defined for use with the key management API.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **keymgmt.h** was derived.

ID Mapping API Data Types

Synopsis

```
#include  <dce/secidmap.h>
```

Data Types

No special data types are defined for the ID mapping API.

Constants

No special constants are defined for the ID mapping API.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **secidmap.h** was derived.

Password Management API Data Types

Synopsis

```
#include  <dce/sec_pwd_mgmt.h>
```

Data Types

The following data types are used in **sec_pwd_mgmt_** calls:

sec_passwd_mgmt_handle_t
: A pointer to an opaque handle consisting of password management information about a principal. It is returned by **sec_pwd_mgmt_setup()**.

Constants

There are no constants specially defined for use with the password management API.

Files

/usr/include/dce/sec_pwd_mgmt.idl
: The **idl** file from which **sec_pwd_mgmt.h** was derived.

Public Key API Data Types

Synopsis

```
#include  <dce/sec_pk.h>
```

Data Types

The following data types are used in **sec_pk** calls:

sec_pk_domain_t
A UUID of type **uuid_t** associated with the application domain in which a public or private key is used.

sec_pk_usage_flags_t
A 32k-bit set of key-usage flags that describe the use of a key or key pair. The flags are:

sec_pk_usage_digitalSignature

sec_pk_usage_nonRepudiation

sec_pk_usage_keyEncipherment

sec_pk_usage_keyAgreement

sec_pk_usage_keyCertSign

sec_pk_usage_offLineCRLSign

These flags are described in the X.509 (1993E) AM 1 standard.

sec_pk_data_t
A structure that points to an X.509 or X.511 ASN.1 DER-encoded value. The **sec_pk_data_t** data type acts as a base for the following data types, which are aliases for **sec_pk_data_t**:

sec_pk_gen_data_t

sec_pk_pubkey_t

sec_pk_pvtkey_t

sec_pk_signed_t

sec_pk_encrypted_t

sec_pk_algorithm_id_t

The alias data types indicate the specific information pointed to by **sec_pk_data_t**. Instead of using **sec_pk_data_t** directly, use the alias data types.

The **sec_pk_data_t** data type consists of the following elements:

len The size of **data**.

data A pointer to a character string.

sec_pk_gen_data_t
A structure that acts as an alias to a **sec_pk_data_t** that contains plain ASCII data.

sec_pk_pubkey_t
A structure that acts as an alias to a **sec_pk_data_t** that contains an X.509 ASN.1 DER-encoded value of type **SubjectPublicKeyInfo**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_pvtkey_t
A structure that contains an X.509 ASN.1 DER-encoded private key value. The key format depends on the public key infrastructure.

This data type assumes that the public key infrastructure provides functions for generating a private key in this format.

sec_pk_signed_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **SIGNED**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_encrypted_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **ENCRYPTED**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

sec_pk_algorithm_id_t
A structure that contains an X.509 ASN.1 DER-encoded value of type **AlgorithmIdentifier**. This data type assumes that the public key infrastructure provides functions for generating a public key in this format.

Constants

The following constants are used in **sec_pk** calls:

The following unsigned 32-bit constants, which are used with the **sec_pk_usage_flags_t** data type, correspond to **KeyUsage** types defined in DAM 1 (Dec 1995) to X.509 (1993):

sec_pk_usage_digitalSignature

sec_pk_usage_nonRepudiation

sec_pk_usage_keyEncipherment

sec_pk_usage_dataEncipherment

sec_pk_usage_keyAgreement

sec_pk_usage_keyCertSign

sec_pk_usage_offLineCRLSign

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **sec_pk.h** was derived.

audit_intro

Purpose

Introduction to the DCE audit API runtime

Description

This introduction gives general information about the DCE audit application programming interface (API) and an overview of the following parts of the DCE audit API runtime:

- Runtime services
- Environment variables
- Data types and structures
- Permissions required

Runtime Services

The following is an alphabetical list of the audit API routines. With each routine name is its description. The types of application program that will most likely call the routine are enclosed in parentheses.

dce_aud_close()
Closes an audit trail (client/server applications, audit trail analysis and examination tools).

dce_aud_commit()
Performs the audit action(s) (client/server applications).

dce_aud_discard()
Discards an audit record which releases the memory (client/server applications, audit trail analysis and examination tools).

dce_aud_free_ev_info()
Frees the memory allocated for an event information structure returned from calling the **dce_aud_get_ev_info()** function (audit trail analysis and examination tools).

dce_aud_free_header()
Frees the memory allocated to a designated audit record header structure (audit trail analysis and examination tools).

dce_aud_get_ev_info()
Gets the event-specific information of a specified audit record (audit trail analysis and examination tools).

dce_aud_get_header()
Gets the header of a specified audit record (audit trail analysis and examination tools).

dce_aud_length()
Gets the length of a specified audit record (client/server applications, audit trail analysis and examination tools).

dce_aud_next()
Reads the next audit record from a specified audit trail into a buffer (audit trail analysis and examination tools).

dce_aud_open()
Opens a specified audit trail for read or write (client/server applications, audit trail analysis and examination tools).

dce_aud_print()
Formats an audit record into a human-readable form (audit trail analysis and examination tools).

dce_aud_put_ev_info()
Adds event-specific information to a specified audit record buffer (client/server applications).

dce_aud_set_trail_size_limit()
Sets a limit to the audit trail size (client/server applications).

dce_aud_start()
Determines whether a specified event should be audited given the client's binding information and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_name()
Determines whether a specified event should be audited given the client/server name and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_pac()
Determines whether a specified event should be audited given the client's privilege attribute certificate (PAC) and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_server_binding()
Determines whether a specified event should be audited given the server's binding information and the event outcome. If the event should be audited or if it is not yet known whether the event should be audited because the event outcome is still unknown, memory for the audit record descriptor is allocated and the address of this memory is returned to the caller (client/server applications).

dce_aud_start_with_uuid()
Determines whether a specified event should be audited given the client/server UUID and the event outcome. If the event must be audited, or if the outcome of the event is not yet known, the memory for the audit record descriptor is allocated and the address of this structure is returned to the caller (client/server applications).

Audit Data Types

The following subsections list the data types and structures used by applications to perform auditing and to analyze audit trails.

Event-Specific Information
The audit APIs allow applications to include event-specific information in

audit records. Event-specific information must be represented as information items using the following data type.

```
typedef struct {
        unsigned16 format;
        union {
            idl_small_int small_int;
            idl_short_int short_int;
            idl_long_int long_int;
            idl_hyper_int hyper_int;
            idl_usmall_int usmall_int;
            idl_ushort_int ushort_int;
            idl_ulong_int ulong_int;
            idl_uhyper_int uhyper_int;
            idl_short_float short_float;
            idl_long_float long_float;
            idl_boolean boolean;
            uuid_t uuid;
            utc_t utc;
            sec_acl_t * acl;
            idl_byte * byte_string;
            idl_char * char_string;
        } data;
} dce_aud_ev_info_t;
```

The *format* field of the above data structure defines formatting information that is used to determine the type of the data referenced by the *data* field. The following table shows possible values of the *format* field, their corresponding data types, and their sizes.

Table 34. Event Data Format Specifiers—intro(3sec)

Specifier	Data Type	Size
aud_c_evt_info_small_int	**idl_small_int**	1 byte
aud_c_evt_info_short_int	**idl_short_int**	2 bytes
aud_c_evt_info_long_int	**idl_long_int**	4 bytes
aud_c_evt_info_hyper_int	**idl_hyper_int**	8 bytes
aud_c_evt_info_usmall_int	**idl_usmall_int**	1 bytes
aud_c_evt_info_ushort_int	**idl_ushort_int**	2 bytes
aud_c_evt_info_ulong_int	**idl_ulong_int**	4 bytes
aud_c_evt_info_uhyper_int	**idl_uhyper_int**	8 bytes
aud_c_evt_info_short_float	**idl_short_float**	4 bytes
aud_c_evt_info_long_float	**idl_long_float**	8 bytes
aud_c_evt_info_boolean	**idl_boolean**	1 byte
aud_c_evt_info_uuid	**uuid_t**	16 bytes
aud_c_evt_info_utc	**utc_t**	16 bytes
aud_c_evt_info_acl	**sec_acl_t ***	variable size
aud_c_evt_info_byte_string	**idl_byte ***	variable size
aud_c_evt_info_char_string	**idl_char ***	variable size

Byte strings and character strings are terminated with a 0 (zero) byte. New data types can be added to this list if they are used frequently. Servers could use the pickling service of the IDL compiler to encode complex data types into byte strings that are to be included in an audit record.

Audit Record Header Data Structure

The following data structure is used to store header information obtained

from an audit record. This structure is normally only used by audit trail analysis and examination tools. That is, it is hidden from client/server applications.

```
typedef struct {
        unsigned32      format;
        uuid_t          server;
        unsigned32      event;
        unsigned16      outcome;
        unsigned16      authz_st;
        uuid_t client;
        uuid_t cell;
        unsigned16      num_groups;
        utc_t time;
        char *addr;
        uuid_t *groups;
} dce_aud_hdr_t;
```

format
Contains the version number of the tail format of the event used for the event-specific information. With this format version number, the audit analysis tools can accommodate changes in the formats of the event-specific information. For example, the event-specific information of an event may initially be defined to be a 32-bit integer, and later changed to a character string. Format version 0 (zero) is assigned to the initial format for each event.

server Contains the UUID of the server that generates the audit record.

event Contains the event number.

outcome
Indicates whether the event failed or succeeded. If the event failed, the reason for the failure is given.

authz_st
Indicates how the client is authorized: by a name or by a DCE privilege attribute certificate (PAC).

client Contains the UUID of the client.

cell Contains the UUID of the client's cell.

num_groups
Contains the number of local group privileges the client used for access.

groups
Contains the UUIDs of the local group privileges that are used by the client for the access. By default, the group information is not be included in the header (num_groups is set to 0 in this case), to minimize the size of the audit records. If the group information is deemed as important, it can be included.

Information about foreign groups (global groups that do not belong to the same cell where the client is registered) is not included in this version of audit header but may be included in later versions when global groups are supported.

time Contains a timestamp of **utc_t** type that records the time when the server committed the audit record (that is, after providing the event information through audit API function calls). Recording this time, rather than recording the time when the audit record is appended to an audit trail, will better maintain the sequence of events. The implementation of the audit subsystem may involve communication

between the server and a remote audit daemon, incurring indefinite delays by network problems or intruders. The inaccuracy in the **utc_t** timestamp may be useful for correlating events. When searching for events in an audit trail that occur within a time interval, if the results of the comparisons between the time of an event and the interval's starting and ending times is **maybe** (because of inaccuracies), then the event should be returned.

addr Records the client's address (port address of the caller). Port addresses are not authenticated. A caller can provide a fraudulent port address to a DCE server. However, if this unauthenticated port address is deemed to be useful information, a DCE server can record this information using this field.

The identity of the server cell is not recorded in the header, because of the assumption that all audit records in an audit trail are for servers within a single cell, and implicitly, the server cell is the local cell.

Audit Record Descriptor

An opaque data type, **dce_aud_rec_t**, is used to represent an audit record descriptor. An audit record descriptor may be created, manipulated, or disposed of by the following functions: The functions **dce_aud_start()**, **dce_aud_start_with_pac()**, **dce_aud_start_with_name()**, **dce_aud_start_with_server_binding()**, and **dce_aud_next()** return a record descriptor. The function **dce_aud_put_ev_info()** adds event information to an audit record through a record descriptor. The functions **dce_aud_get_header()**, **dce_aud_get_ev_info()**, and **dce_aud_length()** get the event and record information through a record descriptor. The function **dce_aud_commit()** commits an audit record through its descriptor. The function **dce_aud_discard()** disposes of a record descriptor. The function **dce_aud_discard()** is necessary only after reading the record (that is, after invoking **dce_aud_next()**.

Audit Trail Descriptor

An opaque data type, **dce_aud_trail_t**, is used to represent an audit trail descriptor. The **dce_aud_open()** function opens an audit trail and returns a trail descriptor; **dce_aud_next()** obtains an audit record from this descriptor; and **dce_aud_commit()** commits an audit record from and to an opened audit trail through this descriptor. The **dce_aud_close()** function disposes of this descriptor.

Environment Variables

The audit API routines use the following environment variables:

DCEAUDITOFF

If this environment variable is defined at the time the application is started, auditing is turned off.

DCEAUDITFILTERON

If this environment variable is defined, filtering is enabled.

DCEAUDITTRAILSIZE

Sets the limit of the audit trail size. This variable overrides the limit set by the **dce_aud_set_trail_size_limit()** function.

Permissions Required

To use an audit daemon's audit record logging service, you need the log (**l**) permission to the audit daemon.

Related Information

Books: *OSF DCE Administration Commands Reference*, *OSF DCE Application Development Guide*.

pkc_intro

Purpose

Introduction to trust list facilities API

Description

This reference page describes the data types used by the trust list facility.

Overview of the Facility

Retrieving keys using this API is a three step process.

The first step involves creating a **pkc** structure called a trust list, which reflects the caller's initial trust. A trust list is a list of {name, key} pairs or certificates that are trusted *a priori*.

An empty trust list is created through a call to the routine **pkc_init_trustlist(3sec)**, and entries are inserted into a trust list by a call to **pkc_append_to_trustlist(3sec)**.

Once the trust list is complete, the application should next call **pkc_init_trustbase(3sec)**. This routine takes the trust list and processes it to produce a stucture called a trust base, which reflects any transitive trust, independent of the name of any desired target.

Creation of the trust base (and the prerequisite trust list) is expected to be performed at application startup, although it can be done any time prior to key retrieval. All processing up to this point is independent of the name(s) of principals whose keys are to be retrieved, and the trust base may be used for multiple key retrieval operations.

Once a trust base has been obtained, it may be used for key retrieval. Keys are retrieved for a given target principal using the **pkc_retrieve_keys(3sec)** routine, which takes a trust base and a name and returns an array of keys.

Data Structures

The following data structures are used by the trust list facilities.

- The **trust_type_t** type consists of an enumeration of the different possible varieties of trust:
  - **UNTRUSTED**

    No trust (e.g., unauthenticated).
  - **DIRECT_TRUST**

    Direct trust via third party (e.g., authenticated registry).
  - **CERTIFIED_TRUST**

    Trust certified by caller's trust base.
- The **certification_flags_t** structure describes the trust that can be placed in a returned key. It contains the following fields:
  - **trust_type**

    A **trust_type_t** value expressing the style of trust.
  - **missing_crls**

A **char**; its value is TRUE (not 0) if one or more CRLs are missing.

– **revoked**

A **char** whose value is TRUE (not 0) if any certificate has been revoked (even if it was still valid at the retrieval time).

- The **cert_t** structure contains the following fields:
  – **version**

  An **int** whose value must be 0.

  – **cert**

  A pointer to an **unsigned char** representing the ASN.1 encoding of a certificate.

  – **size**

  A **size_t** which represents the size of the encoding.
- The **trusted_key_t** structure contains the following fields:
  – **version**

  An **int** whose value must be 0.

  – **ca**

  A pointer to an **unsigned char** (**x500 char**) string which represents the name of the Certification Authority whose key this is. For example, **/.../foo_cell/ca** or **/.../C=US/O=dec/CN=foo_cell/ca**.

  – **key**

  A pointer to an **unsigned char** representing the Certification Authority's ASN.1 key.

  – **size**

  A **size_t** representing the size of the CA's ASN.1 key.

  – **startDate**

  An **utc_t** representing the time at which the key begins to be valid.

  – **endDate**

  An **utc_t** representing the time at which the key ceases to be valid.
- The **trustitem_t** structure holds either a key, or a certificate. It has the following fields:
  – **type**

  An **int** whose value specifies either that the structure holds a key (**IS_KEY**) or a certificate (**IS_CERT**).

  – Depending on the value of **type**, the structure additionally contains a **trusted_key_t** (if **IS_KEY**) or a **cert_t** (if **IS_CERT**).
- The **selection_t** structure is defined for future enhancements that will enable users to specify usages for the key being retrieved. However, its contents are currently ignored.

Related Information

Functions: **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

crypto_intro

Purpose

Introduction to the signature algorithm API registration facility

Description

This reference page describes the data types used by the signature algorithm (or ″cryptographic″) module registration API.

Accessing and Using Cryptographic Modules

Cryptographic implementations (also known as ″algorithms″) are identified by OIDs (object identifiers).

Policy implementors are recommended to access cryptographic modules mainly through the following routines, which perform all locking necessary to make the calls thread safe, and also transparently handle any context information that a given cryptographic implementation may need.

- **pkc_crypto_get_registered_algorithms(3sec)**

  Call this routine to get an OID set describing the currently registered algorithm implementations.
- **pkc_crypto_sign(3sec)**

  Call this routine to get data signed.
- **pkc_crypto_verify_signature(3sec)**

  Call this routine to verify signed data.
- **pkc_crypto_generate_keypair(3sec)**

  Call this routine to generate a pair of public/private keys.

Information about a cryptographic module may be obtained by calling **pkc_crypto_lookup_algorithm(3sec)**.

Data can also be signed and verified by looking up the desired algorithm (with **pkc_crypto_lookup_algorithm(3sec)**) and then explicitly calling the module's **(sign)()** or **verify()** routine, although in this case the calling application must take care to avoid multi-threading problems, and is also responsible for opening the crypto module prior to use, and closing it afterwards.

Implementing Cryptographic Modules

Every cryptographic module must export a **pkc_signature_algorithm_t** object.

The **pkc_signature_algorithm_t** data type is used to register a new cryptographic module with the certification API. It fully describes a specific implemented cryptographic algorithm, and provides entry points to its **sign()** and **verify()** functions. It is defined as follows:

```
typedef struct {
OM_uint32 version;
  gss_OID_desc alg_id>;
  pkc_alg_flags_t flags;
  char reserved[32 - sizeof(pkc_alg_flags_t)];
  char * (* name)(void);
```

```
unsigned32 (*open) (void** context);
unsigned32 (*close) (void** context);
unsigned32 (*verify) (void ** context,
  sec_pk_gen_data_t * data,
  sec_pk_data_t * public_key,
  sec_pk_data_t * signature);
unsigned32 (*sign) (void ** context,
  sec_pk_gen_data_t * data,
  sec_pk_data_t * private_key,
  sec_pk_data_t * signature);
unsigned32 (*generate_keypair) (void ** context,
  unsigned32 size,
  void * alg_info,
  sec_pk_data_t * private_key,
  sec_pk_data_t * public_key);} pkc_signature_algorithm_t;
```

The **(name)()**, **(open)()**, **(close)()**, **(verify)()**, **(sign)()** and **(generate_keypair)()** routines must be implemented by the application implementing the algorithm and registered by calling the **pkc_crypto_register_signature_alg(3sec)** routine. Note, however, that all the routines except for **(verify)()** and **(name)()** are optional. Explanations of all the fields in **pkc_signature_algorithm_t** are contained in the following subsections.

Cryptographic Module Data Fields

The structure contains the following data fields:

version
: Identifies the version of the certification API for which the module is implemented. The value of this field is always **pkc_V1** for DCE 1.2.

alg_id
: An object identifier that identifies the algorithm; the OID that appears in certificates signed by the algorithm.

flags
: Describes whether the module's **(sign)()** and **(verify)()** functions are threadsafe, and whether the module supports simultaneous crypto sessions.

The **version** and **alg_id** fields are required for all versions of this data structure. Other fields may be version dependent.

Cryptographic Module Functions

NULL may be supplied as the address of the **(open)()**, **(close)()**, **(sign)()**, or **(generate_keypair)()** routines, if the cryptographic module does not provide or require the corresponding feature; the presence of these functions in a cryptographic module is optional. However, all cryptographic modules must provide **(verify)()** and **(name)()** functions.

Algorithm Flags Data Type

The **pkc_alg_flags_t** data type is used to record various information about a cryptographic module. It is defined as follows:

```
typedef struct {
char threadsafe;
char multi_session;}
pkc_alg_flags_t;
```

The structure contains two fields which have the following meanings:

threadsafe
Has a non-zero (TRUE) value if the module's **(sign)()** and **(verify)()** routines may be safely called simultaneously (within a single crypto session) by multiple threads.

multi_session
Has a non-zero (TRUE) value if the module implementation supports multiple simultaneous crypto sessions.

(name)()

Purpose

(name)() - Returns the algorithm name as a string for use in diagnostic or auditing messages

Synopsis

```
char * (* name)(void);
```

Description

The name should be returned in storage allocated using the **pkc_alloc()** function defined in **pkc_base.h**. Note that this is the only cryptographic module routine that may be called without first calling the **(open)()** routine.

This routine is mandatory.

(open)(), (close)()

Purpose

(open)() - Opens and initializes the cryptographic module

(close)() - Closes the cryptographic module

Both routines are optional.

Synopsis

unsigned32 (*open) (void_context_);**

unsigned32 (*close) (void_context_);**

Parameters

Output

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

Description

Before invoking any of the module's encryption routines (e.g., **(sign)()** or **(verify)()**, the certification API will invoke the module's **(open)()** function. Once the module's **(close)()** routine has been invoked, the certification facility will invoke **(open)()** again before making any further calls to the module.

Both the **(open)()** and the **(close)()** routines require only one argument, *context*. If the cryptographic module requires state information to be maintained between calls, it may use the *context* parameter to do this. The information is initialized by the **(open)()** routine and returned as an opaque object to the caller, who then passes the parameter to subsequent **(sign)()**, **(verify)()**, **(generate_keypair)()**, or **(close)()** calls.

Note that if the **(open)()** routine stores any state in the *context* parameter, the **(close)()** routine should free this storage.

(sign)()

Purpose

(sign)() - Calculates a signature over the supplied data using the specified key

Synopsis

```
unsigned32 (*sign) (void ** context,
    sec_pk_gen_data_t * data,
    sec_pk_data_t * private_key,
    sec_pk_data_t **signature);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

data The certificate data that is to be signed.

private_key
Key to use to generate the signature, provided as a BER-encoded **PrivateKeyInfo** object, as defined in PKCS#8, as appropriate for the algorithm.

Output

signature
The signature generated on the data passed. Storage allocation should be performed by calling the **pkc_alloc()** and **pkc_free()** functions defined in **pkc_base.h**.

Description

The **(sign)()** routine calculates a signature over the supplied data, using the specified key. The *private_key* parameter will be a BER-encoded **PrivateKeyInfo** data object. The *signature* should be returned by the function; storage allocation should be performed by calling the **pkc_alloc()** and **pkc_free()** functions defined in **pkc_base.h**.

This routine is optional.

(verify)()

Purpose

(verify)() - Checks the supplied signature against the supplied data, thus verifying the certificate in which the data and the signature appear

Synopsis

```
unsigned32 (*verify) (void ** context,
    sec_pk_gen_data_t * data,
    sec_pk_data_t * public_key,
    sec_pk_data_t *signature);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

data The entire **certificateInfo**.

public_key
The public key to use on the signature.

signature
The signature to be verified.

Description

The **(verify)()** routine checks the supplied signature against the supplied data. *public_key* is a **SubjectPublicKeyInfo** data structure, encoded in BER, as found within an X.509 certificate.

The routine should return 0 for a correct signature, **pkc_invalid_signature** for an incorrect signature, or another DCE-defined error status to indicate any other errors.

This routine must be implemented in any cryptographic module.

(generate_keypair)()

Purpose

(generate_keypair)() - Generates a pair of public and private keys

Synopsis

```
unsigned32  (*generate_keypair)
(void  **  context,
    unsigned32  size,
    void  *alg_info,
    sec_pk_data_t  *  private_key,
    sec_pk_data_t  *public_key);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

size Specifies the key size.

alg_info
Specifies the crypto module.

Output

private_key
The generated private key.

public_key
The generated public key.

Description

The **(generate_keypair)()** routine generates a pair of private and public keys. The *size* parameter should be used by the routine to determine the key size in some way (for the RSA algorithm, for example, it should be treated as the number of bits in the key modulus). The *private_key* and *public_key* parameters should return BER-encoded **PrivateKeyInfo** and **SubjectPublicKeyInfo** data objects respectively. The *alg_info* parameter can be used for algorithm-specific information to modify the key generation process. However, all crypto modules that offer this function should be prepared to operate when **NULL** is supplied for this parameter.

This routine is optional.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

policy_intro

Purpose

Introduction to the policy module registration and service facility

Description

This reference page describes the data types used by the policy module registration and service API.

The routines documented here are intended for the use of policy implementors. Regular users invoke a policy via the high-level API (e.g., **pkc_retrieve_keyinfo(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, etc.) which calls the routines documented below internally.

Accessing Policy Modules

Policy modules are identified by OIDs (object identifiers). A policy module is accessed by passing its identifying OID to **pkc_plcy_lookup_policy(3sec)**.

There are two ways of retrieving a key: either by looking up the desired policy module and then explicitly calling its **(retrieve_keyinfo)()** routine; or by simply calling the **pkc_plcy_retrieve_keyinfo(3sec)** routine, identifying the desired policy by means of an OID passed directly to the call. The latter method, in which the operation is performed in one step, is the recommended one.

Policy Flags Data Type

The **pkc_plcy_flags_t** data type is used to record various information about a policy module. It is defined as follows:

```
typedef struct {char threadsafe; char multi_session;} pkc_plcy_flags_t;
```

The structure contains two fields which have the following meanings:

threadsafe
: Has a non-zero (TRUE) value if the policy's **retrieve_keyinfo()** function may be safely called simultaneously (within a single policy session) by multiple threads.

multi_session
: Has a non-zero (TRUE) value if the policy implementation supports multiple simultaneous policy sessions.

Policy Module Data Type

The **pkc_policy_t** data type is used to register a new policy module with the certification API. It fully describes a policy module's functionality, and provides entry points to its key retrieval functions. It is defined as follows:

```
typedef struct {
OM_uint32 version;
gss_OID_desc policy_id;
pkc_plcy_flags_t flags;
char reserved[32 - sizeof(pkc_plcy_flags_t)];
char * (* name) (void);
unsigned32 (*open) (void** context);
```

```
unsigned32 (*close) (void** context);
unsigned32 (*establish_trustbase) (void ** context,
     const pkc_trust_list_t & initial_trust,
     const utc_t * date,
     pkc_usage_t desired_usage,
     char initial_explicit_policy_required,
     pkc_trust_list_t & out_trust);


  unsigned32 (*retrieve_keyinfo) (void **context,
     const pkc_trust_list_t &trust,
     const x500name &subjectName,
     const utc_t * date,
     const uuid_t & domain,
     pkc_key_usage_t desired_usage,
     char initial_explicit_policy_required,
     pkc_key_information_t &key);
  unsigned32 (*delete_trustbase) (void **context,
     void ** trust_base_handle);
  unsigned32 (*delete_keyinfo) (void ** context,
     void ** keys_handle);
  unsigned32 (*get_key_count) (void ** context,
     void * keys_handle,
     size_t * key_count);
  unsigned32 (*get_key_data) (void ** context,
     void * keys_handle,
     unsigned key_index,
     unsigned char ** key_data,
     size_t * key_length);
  unsigned32 (*get_key_trust) (void ** context,
     void * keys_handle,
     unsigned key_index,
     certification_flags_t * flags uuid_t * domain,
     pkc_generic_key_usage_t * usages);
  unsigned32 (*get_key_certifier_count) (void **context,
     void * keys_handle,
     unsigned key_index,
     size_t * ca_count);
  unsigned32 (*get_key_certifier_info) (void **context,
     void * keys_handle,
     unsigned key_index,
     unsigned ca_index,
     char ** ca_name,
     utc_t * certification_start,
     utc_t * certification_expiration,
     char  * is_crl_valid,
     utc_t * last_crl_seen,
     utc_t * next_crl_expected);
  } pkc_policy_t;
```

The **(name)()**, **(open)()**, **(close)()**, **(establish_trustbase()**, **(*get_key_count)()**, **(*get_key_data)()**, **(*get_key_trust)()**, **(*get_key_certifier_count)()**, **(*get_key_certifier_info)()**, and **(*retrieve_keyinfo)()** routines must be implemented by the application implementing the module and registered using the **pkc_register_policy(3sec)** routine. Note, however, that only **(*retrieve_keyinfo)()**, **(*get_key_count)()**, **(*get_key_certifier_count)()** and **(*get_key_data)()** are required. Explanations of all the fields in **pkc_policy_t** are contained in the following subsections.

Policy Module Data Fields

The structure contains the following data fields:

version
Identifies the version of the certification API for which the module is implemented. The value of this field is always **pkc_V1** for DCE 1.2.

policy_id
An object identifier that identifies the policy.

flags Describes whether the module's key retrieval function is threadsafe, and whether the module supports simultaneous policy sessions.

The **version** and *alg_id* fields are required for all versions of this data structure. Other fields may be version dependent.

Policy Module Functions

NULL may be supplied as the address of the **(name)()**, **(open)()**, **(establish_trustbase)()** or **(close)()** routines, if the policy module does not provide or require the corresponding feature; the presence of these functions in a policy module is optional. However, all policy modules must provide **(*retrieve_keyinfo)()**, **(*get_key_count)()**, **(*get_key_certifier_count)()** and **(*get_key_data)()** functions.

(name)()

Purpose

(name)() — Returns the policy name as a string, suitable for use in diagnostic or auditing messages

This routine is optional.

Synopsis

```
char * (* name) (void);
```

Description

The name should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**. The caller of this routine is expected to invoke **pkc_free(3sec)**to release the storage once the name is no longer required.

Note that this is the only policy module routine that may be called without first calling the **(open)()** routine.

(open)(), (close)()

Purpose

(open)() — Opens and initializes the policy module

(close)() — Closes the policy module

Both these routines are optional.

Synopsis

unsigned32 (*open) (void_context_);**

unsigned32 (*close) (void_context_);**

Parameters

Output

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

Description

Before invoking any policy routines (e.g., **(retrieve_keyinfo)()**), the certification API will invoke the module's **(open)()** function. Once the module's **(close)()** routine has been invoked, the certification facility will invoke **(open)()** again before making any further calls to the module.

Both the **(open)()** and the **(close)()** routines require only one argument, *context*. If the policy module requires state information to be maintained between calls, it may use the *context* parameter to do this. The information is initialized by the **(open)()** routine and returned as an opaque object to the caller, who then passes the parameter to subsequent **(retrieve_keyinfo)()**, **(establish_trustbase)()**, or **(close)()** calls.

Note that if the **(open)()** routine stores any state in the *context* parameter, the **(close)()** routine should free this storage.

(establish_trustbase)()

Purpose

(establish_trustbase)() — Initializes a trust base

Synopsis

```
unsigned32 (*establish_trustbase)
(void ** context,
    const pkc_trust_list_t & initial_trust, const utc_t * date,
    char initial_explicit_policy_required,
    pkc_trust_list_t &out_trust);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

initial_trust
Specifies the caller's initial trust.

date Specifies time for which information is to be returned.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

out_trust
An extended trust list.

Description

This is a one-time call made by an application to initialize a trust base. It returns the *out_trust* parameter, which contains an extended trust list. After this call is made, the application can call **(retrieve_keyinfo)()** to obtain the public keys of any particular principal. If the trust base does not change, **(retrieve_keyinfo)()** can be used to look up another principal's public key without incurring the cost of another call to **(establish_trustbase)()**. A trust base will not change unless the *initial_trust* list changes.

(*delete_trustbase)()

Purpose

(*delete_trustbase)() — Frees storage allocated for a trust base

This routine is optional.

Synopsis

```
unsigned32  (*delete_trustbase)
(void  **  context,
    void  **trust_base_handle);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

(*delete_keyinfo)()

Purpose

(*delete_keyinfo)() — Frees storage allocated for key information

This routine is optional.

Synopsis

```
unsigned32  (*delete_keyinfo)
(void  **  context,
    void  **  keys_handle);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

Description

(*delete_keyinfo)() frees storage that was allocated for key information.

(*get_key_count)()

Purpose

(*get_key_count)() — Returns number of keys

This routine is optional.

Synopsis

```
unsigned32  (*get_key_count)
(void  **  context,
    void  *  keys_handle,
    size_t  *  key_count);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller.

Output

key_count
Number of keys for the principal.

Description

(*get_key_count)() returns the number of keys for the principal. This value is determined by reference to the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

(*get_key_data)()

Purpose

(*get_key_data)() — Returns a public key

This routine is optional.

Synopsis

```
unsigned32 (*get_key_data) (void
** context,
    void * keys_handle,
    unsigned key_index,
    unsigned char ** key_data,
    size_t * key_length);
```

Parameters

Input

context
: An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
: A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index
: Index (ranging from 0 to *key_count* − 1) of the key desired.

Output

key_data
: The encoded public key.

key_length
: Length of the key data returned.

Description

(*get_key_data)() returns the public key specified by *index*. The *key_data* returned is extracted from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

key_data should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**.

(*get_key_trust)()

Purpose

(*get_key_trust)() — Returns information about key trust

This routine is optional.

Synopsis

```
unsigned32  (*get_key_trust)
(void  **  context,
    void  *  keys_handle,
    unsigned  key_index,
    certification_flags_t  *  flagsuuid_t  *  domain,
    pkc_generic_key_usage_t  *usages);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index
Index (ranging from 0 to *key_count* −1) of the key desired.

Output

flags Information about the trust that can be placed in the key (see below).

domain
Indicates domain of retrieved key. A value of **sec_pk_domain_unspecified** or **NULL** means that the policy does not distinguish keys by domain.

usages
Indicates usage key is intended for.

Description

(*get_key_trust)() returns information about the trust reposed in the key specified by *index*. This information is determined by reference to the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

The returned **certification_flags_t** structure describes the trust that can be placed in the key. It contains the following fields:

- **trust_type**

  A **trust_type_t** value, which will be one of the following:

  – **UNTRUSTED**

    No trust (e.g., unauthenticated).

  - **DIRECT_TRUST**

    Direct trust via third party (e.g., authenticated registry).
  - **CERTIFIED_TRUST**

    Trust certified by caller's trust base.
- **missing_crls**

  A **char** whose value is TRUE (not 0) if one or more CRLs are missing.
- **revoked**

  A **char** whose value is TRUE (not 0) if any certificate has been revoked (even if it was still valid at the retrieval time).

If **domain** and **usages** are passed as non-**NULL** pointers, upon successful return these parameters describe the domain and permitted usages of the specified key. Policies that do not distinguish keys according to domain indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage indicate all usages are permitted.

The returned **usages** is a bit mask which describes the usages, if any, which the key is restricted to. The value is formed by AND-ing together one or more of the following constants:

PKC_KEY_USAGE_AUTHENTICATION
: The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
: The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
: The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
: The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
: The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
: The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
: The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
: The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
: The key can be used to sign transactions

A returned **usages** value of **NULL** (or a value with all bits set) means that the key is suitable for any usage.

(*get_key_certifier_count)()

Purpose

(*get_key_certifier_count)() — Returns number of key's certifying authorities

This routine is optional.

Synopsis

```
unsigned32  (*get_key_certifier_count)
(void  **context,
    void  *  keys_handle,
    unsigned  key_index,
    size_t  *  ca_count);
```

Parameters

Input

context
: An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
: A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index
: Index (ranging from 0 to *key_count* − 1) of the key desired.

Output

ca_count
: Number of certifying authorities for the key.

Description

(*get_key_certifier_count)() returns the number of certifying authorities for the key specified by *index*. This information is determined from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

(*get_key_certifier_info)()

Purpose

(*get_key_certifier_info)() — Returns information about a certifying authority

This routine is optional.

Synopsis

```
unsigned32  (*get_key_certifier_info)
(void  **context,
    void  *  keys_handle,
    unsigned  key_index,
    unsigned  ca_index,
    char  **  ca_name,
    utc_t  *  certification_start,
    utc_t  *  certification_expiration,
    char    *  is_crl_valid,
    utc_t  *  last_crl_seen,
    utc_t  *next_crl_expected);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

keys_handle
A policy specific structure, contained in the **keyinfo_t** structure passed by the original caller (see **pkc_intro(3sec)).**

key_index
Index (ranging from 0 to *key_count* − 1) of the key desired.

ca_index
Index of the certifier about whom information is desired.

Output

ca_name
The name of the certifier.

certification_start
Time at which certification by this certifier starts.

certification_expiration
Time at which certification by this certifier ends.

is_crl_valid
If TRUE, there is a certificate revocation list for this certifier.

last_crl_seen
Time at which certificate revocation list was last seen.

next_crl_expected
Time at which next certificate revocation list is expected.

Description

(*get_key_certifier_info)() returns information about the certifying authority specified by *ca_index* for the key specified by *key_index*.

The desired information is extracted by the routine from the policy-specific structure pointed to by *keys_handle*, a field in the **keyinfo_t** structure passed by the original caller.

Note that any of the return parameters may be passed as NULL if the corresponding information is not required.

The *certifier_name* parameter should be returned in storage allocated using the **pkc_malloc()** function defined in **pkc_common.h**.

(retrieve_keyinfo)()

Purpose

(retrieve_keyinfo)() — Returns the public key for the specified principal

Synopsis

```
unsigned32 (*retrieve_keyinfo)
(void ** context,
    const void * trust_base_handle,
    const x500name & subjectName,
    const utc_t * date,
    const uuid_t & domain,
    pkc_key_usage_t desired_usage,
    char initial_explicit_policy_required,
    void ** keys_handle);
```

Parameters

Input

context
An opaque (to the caller) data structure containing any state information required by the module across calls.

trust_base_handle
Specifies trust base.

subjectName
Specifies the desired subject name.

date Specifies time for which information is to be returned.

domain
Specifies the particular domain to which the key-search operation should be restricted. Specify **sec_pk_domain_unspecified** or **NULL** to indicate that keys for any domain should be retrieved.

desired_usage
Specifies the one or more specific usages to which the key-search operation should be restricted.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

keys_handle
The handle to the public key for the specified target principal.

Description

The **(retrieve_keyinfo)()** routine reads the certificate for the specified principal name, verifies it, and (if the verification is successful) extracts the public key stored in it and returns it to the caller.

The returned key information handle can be interrogated by various **pkc_cert_** routines to extract the actual key and determine the degree of trust that can be placed in the returned key.

If **domain** and **desired_usage** are passed as non-**NULL** pointers, upon successful return these parameters will describe the domain and permitted usage(s) of the specified key. Policies that do not distinguish keys according to domain will indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage will indicate all usages are permitted.

The **desired_usage** parameter consists of a bit mask, formed by AND-ing together one or more of the constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A **NULL** can be specified for **desired_usage** to indicate that keys for any usage should be retrieved.

Note that some of the routine's parameters relate to X.509 version 3 certificates, support for which is not committed for DCE 1.2. The API has been designed with the intent that it be capable of supporting all currently defined versions of X.509, so that it need not change when version 3 support is added. For version 1 or version 2 policies and certificates, the *desired_usage* parameter will be ignored, and the *initial_explicit_policy_required* parameter must be zero (specifying that the policy need not explicitly appear in the first certificate).

Related Information

Functions: **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_trustlist_intro

Purpose

Introduction to the certificate manipulation facility

Description

This reference page describes the data types used by the certificate manipulation facility.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Trust Lists

The trust list is the fundamental object within the certificate manipulation facility. A trust list is a set of keys which are trusted, plus a list of revoked certificate serial numbers. Keys are inserted into a trust list either directly (via the **pkc_add_trusted_key(3sec)** function) or indirectly (via the **pkc_check_cert_against_trustlist(3sec)** function). The latter routine will only add keys if the certificate signature can be verified by a key already in the trust list, and if the certificate has not been revoked.

Currently, trust lists are relatively static objects: once a key is inserted, its trust properties do not change. If, for example, a key is added that is capable of extending the trust in another key within the list, the second key is not automatically updated.

Using the Certificate Manipulation Facility

The way that a policy module is expected to use the facility is as follows.

1. Create an initial trust list containing the directly trusted keys, that is, the start point(s) of all valid trust chains.

   Typically, this set of keys will be used for multiple certificate chain evaluations. If the policy wishes to impose additional path constraints over the constraints expressed within the certificates, it must maintain a master copy of the original trust list and clone it to create a modifiable version for each chain the policy module wants to verify. After verification of a candidate chain, the cloned trust list must be discarded so that the next trial verification starts from a known state.
2. Using the initial trust list as a starting point, the policy module retrieves a chain of certificates and adds them to the trust list one by one, starting with the certificate(s) closest to the start point(s).

   Multiple chains may be evaluated simultaneously using a single trust list for policies that do not wish to impose additional constraints on the trust chain; however the policy module must ensure that for each trust-chain, certificates are added in the correct order. A future auto-update enhancement may lift this requirement.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**,

pkc_revoke_certificate(3sec), **pkc_revoke_certificates(3sec)**. Classes:
pkc_ca_key_usage.class(3sec), **pkc_constraints.class(3sec)**,
pkc_generic_key_usage.class(3sec), **pkc_key_policies.class(3sec)**,
pkc_key_policy.class(3sec), **pkc_key_usage.class(3sec)**,
pkc_name_subord_constraint.class(3sec),
pkc_name_subord_constraints.class(3sec),
pkc_name_subtree_constraint.class(3sec),
pkc_name_subtree_constraints.class(3sec),
pkc_pending_revocation.class(3sec), **pkc_revocation.class(3sec)**,
pkc_revocation_list.class(3sec), **pkc_trust_list.class(3sec)**,
pkc_trust_list_element.class(3sec), **pkc_trusted_key.class(3sec)**.

gssapi_intro

Purpose

Generic security service application programming interface

Description

This introduction includes general information about the generic security service application programming interface (GSSAPI) defined in Internet RFC 1508, *Generic Security Service Application Programming Interface*, and RFC 1509, *Generic Security Service API : C-bindings*. It also includes an overview of error handling, data types, and calling conventions, including the following:

- Integer types
- String and similar data
- Object identifiers (OIDs)
- Object identifier sets (OID sets)
- Credentials
- Contexts
- Authentication tokens
- Major status values
- Minor status values
- Names
- Channel bindings
- Optional parameters

General Information

The GSSAPI provides security services to applications using peer-to-peer communications (instead of DCE-secure RPC). Using DCE GSSAPI routines, applications can perform the following operations:

- Enabling an application to determine another application's user
- Enabling an application to delegate access rights to another application
- Applying security services, such as confidentiality and integrity, on a per-message basis

GSSAPI represents a secure connection between two communicating applications with a data structure called a *security context*. The application that establishes the secure connection is called the *context initiator* or simply *initiator*. The context initiator is like a DCE RPC client. The application that accepts the secure connection is the *context acceptor* or simply *acceptor*. The context acceptor is like a DCE RPC server.

There are four stages involved in using the GSSAPI, as follows:

1. The context initiator acquires a credential with which it can prove its identity to other processes. Similarly, the context acceptor acquires a credential to enable it to accept a security context. Either application may omit this credential acquistion and use their default credentials in subsequent stages. See the section on credentials for more information.

The applications use credentials to establish their global identity. The global identity can be, but is not necessarily, related to the local user name under which the application is running. Credentials can contain either of the following:

- Login context

  The login context includes a principal's network credentials, as well as other account information.

- Principal name and a key

  The key corresponding to the principal name must be registered with the DCE security registration in a key table. A set of GSSAPI routines enables applications to register and use principal names.

2. The communicating applications establish a joint security context by exchanging authentication tokens.

   The security context is a pair of GSSAPI data structures that contain information that is shared between the communicating applications. The information describes the state of each application. This security context is required for per-message security services.

   To establish a security context, the context initiator calls the **gss_init_sec_context()** routine to get a *token*. The token is cryptographically protected, opaque data. The context initiator transfers the token to the context acceptor, which in turn passes the token to the **gss_accept_sec_context()** routine to decode and extract the shared information.

   As part of the establishing the the security context, the context initiator is authenticated to the context acceptor. The context initiator can require the context acceptor to authenticate itself in return.

   The context initiator can *delegate* rights to allow the context acceptor to act as its agent. Delegation means the context initiator gives the context acceptor the ability to initiate additional security contexts as an agent of the context initiator. To delegate, the context initiator sets a flag on the **gss_init_sec_context()** routine indicating that it wants to delegate and sends the returned token in the normal way to the context acceptor. The acceptor passes this token to the **gss_accept_sec_context()** routine, which generates a delegated credential. The context acceptor can use the credential to initiate additional security contexts.

3. The applications exchange protected messages and data.

   The applications can call GSSAPI routines to protect data exchanged in messages. The application sends a protected message by calling the appropriate GSSAPI routine to do the following:

   - Apply protection
   - Bind the message to the appropriate security context

   The application can then send the resulting information to the peer application.

   The application that receives the message passes the received data to a GSSAPI routine, which removes the protection and validates the data.

   GSSAPI treats application data as arbitrary octet strings. The GSSAPI per-message security services can provide either of the following:

   - Integrity and authentication of data origin
   - Confidentiality, integrity, and authentication of data origin

4. When the applications have finished communicating, either one may instruct GSSAPI to delete the security context.

There are two sets of GSSAPI routines, as follows:

- Standard GSSAPI routines, which are defined in the Internet RFC 1508, *Generic Security Service Application Programming Interface*, and RFC 1509, *Generic Security Service API : C-bindings*. These routines have the prefix **gss**_.
- OSF DCE extensions to the GSSAPI routines. These are additional routines that enable an application to use DCE security services. These routines have the prefix **gssdce**_.

The following sections provide an overview of the GSSAPI error handling and data types.

Error Handling

Each GSSAPI routine returns two types of status values:

- Major status values, which are generic API routine errors or calling errors defined in RFC 1509.
- Minor status values, which indicate DCE-specific errors.

If a routine has output parameters that contain pointers for storage allocated by the routine, the output parameters will always contain a valid pointer even if the routine returns an error. If no storage was allocated, the routine sets the pointer to NULL and sets any length fields associated with the pointers (such as in the **gss_buffer_desc** structure) to 0 (zero).

Minor status values usually contain more detailed information about the error. They are not, however, portable between GSSAPI implementations. When designing portable applications, use major status values for handling errors. Use minor status values to debug applications and to display error and error-recovery information to users.

GSSAPI Data Types

This section provides an overview of the GSSAPI data types and their definitions.

Integer Types

The GSSAPI defines the following integer data type:

OM_uint32 32-bit unsigned integer

This integer data type is a portable data type that the GSSAPI routine definitions use for guaranteed minimum bit-counts.

String and Similar Data

Many of the GSSAPI routines take arguments and return values that describe contiguous multiple-byte data, such as opaque data and character strings. Use the **gss_buffer_t** data type, which is a pointer to the buffer descriptor **gss_buffer_desc**, to pass the data between the GSSAPI routines and applications.

The **gss_buffer_t** data type has the following structure:

```
typedef struct gss_buffer_desc_struct {
size_t  length;
void    *value;
} gss_buffer_desc, *gss_buffer_t;
```

The *length* field contains the total number of bytes in the data and the *value* field contains a pointer to the actual data.

When using the **gss_buffer_t** data type, the GSSAPI routine allocates storage for any data it passes to the application. The calling application must allocate the **gss_buffer_desc** object. It can initialize unused **gss_buffer_desc** objects with the value **GSS_C_EMPTY_BUFFER**. To free the storage, the application calls the **gss_release_buffer()** routine.

Object Identifier

Applications use the **gss_OID** data type to choose a security mechanism, either DCE security or Kerberos, and to specify name types. Select a security mechanism by using the following two OIDs:

- To use DCE security, specify either **GSSDCE_C_OID_DCE_KRBV5_DES** or **GSS_C_NULL_OID**.
- To use Kerberos Version 5, specify **GSSDCE_C_OID_KRBV5_DES**.

Use of the default security mechanisms, specified by the constant **GSS_C_NULL_OID**, helps to ensure the portability of the application.

The **gss_OID** data type contains tree-structured values defined by ISO and has the following structure:

```
typedef struct gss_OID_desc_struct {
OM_uint32 length;
void      *elements;
}  gss_OID_desc, *gss_OID;
```

The *elements* field of the structure points to the first byte of an octet string containing the ASN.1 BER encoding of the value of the **gss_OID** data type. The *length* field contains the number of bytes in the value.

The **gss_OID_desc** values returned from the GSSAPI are read-only values. The application should not try to deallocate them.

Object Identifier Sets

The **gss_OID_set** data type represents one or more object identifiers. The values of the **gss_OID_set** data type are used to do the following:

- Report the available mechanisms supported by GSSAPI
- Request specific mechanisms
- Indicate which mechanisms a credential supports

The **gss_OID_set** data type has the following structure:

```
typedef struct gss_OID_set_desc_struct {
int     count;
gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;
```

The *count* field contains the number of OIDs in the set. The *elements* field is a pointer to an array of **gss_oid_desc** objects, each describing a single OID. The application calls the **gss_release_oid_set()** routine to deallocate storage associated with the **gss_OID_set** values that the GSSAPI routines return to the application.

Credentials

Credentials establish, or prove, the identity of an application or other principal.

The **gss_cred_id_t** data type is an atomic data type that identifies a GSSAPI credential data structure.

Contexts

The security context is a pair of GSSAPI data structures that contain information shared between the communicating applications. The information describes the cryptographic state of each application. This security context is required for per-message security services and is created by a successful authentication exchange.

The **gss_ctx_id_t** data type contains an atomic value that identifies one end of a GSSAPI security context. The data type is opaque to the caller.

Authentication Tokens

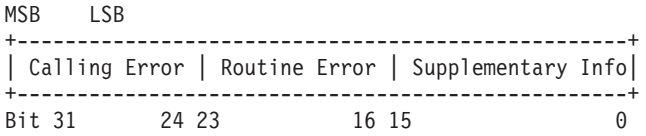
GSSAPI uses tokens to maintain the synchronization between the applications sharing a security context. The token is a cryptographically protected bit string generated by DCE security at one end of the GSSAPI security context for use by the peer application at the other end of the security context. The data type is opaque to the caller.

The applications use the **gss_buffer_t** data type as tokens to GSSAPI routines.

Major Status Values

GSSAPI routines return GSS status codes as their **OM_uint32** function value. These codes indicate either generic API routine errors or calling errors.

A GSS status code can indicate a single, fatal generic API error from the routine and a single calling error. Additional status information can also be contained in the GSS status code. The errors are encoded into a 32-bit GSS status code, as follows:



If a GSSAPI routine returns a GSS status code whose upper 16 bits contain a nonzero value, the call failed. If the calling error field is nonzero, the context initiator's use of the routine was in error. In addition, the routine can indicate additional information by setting bits in the supplementary information field of the status code. The tables that follow describe the routine errors, calling errors, and supplementary information status bits and their meanings.

The following table lists the GSSAPI routine errors and their meanings:

Name	Field Value	Meaning
GSS_S_BAD_MECH	1	The required mechanism is unsupported.
GSS_S_NAME	2	The name passed is invalid.

Name	Field Value	Meaning
GSS_S_NAMETYPE	3	The name passed is unsupported.
GSS_S_BAD_BINDINGS	4	The channel bindings are incorrect.
GSS_S_BAD_STATUS	5	A status value was invalid.
GSS_S_BAD_SIG	6	A token had an invalid signature.
GSS_S_NO_CRED	7	No credentials were supplied.
GSS_S_NO_CONTEXT	8	No context has been established.
GSS_S_DEFECTIVE_TOKEN	9	A token was invalid.
GSS_S_DEFECTIVE _CREDENTIAL	10	A credential was invalid.
GSS_S_CREDENTIALS _EXPIRED	11	The referenced credentials expired.
GSS_S_CONTEXT_EXPIRED	12	The context expired.
GSS_S_FAILURE	13	The routine failed. Check minor status codes.

The following table lists the calling error values and their meanings:

Name	Field Value	Meaning
GSS_S_CALL_INACCESSIBLE _READ	1	Could not read a required input parameter.
GSS_S_CALL_INACCESSIBLE _WRITE	2	Could not write a required output parameter.
GSS_S_BAD_STRUCTURE	3	A parameter was incorrectly structured.

The following table lists the supplementary bits and their meanings.

Name	Bit Number	Meaning
GSS_S_CONTINUE_NEEDED	0 (LSB)	Call the routine again to complete its function.
GSS_S_DUPLICATE_TOKEN	1	The token was a duplicate of an earlier token.
GSS_S_OLD_TOKEN	2	The token's validity period expired; the routine cannot verify that the token is not a duplicate of an earlier token.
GSS_S_UNSEQ_TOKEN	3	A later token has been processed.

All **GSS_S_** symbols equate to complete **OM_uint32** status codes, rather than to bitfield values. For example, the actual value of **GSS_S_BAD_NAMETYPE** (value 3 in the routine error field) is 3 << 16.

The major status code **GSS_S_FAILURE** indicates that DCE security detected an error for which no major status code is available. Check the minor status code for details about the error. See the section on minor status values for more information.

The GSSAPI provides the following three macros:

- **GSS_CALLING_ERROR()**
- **GSS_ROUTINE_ERROR()**
- **GSS_SUPPLEMENTARY_INFO()**

Each macro takes a GSS status code and masks all but the relevant field. For example, when you use the **GSS_ROUTINE_ERROR()** macro on a status code, it returns a value. The value of the macro is arrived at by using only the routine errors field and zeroing the values of the calling error and the supplementary information fields.

An additional macro, **GSS_ERROR()**, lets you determine whether the status code indicated a calling or routine error. If the status code indicated a calling or routine error, the macro returns a nonzero value. If no calling or routine error is indicated, the routine returns a 0 (zero).

Note:

> At times, a GSSAPI routine that is unable to access data can generate a platform-specific signal, instead of returning a **GSS_S_CALL_INACCESSIBLE_READ** or **GSS_S_CALL_INACCESSIBLE_WRITE** status value.

Minor Status Values

The GSSAPI routines return a *minor_status* parameter to indicate errors from either DCE security or Kerberos. The parameter can contain a single error, indicated by an **OM_uint32** value. The **OM_uint32** data type is equivalent to the DCE data type **error_status_t** and can contain any DCE-defined error.

Names

Names identify principals. The GSSAPI authenticates the relationship between a name and the principal claiming the name.

Names are represented in the following two forms:

- A printable form, for presentation to an application
- An internal, canonical form that is used by the API and is opaque to applications

The **gss_import_name()** and **gss_display_name()** routines convert names between their printable form and their **gss_name_t** data type. GSSAPI supports only DCE principal names, which are identified by the constant OID, **GSSCDE_C_OID_DCENAME**.

The **gss_compare_names()** routine compares internal form names.

Channel Bindings

You can define and use channel bindings to associate the security context with the communications channel that carries the context. Channel bindings are communicated to the GSSAPI by using the following structure:

```
typedef struct gss_channel_binding_struct {
OM_uint32       initiator_addrtype;
gss_buffer_desc initiator_address;
OM_uint32       acceptor_addrtype;
gss_buffer_desc aceptor_address;
gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

Use the *initiator_addrtype* and *acceptor_addrtype* fields to initiate the type of addresses contained in the *initiator_address* and *acceptor_address* buffers. The address types and their **addrtype** values are as follows:

Unspecified
GSS_C_AF_UNSPEC

Host-local
GSS_C_AF_LOCAL

DARPA Internet
GSS_C_AF_INET

ARPAnet IMP
GSS_C_AF_IMPLINK

pup protocols (for example, BSP)
GSS_C_AF_PUP

MIT CHAOS protocol
GSS_C_AF_CHAOS

XEROX NS
GSS_C_AF_NS

nbs GSS_C_AF_NBS

ECMA GSS_C_AF_ECMA

datakit protocols
GSS_C_AF_DATAKIT

CCITT protocols (for example, X.25)
GSS_C_AF_CCITT

IBM SNA
GSS_C_AF_SNA

Digital DECnet
GSS_C_AF_DECnet

Direct data link interface
GSS_C_AF_DLI

LAT GSS_C_AF_LAT

NSC Hyperchannel
GSS_C_AF_HYLINK

AppleTalk
GSS_C_AF_APPLETALK

BISYNC 2780/3780
GSS_C_AF_BSC

Distributed system services
GSS_C_AF_DSS

OSI TP4
GSS_C_AF_OSI

X25 GSS_C_AF_X25

No address specified
GSS_C_AF_NULLADDR

The tags specify address families rather than addressing formats. For address families that contain several alternative address forms, the *initiator_address* and the *acceptor_address* fields should contain sufficient information to determine which address form is used. Format the bytes that contain the addresses in the order in which the bytes are transmitted across the network.

The GSSAPI creates an octet string by concatenating all the fields (*initiator_addrtype*, *initiator_address*, *acceptor_addrtype*, *acceptor_address*, and *application_data*). The security mechanism signs the octet string and binds the signature to the token generated by the **gss_init_sec_context()** routine. The context acceptor presents the same bindings to the **gss_accept_sec_context()** routine, which evaluates the signature and compares it to the signature in the token. If the signatures differ, the **gss_accept_sec_context()** routine returns a **GSS_S_BAD_BINDINGS** error, and the context is not established.

Some security mechanisms check that the *initiator_address* field of the channel bindings presented to the **gss_init_sec_context()** routine contains the correct network address of the host system. Therefore portable applications should use either the correct address type and value or the **GSS_C_AF_NULLADDR** for the *initiator_addrtype* address field. Some security mechanisms include the channel binding data in the token instead of a signature, so portable applications should not use confidential data as channel-binding components. The GSSAPI does not verify the address or include the plain text bindings information in the token.

Optional Parameters

In routine descriptions, *optional parameters* allow the application to request default behaviors by passing a default value for the parameter. The following conventions are used for optional parameters:

Convention	Value Default	Explanation
gss_buffer_t types	**GSS_C_NO_BUFFER**	For an input parameter, indicates no data is supplied. For an output parameter, indicates that the information returned is not required by the application.
Integer types (input)		Refer to the reference pages for default values.
Integer types (output)	NULL	Indicates that the application does not require the information.
Pointer types (output)	NULL	Indicates that the application does not require the information.
OIDs	**GSS_C_NULL_OID**	Indicates the default choice for name type or security mechanism.
OID sets	**GSS_C_NULL_OID_SET**	Indicates the default set of security mechanisms, DCE security and Kerberos.

Convention	Value Default	Explanation
Credentials	**GSS_C_NO_CREDENTIAL**	Indicates that the application should use the default credential handle.
Channel bindings	**GSS_C_NO_CHANNEL _BINDINGS**	Indicates that no channel bindings are used.

Related Information

Books: *OSF DCE Application Development Guide—Core Components*.

dce_acl_copy_acl

Purpose

Copies an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_copy_acl(
     sec_acl_t *source
     sec_acl_t *target
     error_status_t *status);
```

Parameters

Input

source A pointer to the ACL to be copied.

target A pointer to the new ACL that is to receive the copy.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_copy_acl()** routine makes a copy of a specified ACL. The caller passes the space for the target ACL, but the space for the **sec_acl_entries** array is allocated. To free the allocated space, call **dce_acl_obj_free_entries()**, which frees the entries, but not the ACL itself.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

rpc_s_no_memory
The **rpc_sm_allocate()** routine could not obtain memory.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_free_entries(3sec)**.

dce_acl_inq_acl_from_header

Purpose

Retrieves the UUID of an ACL from an item's header in a backing store

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_inq_acl_from_header(
     dce_db_header_t db_header
     sec_acl_type_t sec_acl_type
     uuid_t *acl_uuid
     error_status_t *status);
```

Parameters

Input

db_header
The backing store header containing the ACL object.

sec_acl_type
The type of ACL to be identified:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

Output

acl_uuid
A pointer to the UUID of the ACL object.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_acl_from_header()** routine gets the UUID for an ACL object of the specified type from the specified backing store header.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

db_s_key_not_found
The specified key was not found in the backing store. (This error is passed through from **dce_db_fetch()**.)

db_s_bad_index_type
The key's type is wrong, or else the backing store is not by name or by UUID. (This error is passed through from **dce_db_fetch()**.)

sec_acl_invalid_type
The *sec_acl_type* parameter does not contain a valid type.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_resolve_by_name(3sec)**, **dce_acl_resolve_by_uuid(3sec)**.

dce_acl_inq_client_creds

Purpose

Returns the client's credentials

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_inq_client_creds(
     handle_t handle
     sec_cred_pa_handle_t *creds
     error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

Output

creds A pointer to the returned credentials, or NULL if unauthorized.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_client_creds()** routine returns the client's security credentials found through the RPC binding handle.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

rpc_s_authn_authz_mismatch
Either the client, or the server, or both is not using the **rpc_c_authz_dce** authorization service.

rpc_s_invalid_binding
Invalid RPC binding handle.

rpc_s_wrong_kind_of_binding
Wrong kind of binding for operation.

rpc_s_binding_has_no_auth
Binding has no authentication information. The client or the server should have called **rpc_binding_set_auth_info()**.

Related Information

Functions: **dce_acl_inq_client_permset(3sec)**, **dce_acl_inq_permset_for_creds(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_client_permset

Purpose

Returns the client's permissions corresponding to an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_inq_client_permset(
     handle_t handle
     uuid_t *mgr_type
     uuid_t *acl_uuid
     uuid_t *owner_id
     uuid_t *group_id
     sec_acl_permset_t *permset
     error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl_uuid
A pointer to the UUID of the ACL.

owner_id
Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id
Identifies the group to which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

Output

permset
The set of permissions allowed to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_client_permset()** routine returns the client's permissions that correspond to the ACL. It finds the ACL in the database as defined for this ACL

manager type with **dce_acl_register_object_type()**. The client's credentials are determined from the binding handle. The ACL and credentials determine the permission set.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

acl_s_bad_manager_type
The *mgr_type* parameter does not match the manager type in the ACL itself.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_inq_client_pac(3sec)**, **dce_acl_inq_permset_for_pac(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_permset_for_creds

Purpose

Determines a principal's complete extent of access to an object

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_inq_permset_for_creds(
     sec_cred_pa_handle_t *creds
     sec_acl_t *ap
     uuid_t *owner_id
     uuid_t *group_id
     sec_acl_posix_semantics_t posix_semantics
     sec_acl_permset_t *perms
     error_status_t *status);
```

Parameters

Input

creds The security credentials that represent the principal.

ap The ACL that represents the object.

owner_id
Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id
Identifies the group in which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

posix_semantics
This parameter is currently unused in OSF's implementation.

Output

perms A bit mask containing a 1 bit for each permission granted by the ACL and 0 (zero) bits elsewhere.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **dce_acl_inq_permset_for_creds()** routine returns a principal's complete extent of access to some object. This routine is useful for implementing operations such as the conventional UNIX access function.

The values allowed for the credentials representing the principal include NULL or unauthenticated.

The routine normally returns TRUE, even when the access permissions are determined to be all 0 (zero) bits (**dce_acl_c_no_permissions**). It returns FALSE only on illogical error conditions (such as unsupported ACL entry types), in which case the status output gets the error status code and the *perms* is set to **dce_acl_c_no_permissions**.

All ACL entry types (of type **sec_acl_entry_type_t**) are supported by this routine

Notes

The meanings of the permission bits have no effect on the action of the **dce_acl_inq_permset_for_creds()** routine. The interpretation of the bits is left entirely to the application.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_inq_client_creds(3sec)**, **dce_acl_inq_client_permset(3sec)**, **dce_acl_register_object_type(3sec)**.

dce_acl_inq_prin_and_group.3sec

Purpose

Inquires the principal and group of an RPC caller

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_inq_prin_and_group(
     handle_t handle
     uuid_t *principal
     uuid_t *group
     error_status_t *status);
```

Parameters

Input

handle The remote procedure call binding handle.

Output

principal
The UUID of the principal of the caller of the RPC.

group The UUID of the group of the caller of the RPC.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_inq_prin_and_group()** routine finds the principal and group of the caller of a remote procedure call. This information is useful for filling in the *owner_id* and *group_id* fields of standard data or object headers. Setting the owner and group make sense only if your ACL manager will handle owners and groups, which you specify with the **dce_acl_c_has_owner** and **dce_acl_c_has_groups** flags to **dce_acl_register_object_type()**.

If the caller is unauthenticated, the principal and group are filled with the **NIL** UUID, generated through **uuid_create_nil()**.

Examples

```
dce_db_std_header_init(db, &data, ..., &st);
dce_acl_inq_prin_and_group(h, \
 &data.h.owner_id, &data.h.group_id, &st);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages. The **dce_acl_inq_prin_and_group()** routine can return errors from

dce_acl_inq_client_creds(), **sec_cred_get_initiator()**, and **sec_cred_get_pa_data()**. It generates no error messages of its own.

Related Information

Functions: **dce_acl_register_object_type(3sec)**.

dce_acl_is_client_authorized

Purpose

Checks whether a client's credentials are authenticated

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_is_client_authorized(
     handle_t handle
     uuid_t *mgr_type
     uuid_t *acl_uuid
     uuid_t *owner_id
     uuid_t *group_id
     sec_acl_permset_t desired_perms
     boolean32 *authorized
     error_status_t *status);
```

Parameters

Input

handle The client's binding handle.

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl_uuid
A pointer to the UUID of the ACL.

owner_id
Identifies the owner of the object that is protected by the specified ACL. If the **sec_acl_e_type_user_obj** ACLE (ACL entry) exists, then the *owner_id* (**uuid_t** pointer) can not be NULL. If it is, then the error **sec_acl_expected_user_obj** is returned.

group_id
Identifies the group to which the object that is protected by the specified ACL belongs. If the a **sec_acl_e_type_group_obj** ACLE exists, the *group_id* (**uuid_t** pointer) can not be NULL. If it is, the error **sec_acl_expected_group_obj** is returned.

desired_perms
A permission set containing the desired privileges. This is a 32-bit set of permission flags. The flags may represent the conventional file system permissions (read, write, and execute), the extended AFS permissions (owner, insert, and delete), or some other permissions supported by the specific application ACL manager. For example, a bit that is unused for file system permissions may mean withdrawals are allowed for a bank ACL manager, while it may mean matrix inversions are allowed for a CPU ACL manager. The *mgr_type* identifies the semantics of the bits.

Output

authorized
A pointer to the TRUE or FALSE return value of the routine.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_is_client_authorized()** routine returns TRUE in the *authorized* parameter if and only if all of the desired permissions (represented as bits in *desired_perms*) are included in the actual permissions corresponding to the *handle*, the *mgr_type*, and the *acl_uuid* UUID. Otherwise, the returned value is FALSE.

Notes

The routine's return value is **void**. The returned **boolean32** value is in the *authorized* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

acl_s_bad_manager_type
The *mgr_type* does not match the manager type in the ACL itself.

error_status_ok
The call was successful.

dce_acl_obj_add_any_other_entry

Purpose

Adds permissions for any_other ACL entry to a given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_any_other_entry(
     sec_acl_t *acl
     sec_acl_permset_t permset
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset
The permissions to be granted to **sec_acl_e_type_any_other**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_any_other_entry()** routine adds an ACL entry for **sec_acl_e_type_any_other** access to the specified ACL. It is equivalent to calling the **dce_acl_obj_add_obj_entry()** routine with the **sec_acl_e_type_any_other** entry type, but is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_obj_entry(3sec)**.

dce_acl_obj_add_foreign_entry

Purpose

Adds permissions for an ACL entry for a foreign user or group to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_foreign_entry(
     sec_acl_t *acl
     sec_acl_entry_type_t entry_type
     sec_acl_permset_t permset
     uuid_t *realm
     uuid_t *id
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type
Must be one of the following types:

- **sec_acl_e_type_foreign_user**
- **sec_acl_e_type_foreign_group**
- **sec_acl_e_type_for_user_deleg**
- **sec_acl_e_type_for_group_deleg**

permset
The permissions to be granted to the foreign group or foreign user.

realm The UUID of the foreign cell.

id The UUID identifying the foreign group or foreign user.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_foreign_entry()** routine adds an ACL entry for **sec_acl_e_type_foreign_xxx** access to the specified ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not one of the four specified types.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**, **sec_id_parse_name(3sec)**.

dce_acl_obj_add_group_entry

Purpose

Adds permissions for a group ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_group_entry(
     sec_acl_t *acl
     sec_acl_permset_t permset
     uuid_t *group
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset
The permissions to be granted to the group.

group The UUID identifying the group.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_group_entry()** routine adds a group ACL entry to the given ACL. It is equivalent to calling the **dce_acl_obj_add_id_entry()** routine with the **sec_acl_e_type_group** entry type, but is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**.

dce_acl_obj_add_id_entry

Purpose

Adds permissions for an ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_id_entry(
     sec_acl_t *acl
     sec_acl_entry_type_t entry_type
     sec_acl_permset_t permset
     uuid_t *id
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type
Must be one of the following types:

- **sec_acl_e_type_user**
- **sec_acl_e_type_group**
- **sec_acl_e_type_foreign_other**
- **sec_acl_e_type_user_deleg**
- **sec_acl_e_type_group_deleg**
- **sec_acl_e_type_for_other_deleg**

permset
The permissions to be granted to the **user**, **group**, or **foreign_other**.

id The UUID identifying the **user**, **group**, or **foreign_other** to be added

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_id_entry()** routine adds an ACL entry (user or group, domestic or foreign) to the given ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not one of the six specified types.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_group_entry(3sec)**, **dce_acl_obj_add_user_entry(3sec)**.

dce_acl_obj_add_obj_entry

Purpose

Adds permissions for an object (obj) ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_obj_entry(
     sec_acl_t *acl
     sec_acl_entry_type_t entry_type
     sec_acl_permset_t permset
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

entry_type
Must be one of these types:

- **sec_acl_e_type_unauthenticated**
- **sec_acl_e_type_any_other**
- **sec_acl_e_type_mask_obj**
- **sec_acl_e_type_user_obj**
- **sec_acl_e_type_group_obj**
- **sec_acl_e_type_other_obj**
- **sec_acl_e_type_user_obj_deleg**
- **sec_acl_e_type_group_obj_deleg**
- **sec_acl_e_type_other_obj_deleg**
- **sec_acl_e_type_any_other_deleg**

permset
The permissions to be granted.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_obj_entry()** routine adds an **obj** ACL entry to the given ACL.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_duplicate_entry
An **obj** ACL entry type already exits for the given ACL.

sec_acl_invalid_entry_type
The type specified in *entry_type* is not a valid ACL entry type.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_any_other_entry(3sec)**, **dce_acl_obj_add_unauth_entry(3sec)**.

dce_acl_obj_add_unauth_entry

Purpose

Adds permissions for unauthenticated ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_unauth_entry(
     sec_acl_t *acl
     sec_acl_permset_t permset
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset
The permissions to be granted for **sec_acl_e_type_unauthenticated**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_unauth_entry()** routine adds ACL entry for **sec_acl_e_type_unauthenticated** to the given ACL. It is equivalent to calling the **dce_acl_obj_add_obj_entry()** routine with the **sec_acl_e_type_unauthenticated** entry type, but it is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_obj_entry(3sec)**.

dce_acl_obj_add_user_entry

Purpose

Adds permissions for a user ACL entry to the given ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_add_user_entry(
     sec_acl_t *acl
     sec_acl_permset_t permset
     uuid_t *user
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be modified.

permset
The permissions to be granted to the user.

user The UUID identifying the user to be added.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_add_user_entry()** routine adds a user ACL entry to the given ACL. It is equivalent to calling the **dce_acl_obj_add_id_entry()** routine with the **sec_acl_e_type_user** entry type, but it is more convenient.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_add_id_entry(3sec)**.

dce_acl_obj_free_entries

Purpose

Frees space used by an ACL's entries

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_free_entries(
     sec_acl_t *acl
     error_status_t *status);
```

Parameters

Input

acl A pointer to the ACL that is to be freed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_free_entries()** routine frees space used by an ACL's entries, then sets the pointer to the ACL entry array to NULL and the entry count to 0 (zero).

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_init(3sec)**.

dce_acl_obj_init

Purpose

Initializes an ACL

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_obj_init(
     uuid_t *mgr_type
     sec_acl_t *acl
     error_status_t *status);
```

Parameters

Input

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

acl A pointer to the ACL that is to be created.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_obj_init()** routine initializes an ACL. The caller passes in the pointer to the already-existing ACL structure (of type **sec_acl_t**), for which the caller provides the space.

Examples

This example shows the use of **dce_acl_obj_init()** and the corresponding routine to free the entries, **dce_acl_obj_free_entries()**.

```
sec_acl_t acl;
extern uuid_t my_mgr_type;
error_status_t status;
dce_acl_obj_init(&my_mgr_type, &acl, &status);
/* ... use the ACL ... */
dce_acl_obj_free_entries(&acl, &status);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_obj_free_entries(3sec)**.

dce_acl_register_object_type

Purpose

Registers an ACL manager's object type

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_register_object_type(
     dce_db_handle_t db
     uuid_t *mgr_type
     unsigned32 printstring_size
     sec_acl_printstring_t *printstring
     sec_acl_printstring_t *mgr_info
     sec_acl_permset_t control_perm
     sec_acl_permset_t test_perm
     dce_acl_resolve_func_t resolver
     void *resolver_arg
     unsigned32 flags
     error_status_t *status);
```

Parameters

Input

db The *db* parameter specifies the handle to the backing store database in which the ACL objects are stored. It must be indexed by UUID and not use backing store headers. The database is obtained through **dce_db_open()**, which is called prior to this routine.

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

printstring_size
The number of items in the *printstring* array.

printstring
An array of **sec_acl_printstring_t** structures containing the printable representation of each specified permission. These are the printstrings used by **dcecp** or other ACL editors.

mgr_info
A single **sec_acl_printstring_t** containing the name and short description for the given ACL manager.

control_perm
The permission set needed to change an ACL, typically **sec_acl_perm_control**. If the value is 0, then anyone is allowed to change the ACL. The permission must be listed in the **printstring**.

test_perm
The permission set needed to test an ACL, typically **sec_acl_perm_test**. If the value is 0, then anyone is allowed to test the ACL. The permissions must be listed in the **printstring**.

resolver
The function for finding an ACL's UUID.

resolver_arg
The argument to pass to the *resolver* function. If using **dce_acl_resolve_by_name()** or **dce_acl_resolve_by_uuid()**, then pass the database handle to the name or UUID backing store database. The backing store must use the standard backing store header. See **dce_db_open(3dce)**.

flags A bit mask with the following possible bit values:

dce_acl_c_orphans_ok
If this bit is specified, it is possible to replace an ACL with one in which no control bits are turned on in any of the ACL entries. (Use the **rdacl_replace** operation to replace an ACL.) This is a write-once operation, and once it has been done, no one can change the ACL.

dce_acl_c_has_owner
If this bit is set, then the ACL manager supports the concept of user owners of objects. This is required to use ACL entries of type **user_obj** and **user_obj_deleg**. entries such as **sec_acl_e_type_user_obj**.

dce_acl_c_has_groups
A similar bit for group owners of objects.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_register_object_type()** routine registers an ACL manager's object types with the ACL library.

The *resolver* function may be the **dce_acl_resolve_by_name()** or the **dce_acl_resolve_by_uuid()** routine, if the application uses the standard header in the backing store database, or it may be some other user-supplied routine, as appropriate. A user-supplied routine must be of type **dce_acl_resolve_func_t**. The *resolver* function finds the UUID of the ACL of the given object. The *resolver*'s parameters must match the type **dce_db_convert_func_t** defined in the file **<dce/aclif.h>**. Observe the use of the resolver function **dce_acl_convert_func()** in **EXAMPLES**.

Unless the **dce_acl_c_orphans_ok** bit is set in the *flags* parameter, all ACLs must always have *someone* able to modify the ACL.

Another way to express this is that if **dce_acl_c_orphans_ok** is cleared in a call to **dce_acl_register_object_type()** where a *control_perm* value is specified, then a subsequent ACL replacement using an ACL that has no control bits set in any nondelegation entry will fail, resulting in the **acl_s_no_control_entries** error. If **dce_acl_c_orphans_ok** is set, but no *control_perm* bits are specified, then **dce_acl_c_orphans_ok** is ignored, and the replacement works in all cases.

Files

/usr/include/dce/aclif.h
Definition of **dce_acl_resolve_func_t**.

Examples

The **dce_acl_register_object_type()** routine should be called once for each type of object that the server manages. A typical call is shown below. The sample code defines three variables: the manager printstring, the ACL printstrings, and the ACL database. Note that the manager printstring does not define any permission bits; they will be set by the library to be the union of all permissions in the ACL printstring. The code also uses the global **my_uuid** as the ACL manager type UUID. The ACL printstring uses the standard **sec_acl_perm_XXX** bits.

```
include <dce/aclif.h>

/* Manager help. */
sec_acl_printstring_t my_acl_help = {
    "me", "My manager"
};

/*
 * ACL permission descriptions;
 * these are from /usr/include/dce/aclbase.idl
 * This example refrains from redefining any of the
 * conventionally established bits.
 */

sec_acl_printstring_t my_printstring[] = {
    { "r", "read", sec_acl_perm_read },
    { "f", "foobar", sec_acl_perm_unused_00000080 },
    { "w", "write", sec_acl_perm_write },
    { "d", "delete, sec_acl_perm_delete },
    { "c", "control", sec_acl_perm_control }
};

dce_db_open("my_acldb", NULL,
    dce_db_c_std_header | dce_db_c_index_by_uuid,
    (dce_db_convert_func_t)dce_acl_convert_func,
    &dbh, &st);

dce_acl_register_object_type(dbh, &my_manager_uuid,
    sizeof my_printstring / sizeof my_printstring[0],
    my_printstring, &my_acl_help, sec_acl_perm_control,
    0, xxx_resolve_func, NULL, 0, &st);
```

If the ACL manager can use the standard collection of ACL bits (that is, has not defined any special ones), then it can use the global variable **dce_acl_g_printstring** that predefines a printstring. Here is an example of its use:

```
dce_acl_register_object_type(acl_db, &your_mgr_type,
    sizeof dce_acl_g_printstring / sizeof dce_acl_g_printstring[0],
    dce_acl_g_printstring, &your_acl_help,
    dced_perm_control, dced_perm_test, your_resolver, NULL, 0, st);
```

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_group_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission.

acl_s_owner_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_user_obj** or **sec_acl_e_type_user_obj_deleg** to a manager that does not support object users ownership.

acl_s_group_not_allowed
In a **rdacl_replace** operation an attempt was made to add an ACL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission. CL entry of type **sec_acl_e_type_group_obj** or **sec_acl_e_type_group_obj_deleg** to a manager that does not support object group ownership.

acl_s_no_control_entries
In a **rdacl_replace** operation an attempt was made to replace the ACL where no entries have control permission.

Related Information

Functions: **dce_acl_resolve_by_name(3sec)**, **dce_acl_resolve_by_uuid(3sec)**, **dce_db_open(3dce)**.

dce_acl_resolve_by_name

Purpose

Finds an ACL's UUID, given an object's name

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

void  dce_acl_resolve_by_name(
     handle_t handle
     sec_acl_component_name_t component_name
     sec_acl_type_t sec_acl_type
     uuid_t *mgr_type
     boolean32 writing
     void *resolver_arg
     uuid_t *acl_uuid
     error_status_t *status);
```

Parameters

Input

handle A client binding handle passed into the server stub. Use **sec_acl_bind()** to create this handle.

component_name
A character string containing the name of the target object.

sec_acl_type
The type of ACL to be resolved:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

writing This parameter is ignored in OSF's implementation.

resolver_arg
This argument is passed into **dce_acl_register_object_type()**. It should be a handle for a backing store indexed by name.

Output

acl_uuid
The ACL UUID, as resolved by **dce_acl_resolve_by_name()**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_resolve_by_name()** routine finds an ACL's UUID, given an object's name, as provided in the *component_name* parameter. The user does not call this function directly. It is an instance of the kind of function provided to the *resolver* argument of **dce_acl_register_object_type()**.

If **dce_acl_resolve_by_name()** and **dce_acl_resolve_by_uuid()** are inappropriate, the user of **dce_acl_register_object_type()** must provide some other *resolver* function.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_register_object_type(3sec)**, **dce_acl_resolve_by_uuid(3sec)**, **dce_db_open(3dce)**, **dce_db_header_fetch(3dce)**.

dce_acl_resolve_by_uuid

Purpose

Finds an ACL's UUID, given an object's UUID

Synopsis

```
#include <dce/dce.h>
#include <dce/aclif.h>

dce_acl_resolve_func_t  dce_acl_resolve_by_uuid(
     handle_t handle
     sec_acl_component_name_t component_name
     sec_acl_type_t sec_acl_type
     uuid_t *mgr_type
     boolean32 writing
     void *resolver_arg
     uuid_t *acl_uuid
     error_status_t *status);
```

Parameters

Input

handle A client binding handle passed into the server stub. Use **sec_acl_bind()** to create this handle.

component_name
A character string containing the name of the target object. (The **dce_acl_resolve_by_uuid()** routine ignores this parameter.)

sec_acl_type
The type of ACL to be resolved:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

mgr_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them.

writing This parameter is ignored in OSF's implementation.

resolver_arg
This argument is passed into **dce_acl_register_object_type()**. It should be a handle for a backing store indexed by UUID.

Output

acl_uuid
The ACL UUID, as resolved by **dce_acl_resolve_by_uuid()**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **dce_acl_resolve_by_uuid()** routine finds an ACL's UUID, given an object's UUID, as provided through the *handle* parameter. The user does not call this function directly. It is an instance of the kind of function provided to the *resolver* argument of **dce_acl_register_object_type()**.

If **dce_acl_resolve_by_uuid()** and **dce_acl_resolve_by_name()** are inappropriate, the user of **dce_acl_register_object_type()** must provide some other *resolver* function.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **dce_acl_register_object_type(3sec)**, **dce_acl_resolve_by_name(3sec)**, **dce_db_open(3dce)**, **dce_db_header_fetch(3dce)**.

dce_aud_close

Purpose

Closes an audit trail file. Used by client/server applications and audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_close(
     dce_aud_trail_t at
     unsigned32 *status);
```

Parameters

Input

at A pointer to an audit trail descriptor returned by a previous call to **dce_aud_open()**.

Output

status The status code returned by this routine.

Description

The **dce_aud_close()** function releases data structures of file openings, RPC bindings, and other memory associated with the audit trail that is specified by the audit trail descriptor.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_open(3sec)**.

dce_aud_commit

Purpose

Writes the audit record in the audit trail file. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_commit(
     dce_aud_trail_t at
     dce_aud_rec_t ard
     unsigned32 options
     unsigned16 format
     unsigned32 outcome
     unsigned32* status);
```

Parameters

Input

at Designates an audit trail file to which the completed audit record will be written. The audit trail file must have been previously opened by a successful call to the **dce_aud_open()** function.

ard Designates an audit record descriptor that was returned by a previously successful call to one of the **dce_aud_start_ *()** functions. The content of this record buffer will be appended to the audit trail specified by *at*.

options

Bitwise **OR** of option values described below. A value of 0 (zero) for *options* results in the default operation (normal writing to the file without flushing to stable storage). The possible option value is

aud_c_evt_commit_sync
Flushes the audit record to stable storage before the function returns.

aud_c_evt_always_log
Unconditionally logs the audit record to the audit trail.

aud_c_evt_always_alarm
Unconditionally displays the audit record on the console.

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accommodate changes in the formats of the event specific information, or use different formats dynamically.

outcome

The event outcome to be stored in the header. The possible event-outcome values are as follows:

aud_c_esl_cond_success
The event completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is not known. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

status Returns the status code from this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_commit()** function determines whether the event should be audited given the event outcome. If it should be audited, the function completes the audit record identified by **ard** and writes it to the audit trail designated by **at**. If any of the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** options is selected, the event is always audited (logged or an alarm message is sent to the standard output).

If the **aud_c_evt_commit_sync** option is selected, the function attempts to flush the audit record to stable storage. If the stable storage write cannot be performed, the function either continues to try until the stable-storage write is completed or returns an error status.

Upon successful completion, **dce_aud_commit()** calls **dce_aud_discard()** internally to release the memory of the audit record that is being committed.

The caller should not change the outcome between the **dce_aud_start()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_wrong_protection_level
Client used the wrong protection level.

aud_s_dmn_disabled
The daemon is disabled for logging.

aud_s_log_access_denied
The client's access to the Audit log was denied.

aud_s_cannot_gettime
The audit library cannot backup a trail file due to failure of the **utc_gettime()** call.

aud_s_cannot_getgmtime
The audit library cannot backup a trail file due to failure of the **utc_gmtime()** call.

aud_s_rename_trail_file_rc
Cannot rename the audit trail file.

aud_s_cannot_reopen_trail_file_rc
Internally, the audit trail file was being reopened and the reopening of the file failed.

aud_s_rename_trail_index_file_rc
Internally, the audit trail index file was being renamed and the renaming of the file failed.

aud_s_cannot_reopen_trail_index_file_rc
Internally, the audit trail index file was being reopened and the reopening of the file failed.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_invalid_outcome
The event outcome parameter that was provided is invalid.

aud_s_outcomes_inconsistent
The event outcome parameter is inconsistent with the outcome parameter provided in the **dce_aud_start()** call.

aud_s_trl_write_failure
The audit record cannot be written to stable storage.

aud_s_ok
The call was successful.

Status codes passed from dce_aud_discard()

Status codes passed from rpc_binding_inq_auth_caller()

Status codes passed from dce_acl_is_client_authorized()

Status codes passed from audit_pickle_dencode_ev_info() (RPC idl compiler)

Related Information

Functions: **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_discard

Purpose

Discards an audit record (releases the memory). Used by client/server applications and trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_discard(
     dce_aud_rec_t ard
     unsigned32* status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to one of the **dce_aud_start_*()** functions or the **dce_aud_next()** function.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_discard()** function releases the memory used by the audit record descriptor and the associated audit record that is to be discarded.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from dce_aud_free_header()

Related Information

Functions: **dce_aud_open(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_free_ev_info

Purpose

Frees the memory allocated for an event information stucture returned from calling **dce_aud_get_ev_info()**. Used by the audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_free_ev_info(
     dce_aud_ev_info_t *event_info
     unsigned32 *status);
```

Parameters

Input

event_info
Designates an event-specific information item returned from a previous successful call to the **dce_aud_get_ev_info()** function.

Output

status The status code returned by this routine.

Description

The **dce_aud_free_ev_info()** function frees the memory allocated for an event information stucture returned by a previous successful call to the **dce_aud_get_ev_info()** function.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_get_ev_info(3sec)**, **dce_aud_next(3sec)**.

dce_aud_free_header

Purpose

Frees the memory allocated to a designated audit record header structure. Used by the audit trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

void  dce_aud_free_header(
     dce_aud_hdr_t *header
     unsigned32 *status);
```

Parameters

Input

ard Designates a pointer to an audit record header structure that was returned by a previous successful call to the **dce_aud_get_header()** function.

Output

status The status code returned by this routine.

Description

The **dce_aud_free_header()** frees the memory allocated to a designated audit record header structure. The designated audit record header is usually obtained from an audit record by calling **dce_aud_get_header()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_get_header(3sec)**, **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_get_ev_info

Purpose

Returns a pointer to an event information stucture (**dce_aud_ev_info_t**). Used by the audit trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

void  dce_aud_get_ev_info(
     dce_aud_rec_t ard
     dce_aud_ev_info_t **event_info
     unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to the **dce_aud_next()** function.

Output

event_info
Returns an event-specific information item of the designated audit record. Returns NULL if there are no more information items.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_get_ev_info()** function returns a pointer to an event information structure. The designated record is usually obtained from an audit trail by calling **dce_aud_open()** and **dce_aud_next()**. If there is more than one item of event-specific information in the audit record, then one item is returned through one call to **dce_aud_get_ev_info()**. The order in which the items are returned is the same as the order in which they were included in the audit record through **dce_aud_put_ev_info()** calls. This function allocates the memory to hold the human-readable representation of the audit record and returns the address of this memory.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_get_header

Purpose

Gets the header of a specified audit record. Used by the audit trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_get_header(
     dce_aud_rec_t ard
     dce_aud_hdr_t **header
     unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to the **dce_aud_next()** function.

Output

header
Returns the header information of the designated audit record.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_get_header()** function gets the header information of a designated audit record. The designated record is usually obtained from an audit trail by calling **dce_aud_open()** and **dce_aud_next()**.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**.

dce_aud_length

Purpose

Gets the length of a specified audit record. Used by client/server applications and trail analysis and examination tools

Synopsis

```
#include <dce/audit.h>

unsigned32  dce_aud_length(
     dce_aud_rec_t ard
     unsigned32 *status);
```

Parameters

Input

ard Designates an audit record descriptor that was returned by a previously successful call to **dce_aud_next()**, or one of the **dce_aud_start_ *()** functions.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_length()** function gets the length of a designated audit record. The designated record (in binary format) may be obtained from an audit trail by calling the **dce_aud_open()** and **dce_aud_next()** functions.

Applications can use this function to know how much space an audit record will use before it is committed. This function can also be used by audit trail analysis and examination tools to determine the space that a previously committed audit record uses before it is read.

Return Values

The size of the specified audit record in number of bytes.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_ok
The call was successful.

Status codes passed from idl_es_encode_dyn_buffer()

Status codes passed from audit_pickle_dencode_ev_info()
(RPC IDL compiler)

Status codes passed from idl_es_handle_free()

Status codes passed from rpc_sm_client_free()

Related Information

Functions: **dce_aud_next(3aud)**, **dce_aud_open(3aud)**, **dce_aud_put_ev_info(3aud)**, **dce_aud_start(3aud)**, **dce_aud_start_with_name(3aud)**, **dce_aud_start_with_pac(3aud)**, **dce_aud_start_with_server_binding(3aud)**.

dce_aud_next

Purpose

Reads the next audit record from a specified audit trail file into a buffer. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_next(
     dce_aud_trail_t *at
     char *predicate
     unsigned16 format
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

predicate

Criteria for selecting the audit records that are to be read from the audit trail file. A predicate statement consists of an attribute and its value, separated by any of the following operators: **=** (equal to), **<** (less than), **<=** (less than or equal to), **>** (greater than), and **>=** (greater than or equal to):

- *attribute***=** *value*
- *attribute***>** *value*
- *attribute***>=** *value*
- *attribute***<** *value*
- *attribute***<=** *value*

Attribute names are case sensitive, and no space is allowed within a predicate expression. Multiple predicates are delimited by a comma, in the following form:

```
attribute1= value1, attribute2> value2, ...
```

No space is allowed between predicates. Note that when multiple predicates are defined, the values are logically ANDed together.

The possible attribute names, their values, and allowable operators are as follows:

SERVER

The UUID of the server principal that generated the record. The attribute value must be a UUID string. Operator allowed: **=** (equal to).

EVENT

The audit event number. The attribute value must be a hexadecimal number. Operator allowed: **=** (equal to).

OUTCOME
The event outcome of the record. The possible attribute values are **SUCCESS**, **FAILURE**, **PENDING**, or **DENIAL**. Operator allowed: **=** (equal to).

STATUS
The authorization status of the client. The possible attribute values are **DCE** for DCE authorization (PAC based), and **NAME** for name-based authorization. Operator allowed: **=** (equal to).

CLIENT
The UUID of the client principal. The attribute value must be a UUID string. Operator allowed: **=** (equal to).

TIME The time the record was generated. The attribute value must be a null-terminated string that expresses an absolute time. Operators allowed: **<=** (less than or equal to), **<** (less than), **>=** (greater than or equal to), and **>** (greater than).

CELL The UUID of the client's cell. The attribute value must be a UUID string. Operator allowed: **=** (equal to).

GROUP
The UUID of one of the client's group(s). The attribute value must be a UUID string. Operator allowed: **=** (equal to).

ADDR The address of the client. The attribute is typically the string representation of an RPC binding handle. Operator allowed: **=** (equal to).

FORMAT
The format version number of the audit event record. The attribute value must be an integer. Operators allowed: **=** (equal to), **<** (less than), and **>** (greater than).

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accomodate changes in the formats of the event specification information, or use different formats dynamically.

Output

ard A pointer to the audit record descriptor containing the returned record.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. See ''Errors'' for a list of the possible status codes and their meanings.

Description

The **dce_aud_next()** function attempts to read the next record from the audit trail file specified by the audit trail descriptor, *at*. This function also defines the predicate to be used to search for the next record and returns a matching record if one exists. The **dce_aud_next()** function can be used to search for successive records in the trail that match the defined predicate. By default, if no predicate is explicitly defined, the function returns the next record from the audit trail.

If no record satisfies the predicate specified for the call, a value of zero (**NULL**) is returned through *ard*.

The value returned through **ard** can be supplied as an input parameter to the functions **dce_aud_get_header()**, **dce_aud_length()**, **dce_aud_discard()**, **dce_aud_print()**, **dce_aud_get_event()**, and **dce_aud_get_ev_info()**.

Storage allocated by this function must be explicitly freed by a call to **dce_aud_discard()** with *ard* as the input parameter.

If the function successfully reads an audit trail record, the cursor associated with the audit trail descriptor *at* will be advanced to the next record in the audit trail. The calling routine does not need to set or move the cursor explicitly.

If no appropriate record can be found in the audit trail, an *ard* value of **NULL** is returned and the cursor is advanced to the end of the audit trail. If a call is unsuccessful, the position of the cursor does not change.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successfully completed.

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid.

aud_s_trail_file_corrupted
The trail file is corrupted.

aud_s_index_file_corrupted
The index trail file is corrupted.

aud_s_cannot_allocate_memory
The **malloc()** call failed.

Status codes passed from idl_es_decode_buffer()

Status codes passed from idl_es_handle_free()

Status codes passed from audit_pickle_dencode_ev_info()
(RPC IDL compiler)

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_get_header(3sec)**, **dce_aud_length(3sec)**, **dce_aud_get_ev_info(3sec)**, **dce_aud_open(3sec)**, **dce_aud_discard(3sec)**, **dce_aud_print(3sec)**, **dce_aud_get_event(3sec)**.

dce_aud_open

Purpose

Opens a specified audit trail file for read or write. Used by client/server applications and trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_open(
     unsigned32 flags
     char *description
     unsigned32 first_evt_number
     unsigned32 num_of_evts
     dce_aud_trail_t *at
     unsigned32 *status);
```

Parameters

Input

flags Specifies the mode of opening. The flags parameter is set to the bitwise OR of the following values:

- **aud_c_trl_open_read**
- **aud_c_trl_open_write**
- **aud_c_trl_ss_wrap**

description
A character string specifying an audit trail file to be opened. If **description** is NULL, the default audit trail file is opened. When the audit trail file is opened for write, the default audit trail is an RPC interface to a local audit daemon.

first_evt_num
The lowest assigned audit event number used by the calling server.

num_of_evts
The number of audit events defined for the calling server.

Output

at A pointer to an audit trail descriptor. When the audit trail descriptor is no longer needed, it must be released by calling the **dce_aud_close()** function.

status Returns the status code from this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_open()** function opens the audit trail file specified by the **description** parameter. If **description** is NULL, the function uses the default audit trail which is an RPC interface to the local audit daemon.

This function must be invoked after the server has finished registering with RPC and before calling **rpc_server_listen()**.

If the **flags** parameter is set to **aud_c_trl_open_read**, the specified file (**description** cannot be null in this case) is opened for reading audit records, using the **dce_aud_next()** function. If **flags** is set to **aud_c_trl_open_write**, the specified file or the default audit trail device is opened and initialized for appending audit records using the **dce_aud_commit()** function. Only one of the **aud_c_trl_open_read** and **aud_c_trl_open_write** flags may be specified in any call to **dce_aud_open()**. If the **flags** parameter is set to **aud_c_trl_ss_wrap**, the audit trail operation is set to **wrap** mode. The **aud_c_trl_ss_wrap** flag has meaning only if you specify the **aud_c_trl_open_write** flag.

If the audit trail specified is a file and the calling server does not have the read and write permissions to the file, a NULL pointer is returned in **at**, and **status** is set to **aud_s_cannot_open_trail_file_rc**. The same values will be returned if the default audit trail file is used (that is, through an audit daemon) and if the calling server is not authorized to use the audit daemon to log records.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

aud_s_trl_invalid_open_flags
The flags argument must include either **aud_c_trl_open_read** or **aud_c_trl_open_write** flag, but not both.

aud_s_cannot_open_dmn_binding_file
The local audit daemon trail file is designated, but the daemon's binding file cannot be opened.

Status codes passed from sec_login_get_current_context()
When the local audit daemon trail file is designated, a login context is needed for making secure audit logging RPC to the audit daemon.

aud_s_cannot_open_dmn_identity_file
The local audit daemon trail file is designated, but the daemon's identity file cannot be opened.

Status codes passed from rpc_binding_set_auth_info()
When the local audit daemon trail file is designated, **dce_aud_open()** sets authentication information in the RPC binding handle for making secure audit logging RPC to the audit daemon. This is done by calling **rpc_binding_set_auth_info()**.

aud_s_cannot_open_trail_file_rc
Cannot open a local trail file.

aud_s_cannot_allocate_memory
Memory allocation failed.

aud_s_cannot_init_trail_mutex
Audit trail mutex initialization failed.

Status codes passed from rpc_server_inq_bindings()
When filtering is turned on, **dce_aud_open()** gets the caller's RPC bindings to be used for registering an RPC interface in receiving filter update notification from the local audit daemon. This is done by calling **rpc_server_inq_bindings()**.

Status codes passed from rpc_binding_to_string_binding()
When filtering is turned on, the caller's RPC bindings are converted to string bindings before they are stored in a file. This is done by calling **rpc_binding_to_string_binding()**.

aud_s_cannot_mkdir
Cannot create a directory for storing the bindings file for the filter update notification interface.

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_next(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_prev

Purpose

Reads the previous audit record from a specified audit trail file into a buffer. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_prev(
     dce_aud_trail_t* at
     char *predicate
     unsigned16 format
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

predicate

Criteria for selecting the audit records that are to be read from the audit trail file. A predicate statement consists of an attribute and its value, separated by any of the following operators: **=** (equal to), **<** (less than), <= (less than or equal to), **>** (greater than), and >= (greater than or equal to).

- *attribute***=** *value*
- *attribute***>** *value*
- *attribute***>=** *value*
- *attribute***<** *value*
- *attribute***<=** *value*

Attribute names are case sensitive, and no space is allowed within a predicate expression. Multiple predicates are delimited by a comma, in the following form:

```
attribute= value1, attribute> value2,  ...
```

No space is allowed between predicates. Note that when multiple predicates are defined, the values are logically ANDed together.

The possible attribute names, their values, and allowable operators are as follows:

SERVER

The UUID of the server principal that generated the record. The attribute value must be a UUID string. Operator allowed: **=** (equal to).

EVENT

The audit event number. The attribute value must be a hexadecimal number. Operator allowed: **=** (equal to).

OUTCOME
The event outcome of the record. The possible attribute values are: **SUCCESS**, **FAILURE**, **PENDING**, or **DENIAL**. Operator allowed: **=** (equal to).

STATUS
The authorization status of the client. The possible attribute values are **DCE** for DCE authorization (PAC based) and **NAME** for name-based authorization. Operator allowed: **=** (equal to).

TIME The time the record was generated. The attribute value must be a null terminated string that expresses an absolute time. Operators allowed: **<=** (less than or equal to), **<** (less than), **>=** (greater than or equal to), and **>** (greater than).

CELL The UUID of the client's cell. The attribute value must be a UUID string. Operator allowed: **=** (equal to).

GROUP
The UUID of one of the client's group(s). The attribute value must be a UUID string. Operator allowed: **=** (equal to).

ADDR The address of the client. The attribute is typically the string representation of an RPC binding handle. Operator allowed: **=** (equal to).

FORMAT
The format version number of the audit event record. The attribute value must be an integer. Operators allowed: **=** (equal to), **<** (less than), and **>** (greater than).

format Event's tail format used for the event-specific information. This format can be configured by the user. With this format version number, the servers and audit analysis tools can accommodate changes in the formats of the event specification information, or use different formats dynamically.

Output

ard A pointer to the audit record descriptor containing the returned record.

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. See ''Errors'' for a list of the possible status codes and their meanings.

Description

The **dce_aud_prev()** function attempts to read the previous record from the audit trail file specified by the audit trail descriptor, *at*. This function also defines the predicate to be used to search for the previous record and returns a matching record if one exists. **dce_aud_prev()** can be used to search for previous records in the trail file that match the defined predicate. By default, if no predicate is explicitly defined, the function returns the previous record read from the audit trail.

If no record satisfies the predicate specified for the call, a value of zero (**NULL**) is returned in *ard*.

The value returned in *ard* can be supplied as an input parameter to the functions: **dce_aud_get_header()**, **dce_aud_length()**, **dce_aud_discard()**, **dce_aud_print()**, **dce_aud_get_event()**, and **dce_aud_get_ev_info()**.

Storage allocated by this function must be explicitly freed by a call to **dce_aud_discard()** with *ard* as the input parameter.

If the function successfully reads an audit trail record, the cursor associated with the audit trail descriptor *at* will be moved to the previous record in the audit trail file. The calling routine does not need to set or move the file cursor explicitly.

If no appropriate record can be found in the audit trail, an *ard* value of **NULL** is returned and the cursor is set back to the beginning of the audit trail. If a call is unsuccessful, the position of the cursor does not change.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successfully completed

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid

aud_s_trail_file_corrupted
The audit trail is corrupted

aud_s_index_file_corrupted
The index trail file is corrupted

aud_s_cannot_allocate_memory
The **malloc()** call failed

Status codes passed from idl_es_decode_buffer()

Status codes passed from idl_es_handle_free()

Status codes passed from audit_pickle_dencode_ev_info()
(RPC IDL compiler)

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_get_header(3sec)**, **dce_aud_length(3sec)**, **dce_aud_get_ev_info(3sec)**, **dce_aud_open(3sec)**, **dce_aud_discard(3sec)**, **dce_aud_print(3sec)**, **dce_aud_get_event(3sec)**.

dce_aud_print

Purpose

Formats an audit record into human-readable form. Used by audit trail examination and analysis tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_print(
     dce_aud_rec_t ard
     unsigned32 options
     char **buffer
     unsigned32 *status);
```

Parameters

Input

ard An audit record descriptor. This descriptor can be obtained from an opened audit trail by calling **dce_aud_next()** or it can be a new record established by calling one of the **dce_aud_start_ *()** functions.

options

The options governing the transformation of the binary audit record information into a character string. The value of the *options* parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all the optional information (that is, groups, address, and event specific information).

aud_c_evt_groups_info
Includes the groups' information.

aud_c_evt_address_info
Includes the address information.

aud_c_evt_specific_info
Includes the event specific information.

Output

buffer Returns the pointer to a character string converted from the audit record specified by *ard*.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_print()** function transforms the audit record specified by *ard* into a character string and places it in a buffer. The buffer is allocated using **malloc()**, and must later be freed by the caller. (This function allocates the memory to hold the human-readable text of the audit record and returns the address of this memory in the *buffer* parameter.)

The *options* parameter is set to the bitwise OR of flag values defined in the **dce/audit.h** header file. A value of 0 (zero) for options will result in default operation, that is, no group, address, and event-specific information is included in the output string.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The audit record descriptor is invalid.

aud_s_cannot_allocate_memory
The **malloc()** call failed.

aud_s_ok
The call was successful.

Status codes passed from sec_login_get_current_context()

Status codes passed from sec_login_inquire_net_info()

Related Information

Functions: **dce_aud_next(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_put_ev_info

Purpose

Adds event-specific information to a specified audit record buffer. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_put_ev_info(
     dce_aud_rec_t ard
     dce_aud_ev_info_t info
     unsigned32 *status);
```

Parameters

Input

ard A pointer to an audit record descriptor initialized by one of the **dce_aud_start_ *()** functions.

info A data structure containing an event-specific information item that is to be appended to the tail of the audit record identified by **ard**. The possible formats of the event-specific information are listed in the **sec_intro(3sec)** reference page of this book.

Output

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_put_ev_info()** function adds event-specific information to an audit record. The event-specific information is included in an audit record by calling **dce_aud_put_ev_info()** one or more times. The order of the information items included by multiple calls is preserved in the audit record, so that they may be read in the same order by the **dce_aud_get_ev_info()** function. This order is also observed by the **dce_aud_print()** function. The **info** parameter is a pointer to an instance of the self-descriptive **dce_aud_ev_info_t** structure.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_record_descriptor
The input audit record descriptor is invalid.

aud_s_evt_tail_info_exceeds_limit
The tail portion of the audit trail record has exceeded its limit of 4K.

aud_s_ok
The call was successful.

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_reset

Purpose

Resets the cursors and the file pointers of the specified audit trail file. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_reset(
     dce_aud_trail_t *at
     unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened by the function **dce_aud_open()**.

Output

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. For a list of the possible status codes and their meanings, see ''Errors''.

Description

The **dce_aud_reset()** function resets the cursors and the file pointers of the specified audit trail file. The function is used to explicitly reset the current cursors and file pointers to the beginning of the audit trail file.

dce_aud_open() must be called to specify the desired audit trail file. Otherwise, **dce_aud_reset()** will reset the audit trail which is currently set in the value of *at*.

If the call is successful, the file cursors are set to the beginning of the file.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages. The possible status codes and their meanings are:

aud_s_ok
The call was successful

aud_s_invalid_trail_descriptor
The audit trail descriptor is invalid

Related Information

Functions: **dce_aud_rewind(3sec)**, **dce_aud_clean(3sec)**, **dce_aud_open(3sec)**.

dce_aud_rewind

Purpose

Rewinds the specified audit trail file. Used by the trail analysis and examination tools.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_rewind(
     dce_aud_trail_t *at
     unsigned32 *status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for writing by the function **dce_aud_open()**.

Output

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given. For a list of the possible status codes and their meanings, see ''Errors''.

Description

The **dce_aud_rewind()** function rewinds the specified audit trail file. This function can be used to instantly clean up the audit trail file if it is no longer needed.

dce_aud_open() must be called to specify the desired audit trail file, and the specified audit trail file must be opened with the **aud_c_trl_open_write** flag. Otherwise, the routine will rewind the audit trail which is currently set in the value of *at*.

If the call is successful, the file cursors are set to the beginning of the file.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

aud_s_invalid_trail_descriptor
The Audit Trail descriptor is invalid

aud_s_trl_invalid_open_flag
The Audit Trail is opened with open flag

aud_s_rewind_trail_file
The ftruncate() call failed on trail file

aud_s_rewind_index_file
The ftruncate() call failed on index file

Related Information

Functions: **dce_aud_clean(3sec)**, **dce_aud_open(3sec)**.

dce_aud_set_trail_size_limit

Purpose

Sets a limit to the audit trail size. Used by client/server applications.

Synopsis

```
#include <dce/audit.h>

void  dce_aud_set_trail_size_limit(
     dce_aud_trail_t at
     unsigned32 file_size_limit_value
     unsigned32 * status);
```

Parameters

Input

at A pointer to the descriptor of an audit trail file previously opened for reading by the function **dce_aud_open()**.

file_size_limit_value
The desired maximum size of the audit trail file, in bytes.

Output

status Returns the status code of this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_set_trail_size_limit()** function can be used by an application that links with **libaudit** to set the maximum size of the audit trail. This function must be called immediately after calling **dce_aud_open()**.

For added flexibility, the environment variable DCEAUDITTRAILSIZE can also be used to set the maximum trail size limit.

If none of these methods are used for setting the trail size, then a hardcoded limit of 2 megabytes will be assumed.

If set, the value of the environment variable DCEAUDITTRAILSIZE overrides the value set by this function. Any of the values set by DCEAUDITTRAILSIZE or this function overrides the hardcoded default.

When the size limit is reached, the current trail file is copied to another file. The name of this new file is the original filename appended by a timestamp. For example, if the name of the original trail file is **central_trail**, its companion trail file is named **central_trail.md_index**. These two files will be copied to the following locations:

```
central_trail.1994-09-26-16-38-15
central_trail.1994-09-26-16-38-15.md_index
```

When a trail file is copied to a new file by the audit library because it has reached the size limit, a serviceability message is issued to the console notifying the user that an audit trail file (and its companion index file) is available to be backed up. Once the backup is performed, it is advisable to remove the old trail file, so as to prevent running out of disk space.

Auditing will then continue, using the original name of the file, (in our example, **central_trail**).

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_invalid_trail_descriptor
The audit trail descriptor *at* is null.

aud_s_ok
The call is successful.

Related Information

Functions: **dce_aud_open(3sec)**.

dce_aud_start

Purpose

Determines whether a specified event should be audited given the client binding information and the event outcome. Used by client/server applications

Synopsis

```
#include <dce/audit.h>

void  dce_aud_start(
     unsigned32 event
     rpc_binding_handle_t binding
     unsigned32 options
     unsigned32 outcome
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

binding

Specifies the client's RPC binding handle from which the client identification information is retrieved to set the *client*, *cell*, *num_groups*, *groups*, and *addr* fields in the audit record header.

options

Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, or **aud_c_evt_address_info**).

It can also be used to specify whether the audit records are always logged (**aud_c_evt_always_log**) or that an alarm message is always sent to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome
The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable DCEAUDITOFF has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* other than **aud_c_esl_cond_unknown** must be provided when calling the **dce_aud_commit()** function.

status The status code returned by this function. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start()** function determines if an audit record should be generated for the specified event. The decision is based on the event filters, an environment variable (DCEAUDITOFF), the client's identity provided in the **binding** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, *ard*). If the event does not need to be audited, a NULL *ard* is returned. If an internal error(s) has occurred, a NULL pointer is returned in *ard*. If the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start()** function is designed to be used by RPC applications. Non-RPC applications that use the DCE authorization model (that is, DCE ACL and PAC) must use **dce_aud_start_with_pac()**. Non-RPC applications that do not use the DCE authorization model must use **dce_aud_start_with_name()**.

This function obtains the client identity information from the RPC binding handle and records it in the newly-created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. This function can be called multiple times after calling **dce_aud_start()** and before calling **dce_aud_commit()**. A completed audit record will be appended to an audit trail file or sent to the audit daemon (depending on the value of the **description** parameter used in the previous call to **dce_aud_open**) by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start()** call, **dce_aud_start()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from rpc_binding_to_string_binding()

Status codes passed from rpc_string_free()

Status codes passed from dce_aud_start_with_name()

Status codes passed from sec_cred_get_initiator()

Status codes passed from sec_cred_get_v1_pac()

Status codes passed from dce_aud_start_with_pac()

Status codes passed from sec_cred_get_delegate()

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_name

Purpose

Determines whether a specified event should be audited given the client/server name and the event outcome. Used by non-RPC based client/server applications that do not use the DCE authorization model

Synopsis

```
#include <dce/audit.h>

void  dce_aud_start_with_name(
     unsigned32event
     unsigned_char_t *client
     unsigned_char_t *address
     unsigned32 options
     unsigned32 outcome
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

client Specifies the principal name of the remote client/server.

address
Specifies the address of the remote client/server. The address could be in any format of the underlying transport protocol.

options
Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**).

It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed. The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome
The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters or if the environment variable DCEAUDITOFF has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, the function may not be able to determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_name()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (DCEAUDITOFF), the client's identity provided in the input parameters, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_name()** function is designed to be used by non-RPC applications that do not use the DCE authorization model (that is, DCE PAC and ACL). RPC applications must use **dce_aud_start()**. Non-RPC applications that use the DCE authorization model must use **dce_aud_start_with_pac()**.

This function records the input identity parameters in the newly created audit record descriptor.

Event-specific information can be added to the record by using the **dce_aud_put_ev_info()** function, which can be called multiple times after calling any of the **dce_aud_start_** * and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_name()** call, **dce_aud_start_with_name()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_name()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_name()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_name()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from sec_rgy_site_open()

Status codes passed from sec_id_parse_name()

Status codes passed from dce_aud_start_with_pac()

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_pac

Purpose

Determines whether a specified event must be audited given the client's privilege attribute certificate (PAC) and the event outcome. Used by non-RPC based client/server applications that use the DCE authorization model

Synopsis

```
#include <dce/audit.h>

void  dce_aud_start_with_pac(
     unsigned32 event
     sec_id_pac_t *pac
     unsigned_char_t *address
     unsigned32 options
     unsigned32 outcome
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

pac Specifies the client's PAC from which the client's identification information is retrieved to set the *client*, *cell*, *num_groups*, and *groups* fields in the audit record header.

address
Specifies the client's address. The address can be in any format that is native to the underlying transport protocol.

options
Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**). It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups' information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter and indicates that an alarm message must be sent to the system console for the event.

outcome
The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable DCEAUDITOFF has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_pac()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (DCEAUDITOFF), the client's identity provided in the **pac** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, then an audit record descriptor will always be created and returned.

The **dce_aud_start_with_pac()** function is designed to be used by non-RPC applications that use the DCE authorization model (that is, DCE PAC and ACL). RPC applications must use **dce_aud_start()**. Non-RPC applications that do not use the DCE authorization model must use **dce_aud_start_with_name()**.

This function obtains the client's identity information from the client's privilege attribute certificate (PAC) and records it in the newly created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. This function can be called multiple times after calling any of the **dce_aud_start_** * functions and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling the **dce_aud_commit()** function.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_pac()** call, **dce_aud_start_with_pac()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_pac()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_pac()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_pac()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from sec_rgy_site_open()

Status codes passed from sec_rgy_properties_get_info()

Status codes passed from uuid_create_nil()

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

dce_aud_start_with_server_binding

Purpose

Determines whether a specified event must be audited given the server binding information and the event outcome. Used by client/server applications

Synopsis

```
#include <dce/audit.h>

void  dce_aud_start_with_server_binding(
     unsigned32 event
     rpc_binding_handle_t binding
     unsigned32 options
     unsigned32 outcome
     dce_aud_rec_t *ard
     unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

binding
Specifies the server's RPC binding handle from which the server identification information is retrieved to set the client, cell, and addr fields in the audit record header. Note that when an application client issues an audit record, the server identity is represented in the *client* field of the record.

options
This parameter can be used to specify the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**). It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed.

The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_address_info
Includes the server address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter and indicates that an alarm message must be sent to the system console for the event.

outcome
The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_success
The event was completed successfully.

aud_c_esl_cond_denial
The event failed because of access denial.

aud_c_esl_cond_failure
The event failed because of reasons other than access denial.

aud_c_esl_cond_pending
The event is in an intermediate state, and the outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

aud_c_esl_cond_unknown
The event outcome (denial, failure, pending, or success) is still unknown. This outcome exists only between a **dce_aud_start()** (all varieties of this routine) call and the next **dce_aud_commit()** call. You can also use **0** to specify this outcome.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable DCEAUDITOFF has been set, a NULL pointer is returned. If the function is called with **outcome** set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome* must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine was completed successfully or not. If the routine was not completed successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_server_binding()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (DCEAUDITOFF), the server's identity provided in the **binding** parameter, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_server_binding()** function is designed to be used by RPC applications. Non-RPC applications that use the DCE authorization model must use the **dce_aud_start_with_pac()** function. Non-RPC applications that do not use the DCE authorization model must use the **dce_aud_start_with_name()** function.

This function obtains the server identity information from the RPC binding handle and records it in the newly created audit record descriptor.

Event-specific information can be added to the record by calling the **dce_aud_put_ev_info()** function. The **dce_aud_put_ev_info()** function can be called multiple times after calling any of the **dce_aud_start_ *** functions and before

calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_server_binding()** call, **dce_aud_start_with_server_binding()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_server_binding()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of 0 (zero) or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_server_binding()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_server_binding()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from rpc_binding_inq_auth_info()

Status codes passed from rpc_binding_to_string_binding()

Status codes passed from dce_aud_start_with_name()

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**.

dce_aud_start_with_uuid

Purpose

Determines whether a specified event should be audited given the client/server UUID and the event outcome. Used by client/server applications which already know the UUIDs of their clients and wish to avoid the overhead of the audit library acquiring them

Synopsis

```
#include <dce/audit.h>

void  dce_aud_start_with_uuid(
     unsigned32 event
     uuid_t server_uuid
     uuid_t client_uuid
     uuid_ trealm_uuid
     unsigned_char_t *address
     unsigned3 2options
     unsigned32 outcome
     dce_aud_rec_t * ard
     unsigned32 *status);
```

Parameters

Input

event Specifies the event to be audited. This is a 32-bit event number. The *event* field in the audit record header will be set to this number.

server_uuid
Specifies the calling application's principal uuid.

client_uuid
Specifies the remote client/server's principal uuid.

realm_uuid
Specifies the remote client/server's cell uuid.

address
Specifies the remote client/server's address. The address could be in any format of the underlying transport protocol.

options
Specifies the optional header information desired (**aud_c_evt_all_info**, **aud_c_evt_group_info**, **aud_c_evt_address_info**).

It can also be used to specify any of two options: to always log an audit record (**aud_c_evt_always_log**) or to always send an alarm message to the standard output (**aud_c_evt_always_alarm**). If any of these two options is selected, the filter is bypassed. The value of the **options** parameter is the bitwise OR of any selected combination of the following option values:

aud_c_evt_all_info
Includes all optional information (groups and address) in the audit record header.

aud_c_evt_groups_info
Includes the groups information in the audit record header.

aud_c_evt_address_info
Includes the client address information in the audit record header.

aud_c_evt_always_log
Bypasses the filter mechanism and indicates that the event must be logged.

aud_c_evt_always_alarm
Bypasses the filter mechanism and indicates that an alarm message must be sent to the system console for the event.

outcome
The event outcome to be stored in the header. The following event outcome values are defined:

aud_c_esl_cond_unknown
The event outcome (denial, failure, or success) is still unknown.

aud_c_esl_cond_success
The event completed successfully.

aud_c_esl_cond_denial
The event failed due to access denial.

aud_c_esl_cond_failure
The event failed due to reasons other than access denial.

aud_c_esl_cond_pending
The event outcome is pending, being one in a series of connected events, where the application desires to record the real outcome only after the last event.

Output

ard Returns a pointer to an audit record buffer. If the event does not need to be audited because it is not selected by the filters, or if the environment variable DCEAUDITOFF has been set, a NULL pointer is returned. If the function is called with *outcome* set to **aud_c_esl_cond_unknown**, it is possible that the function cannot determine whether the event should be audited. In this case, the audit record descriptor is still allocated and its address is returned to the caller. An *outcome*, different from **unknown**, must be provided prior to logging the record with the **dce_aud_commit()** function.

status The status code returned by this routine. This status code indicates whether the routine completed successfully or not. If the routine did not complete successfully, the reason for the failure is given.

Description

The **dce_aud_start_with_uuid()** function determines if an audit record must be generated for the specified event. The decision is based on the event filters, an environment variable (DCEAUDITOFF), the client's identity provided in the input parameters, and the event outcome (if it is provided in the **outcome** parameter). If this event needs to be audited, the function allocates an audit record descriptor and returns a pointer to it, (that is, **ard**). If the event does not need to be audited, NULL is returned in the *ard* parameter. If either the **aud_c_evt_always_log** or **aud_c_evt_always_alarm** option is selected, an audit record descriptor will always be created and returned.

The **dce_aud_start_with_uuid()** function is designed to be used by RPC applications that know their client's identity in UUID form. Otherwise, RPC applications should use **dce_aud_start()**. Non-RPC applications that use the DCE authorization model should use **dce_aud_start_with_pac()**. The **dce_aud_start_with_name()** function should be used by non-RPC applications that do not use the DCE authorization model.

This function records the input identity parameters in the newly-created audit record descriptor.

Event-specific information can be added to the record by using the **dce_aud_put_ev_info()** function, which can be called multiple times after calling any of the **dce_aud_start_** * and before calling **dce_aud_commit()**. A completed audit record can either be appended to an audit trail file or sent to the audit daemon by calling **dce_aud_commit()**.

This function searches for all relevant filters (for the specified subject and outcome, if these are specified), summarizes the actions for each possible event outcome, and records an outcome-action table with *ard*. If the outcome is specified when calling this function and the outcome does not require any action according to filters, then this function returns a NULL *ard*.

If the *outcome* is not specified in the **dce_aud_start_with_uuid()** call, **dce_aud_start_with_uuid()** returns a NULL *ard* if no action is required for all possible outcomes.

The caller should not change the outcome between the **dce_aud_start_with_uuid()** and **dce_aud_commit()** calls arbitrarily. In this case, the outcome can be made more specific, for example, from **aud_c_esl_cond_unknown** to **aud_c_esl_cond_success** or from **aud_c_esl_cond_pending** to **aud_c_esl_cond_success**.

An outcome change from **aud_c_esl_cond_success** to **aud_c_esl_cond_denial** is not logically correct because the outcome **aud_c_esl_cond_success** may have caused a NULL *ard* to be returned in this function. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this function, or use **aud_c_esl_cond_unknown**.

This function can be called with the *outcome* parameter taking a value of zero or the union (logical OR) of selected values from the set of constants **aud_c_esl_cond_success**, **aud_c_esl_cond_failure**, **aud_c_esl_cond_denial**, and **aud_c_esl_cond_pending**. The *outcome* parameter used in the **dce_aud_commit()** function should take one value from the same set of constants.

If **dce_aud_start_with_uuid()** used a nonzero value for *outcome*, then the constant used for *outcome* in the **dce_aud_commit()** call should have been selected in the **dce_aud_start_with_uuid()** call.

Return Values

No value is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

aud_s_ok
The call was successful.

Status codes passed from dce_aud_start_with_pac()

Related Information

Functions: **dce_aud_commit(3sec)**, **dce_aud_open(3sec)**, **dce_aud_put_ev_info(3sec)**, **dce_aud_start(3sec)**, **dce_aud_start_with_name(3sec)**, **dce_aud_start_with_pac(3sec)**, **dce_aud_start_with_server_binding(3sec)**.

gss_accept_sec_context

Purpose

Establishes a security context between the application and a context acceptor

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_accept_sec_context(
     OM_uint32 *minor_status
     gss_ctx_id_t *context_handle
     gss_cred_id_t verifier_cred_handle
     gss_buffer_t input_token_buffer
     gss_channel_bindings_t input_chan_bindings
     gss_name_t *src_name
     gss_OID *actual_mech_type
     gss_buffer_t output_token
     int *ret_flags
     OM_uint32 *time_rec
     gss_cred_id_t *delegated_cred_handle);
```

Parameters

Input

verifier_cred_handle
Specifies the credential handle (the identity) claimed by the context acceptor. This is optional information. The credential must be either an **ACCEPT** type credential or a **BOTH** type credential. If you do not specify a credential handle and specify instead **GSS_C_NO_CREDENTIAL**, the application can accept a context under any registered identity. Use the **gssdce_register_acceptor_identity()** routine to register an identity before specifying **GSS_C_NO_CREDENTIAL**.

input_token_buffer
Specifies the token received from the context acceptor.

input_chan_bindings
Specifies bindings supplied by the context acceptor.

Allows the context acceptor to bind the channel identification information securely to the security context.

Input/Output

context_handle
Specifies a context handle for a new context. The first time the context acceptor uses the routine, specify **GSS_C_NO_CONTEXT** to set up a specific context. In subsequent calls, use the value returned by this parameter.

Output

src_name
Returns the authenticated name of the context acceptor. This information is optional. If the authenticated name is not required, specify NULL.

To deallocate the authenticated name, pass it to the **gss_release_name()** routine.

actual_mech_type
Returns the security mechanism with which the context was established. The security mechanism will be one of the following:

- **GSSDCE_C_OID_DCE_KRBV5_DES** (for DCE security)
- **GSSDCE_C_OID_KRBV5_DES** (for Kerberos Version 5)

output_token
Returns a token to pass to the context acceptor. If no token is to be passed to the context acceptor, the routine sets the length field of the returned token buffer to 0 (zero).

ret_flags
Returns a bitmask containing six independent flags, each of which requests that the context support a service option. The following symbolic names are provided to correspond to each flag. The symbolic names should be logically ANDed with the value of *ret_flags* to test whether the context supports the service option.

GSS_C_DELEG_FLAG

True Delegated credentials are available from the *delegated_cred_handle* parameter.

False No credentials were delegated.

GSS_C_MUTUAL_FLAG

True The context acceptor requested mutual authentication.

False The context acceptor did not request mutual authentication.

GSS_C_REPLAY_FLAG

True Replayed signed or sealed messages will be detected.

False Replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG

True Out-of-sequence signed or sealed messages will be detected.

False Out-of-sequence signed or sealed messages will not be detected.

GSS_C_CONF_FLAG

True Confidentiality services are available by calling the **gss_seal()** routine.

False Confidentiality services are not available. However, the application can call the **gss_seal()** routine to provide message encapsulation, data-origin authentication, and integrity services.

GSS_C_INTEG_FLAG

True Integrity services can be invoked by calling either the **gss_sign()** or **gss_seal()** routine.

False Integrity services for individual messages are not available.

time_rec
Returns the number of seconds for which the context remains valid. This is optional information. If the time is not required, specify NULL.

delegated_cred_handle
: Returns the credential handle for credentials received from the context acceptor. The credential handle is valid only if delegated credentials are available. If the *ret_flags* parameter is true, the flag **GSS_C_DELEG_FLAG** is set, indicating that delegated credentials are available.

minor_status
: Returns a status code from the security mechanism.

Description

The **gss_accept_sec_context()** routine is the second step in establishing a security context between the context initiator and a context acceptor. In the first step, the context initiator calls the **gss_init_sec_context()** routine. The **gss_init_sec_context()** routine generates a token for the security context and passes it to the context initiator. The context initiator sends the token to the context acceptor.

In the second step, the context acceptor accepts the call from the context initiator and calls the **gss_accept_sec_context()** routine. The **gss_accept_sec_context()** routine expects a value for the *input_token* parameter. The value for the *input_token* parameter is generated by the **gss_init_sec_context()** routine and passed by the initiator to the acceptor.

The **gss_accept_sec_context()** routine can also return a value for the *output_token* parameter. The context acceptor presents the token to the **gss_init_sec_context()** routine. If the acceptor does not need to send a token to the initiator, **gss_accept_sec_context()** sets the length field of the *output_token* parameter to 0 (zero).

To complete establishing the context, the context initiator can require one or more reply tokens from the context acceptor. If the application requires reply tokens, the **gss_accept_sec_context()** routine returns a status value containing **GSS_S_CONTINUE_NEEDED**. The application calls the routine again when the reply token is received from the context acceptor. The application passes the token to the **gss_accept_sec_context()** routine via the *output_token* parameters.

The **gss_accept_sec_context()** routine must find a key to decrypt the token. The token contains the unencrypted principal name of the context acceptor. The acceptor's principal name identifies the key that the context initiator used to encrypt the rest of the token. The **gss_accept_sec_context()** routine matches the principal name with the key in the following way:

- If you specify a credential, the credential and the name in the token must match. The acceptor's principal name (contained in the token) has been registered by a call to the **gssdce_register_acceptor_identity()** routine. The **gss_accept_sec_context()** routine looks in the registered key table.
- If you specify **GSS_C_NO_CREDENTIAL** and the principal name in the token is registered, the **gss_accept_sec_context()** routine, using either the **rpc_server_register_auth_info()** routine or the **gssdce_register_acceptor_identity()** routine, looks in the table specified when you registered the token name.
- If you specify **GSS_C_NO_CREDENTIAL** and the principal name in the token is not registered, the **gss_accept_sec_context()** routine fails and returns the status **GSS_S_FAILURE** because the Generic Security Service Application Programming Interface (GSSAPI) does not know where to find the key.

The following table summarizes how the **gss_accept_sec_context()** routine determines the key for the credential:

You specify ...	Is the principal's name registered?	Then the routine ...
A credential	Yes	Looks in the key table specified in **gssdce_register_acceptor_identity()** or the default key table.
GSS_C_NO_CREDENDIAL	Yes	Looks in the key table specified in **gssdce_register_acceptor_identity()**.
	No	Fails because the principal is not registered. It returns the status code **GSS_S_FAILURE**.

The values returned using the *src_name*, *ret_flags*, *time_rec*, and *delegated_cred_handle* parameters are not defined unless the routine returns the status **GSS_S_COMPLETE**.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_BINDINGS
The *input_token* parameter contains different channel bindings from those specified with the *input_chan_bindings* parameter.

GSS_S_BAD_SIG
The *input_token* parameter contains an invalid signature.

GSS_S_CONTINUE_NEEDED
To complete the context, the **gss_accept_sec_context( )** routine must be called again with a token required from the context acceptor.

GSS_S_CREDENTIALS_EXPIRED
The referenced credentials have expired.

GSS_S_DEFECTIVE_CREDENTIAL
Consistency checks performed on the credential failed.

GSS_S_DEFECTIVE_TOKEN
Consistency checks performed on the *input_token* parameter failed.

GSS_S_DUPLICATE_TOKEN
The *input_token* parameter was already processed. This is a fatal error that occurs during context establishment.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

GSS_S_NO_CONTEXT
The supplied context handle did not refer to a valid context.

GSS_S_NO_CRED
Indicates either the supplied credentials were not valid for context acceptance or the credential handle did not reference any credentials.

GSS_S_OLD_TOKEN
The *input_token* parameter was too old. This is a fatal error that occurs during context establishment.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gss_delete_sec_context(3sec)**, **gss_init_sec_context(3sec)**, **gssdce_register_acceptor_identity(3sec)**.

gss_acquire_cred

Purpose

Allows an application to acquire a handle for an existing named credential

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_acquire_cred(
     OM_uint32 *minor_status
     gss_name_t desired_name
     OM_uint32 time_req
     gss_OID_set desired_mechs
     int cred_usage
     gss_cred_id_t *output_cred_handle
     gss_OID_set *actual_mechs
     OM_int32 *time_rec);
```

Parameters

Input

desired_name
Specifies the principal name to use for the credential.

time_req
Specifies the number of seconds that credentials remain valid.

desired_mechs
Specifies the object identifier (OID) set for the security mechanism to use with the credential, as follows:

DCE security
Specify **GSS_C_NULL_OID_SET**.

Kerberos
Specify **GSSDCE_C_OID_KRBV5_DES**.

Both DCE security and Kerberos
Specify **GSSDCE_C_OID_DCE_KRBV5_DES** and **GSSDCE_C_OID_KRBV5_DES**.

To help ensure portability of your application, request the default security mechanism by specifying **GSS_C_NULL_OID_SET**.

cred_usage
Specify one of the following:

GSS_C_BOTH
Specifies credentials that the context initiator can use to either initiate or accept security contexts.

GSS_C_ACCEPT
Specifies credentials that the context initiator can use only to accept security contexts.

Output

output_cred_handle
Returns the handle for the return credential.

actual_mechs
Returns a set of mechanisms for which the credential is valid. This information is optional. If you do not want a set of mechanisms returned, specify NULL.

time_rec
Returns the actual number of seconds for which the return credential remains valid. This information is optional. If the actual number of seconds is not required, specify NULL.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_acquire_cred( )** routine allows an application to obtain a handle for either an **ACCEPT** or a **BOTH** credential. The application then passes the credential handle to either the **gss_init_sec_context()** routine or the **gss_accept_sec_context()** routine.

Credential handles created by the **gss_acquire_cred()** routine contain a principal name. If the principal name is unregistered, the **gss_acquire_cred()** routine automatically registers the principal in the default key table. You can change the principal's key table by calling the **gssdce_register_acceptor_identify()** routine.

To create an **INITIATE** credential, you must use the **gssdce_login_context_to_cred()** routine.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_MECH
The requested security mechanism is unsupported or unavailable.

GSS_S_BAD_NAMETYPE
The name passed by the *desired_name* parameter is unsupported.

GSS_S_BAD_NAME
An invalid name was passed by the *desired_name* parameter.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

Related Information

Functions: **gssdce_accept_sec_context(3sec)**, **gssdce_create_empty_oid_set(3sec)**,

gssdce_login_context_to_credential(3sec),
gssdce_register_acceptor_identity(3sec), **gss_init_sec_context(3sec)**.

gss_compare_name

Purpose

Allows an application to compare two internal names to determine whether they are equivalent

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_compare_name(
     OM_uint32 *minor_status
     gss_name_t name1
     gss_name_t name2
     int *name_equal);
```

Parameters

Input

name1
Specifies the first internal name.

name2
Specifies the second internal name.

Output

name_equal
Returns one of the following values:

TRUE The names are the same.

FALSE
The names are not the same.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_compare_name()** routine lets an application compare two internal names to determine whether they are the same. This routine does not resolve the names to see if they refer to the same object. It simply compares the input names for equivalence.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_NAMETYPE
The name passed by the *name1* or *name2* parameter is unsupported.

GSS_S_BAD_NAME
An invalid name was passed by the *name1* or *name2* parameter.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_display_name(3sec)**, **gss_import_name(3sec)**, **gss_release_name(3sec)**.

gss_context_time

Purpose

Checks the number of seconds for which the context will remain valid

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_context_time(
     OM_uint32 *minor_status
     gss_ctx_id_t context_handle
     OM_int32 *time_rec);
```

Parameters

Input

context_handle
Specifies the context to be checked.

Output

time_rec
Returns the number of seconds that the context will remain valid. Returns a 0 (zero) if the context has already expired.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_context_time()** routine checks the number of seconds for which the context will remain valid.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CONTEXT_EXPIRED
The context has already expired.

GSS_S_CREDENTIALS_EXPIRED
The context is recognized but the associated credentials have expired.

GSS_S_NO_CONTEXT
The context identified in the *context_handle* parameter was not valid.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

gss_delete_sec_context

Purpose

Deletes a security context

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_delete_sec_context(
     OM_uint32 *minor_status
     gss_ctx_id_t *context_handle
     gss_buffer_t output_token_buffer);
```

Parameters

Input/Output

context_handle
Specifies the context handle for the context to delete.

Output

minor_status
Returns a status code from the security mechanism.

output_token_buffer
Returns a token to pass to the context acceptor.

Description

The **gss_delete_sec_context()** routine deletes a security context. It also deletes the local data structures associated with the security context. When it deletes the context, the routine can generate a token. The application passes the token to the context acceptor. The context acceptor then passes the token to the **gss_process_context_token()** routine, telling it to delete the context and all associated local data structures.

When the context is deleted, the applications cannot use the *context_handle* parameter for additional security services.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

GSS_S_NO_CONTEXT
The supplied context handle did not refer to a valid context.

Related Information

Functions: **gss_accept_sec_context(3sec)**, **gss_init_sec_context(3sec)**, **gss_process_context_token(3sec)**.

gss_display_name

Purpose

Provides to an application the textual representation of an opaque internal name

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_display_name(
     OM_uint32 *minor_status
     gss_name_t input_name
     gss_buffer_t output_name_buffer
     gss_OID *output_name_type);
```

Parameters

Input

input_name
Specifies the name to convert to text.

Output

output_name_buffer
Returns the name as a character string.

output_name_type
Returns the type of name to display as a pointer to static storage. The application should treat this as read-only.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_display_name()** routine provides an application with the text form of an opaque internal name. The application can use the text to display the name but not to print it.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_NAMETYPE
The name passed by the *input_name* parameter is recognized.

GSS_S_BAD_NAME
An invalid name was passed by the *input_name* parameter.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_compare_name(3sec)**, **gss_import_name(3sec)**, **gss_release_name(3sec)**.

gss_display_status

Purpose

Provides an application with the textual representation of a GSSAPI status code that can be displayed to a user or used for logging

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_display_status(
     OM_uint32 *minor_status
     int status_value
     int status_type
     gss_OID mech_type
     int *message_context
     gss_buffer_t status_string);
```

Parameters

Input

status_value
Specifies the status value to convert.

status_type
Specifies one of the following status types:

GSS_C_GSS_CODE
Major status; a GSS status code.

GSS_C_MECH_CODE
Minor status; either a DCE security status code or a Kerberos status code.

mech_type
Specifies the security mechanism. To use DCE security, specify either of the following:

- **GSSDCE_C_OID_DCE_KRBV5_DES**
- **GSS_C_NULL_OID_SET**

To use Kerberos Version 5, specify **GSSDCE_C_OID_KRBV5_DES**.

Input/Output

message_context
Indicates whether the status code has multiple messages to read.

The first time an application calls the routine, you initialize the parameter to 0 (zero). The routine returns the first message. If there are more messages, the routine sets the parameter to a nonzero value. The application calls the routine repeatedly to get the next message, until the *message_context* parameter is zero again.

Output

status_string
Returns the status value as a text message.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_display_status()** routine provides the context initiator with a textual representation of a Generic Security Service Application Programming Interface (GSSAPI) status code so that the application can display the message to a user or log the message. Because some status values can indicate more than one error, the routine enables the calling application to process status codes with multiple messages.

The *message_context* parameter indicates which error message the application should extract from the *status_value* parameter. The first time an application calls the routine, it should initialize the *message_context* parameter to 0 (zero) and return the first message. If there are additional messages to read, the **gss_display_status()** routine returns a nonzero value. The application can call **gss_display_status()** repeatedly to generate a single text string for each call.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_MECH
The translation requires a mechanism that is unsupported or unavailable.

GSS_S_BAD_STATUS
Either the status value was not recognized or the status type was something other than **GSS_C_GSS_CODE** or **GSS_C_MECH_CODE**.

GSS_S_FAILURE
The routine failed. Check the *minor_status* for details.

Related Information

Functions: **gss_accept_sec_context(3sec)**, **gss_acquire_cred(3sec)**, **gss_compare_name(3sec)**, **gss_delete_sec_context(3sec)**, **gss_display_status(3sec)**, **gss_import_name(3sec)**, **gss_inquire_cred(3sec)**, **gssdce_extract_creds_from_sec_context(3sec)**, **gssdce_login_context_to_cred(3sec)**.

gss_import_name

Purpose

Converts a printable name to an internal form

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_import_name(
     OM_uint32 *minor_status
     gss_buffer_t input_buffer_name
     gss_OID input_name_type
     gss_name_t *output_name);
```

Parameters

Input

input_name_buffer
Specifies the buffer containing the printable name to convert.

input_name_type
Specifies the object identifier for the type of printable name.

Specify **GSS_C_NULL_OID** to use the DCE name. You can explicitly request the DCE name by using **GSSDCE_C_OID_DCE_NAME**. To help ensure portability of your application, use the default, **GSS_C_NULL_OID**.

Output

output_name
Returns the name in an internal form.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_import_name()** routine converts a printable name to an internal form.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_NAMETYPE
The name passed by the *input_name* parameter is not recognized.

GSS_S_BAD_NAME
The routine could not interpret the *input_name* parameter as a name of the type specified.

GSS_S_FAILURE
Check the minor status for details.

Related Information

Functions: **gss_compare_name(3sec)**, **gss_display_name(3sec)**, **gss_release_name(3sec)**.

gss_indicate_mechs

Purpose

Allows an application to determine which underlying security mechanisms are available

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_indicate_mechs(
     OM_uint32 *minor_status
     gss_OID_set *mech_set);
```

Parameters

Output

mech_set
: Returns the set of supported security mechanisms. The value of **gss_OID_set** is a pointer to a static storage and should be treated as read-only by the context initiator.

minor_status
: Returns a status code from the security mechanism.

Description

The **gss_indicate_mechs()** routine enables an application to determine which underlying security mechanisms are available. These are DCE security and Kerberos Version 5.

You can use the **gssdce_test_oid_set_member()** routine to check whether a specific security mechanism is available.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
: The routine was completed successfully.

GSS_S_FAILURE
: The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gssdce_test_oid_set_member(3sec)**.

gss_init_sec_context

Purpose

Establishes a security context between the context initiator and a context acceptor

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_init_sec_context(
     OM_uint32 *minor_status
     gss_cred_id_t claimant_cred_handle
     gss_ctx_id_t *context_handle
     gss_name_t target_name
     gss_OID mech_type
     int req_flags
     int time_req
     gss_channel_bindings_t input_channel_bindings
     gss_buffer_t input_token
     gss_OID *actual_mech_types
     gss_buffer_t output_token
     int *ret_flags
     OM_int32 *time_rec);
```

Parameters

Input

claimant_cred_handle
Specifies an optional handle for the credential. To use the default credential, supply **GSS_C_NO_CREDENTIAL**. The credential handle created refers to the DCE default login context. The credential must be either an **INITIATE** or **BOTH** type credential.

target_name
Specifies the name of the context acceptor.

mech_type
Specifies the security mechanism. To use DCE security, specify either of the following:

- **GSS_C_OID_DCE_KRBV5_DES**
- **GSS_C_NULL_OID**

To use Kerberos, specify **GSS_C_OID_KRBV5_DES**.

req_flags
Specifies four independent flags, each of which requests that the context support a service option. The following symbolic names are provided to correspond to each flag. The symbolic names should be logically ORed to form a bit-mask value.

GSS_C_DELEG_FLAG

TRUE Credentials were delegated to the context acceptor.

FALSE
No credentials were delegated.

GSS_C_MUTUAL_FLAG

TRUE The context acceptor has been asked to authenticate itself.

FALSE
The context initiator has not been asked to authenticates itself.

GSS_C_REPLAY_FLAG

TRUE Replayed signed or sealed messages will be detected.

FALSE
Replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG

TRUE Out-of-sequence signed or sealed messages will be detected.

FALSE
Out-of-sequence signed or sealed messages will not be detected.

time_req
Specifies the desired number of seconds for which the context should remain valid. To specify the default validity period, use 0 (zero).

input_chan_bindings
Specifies the bindings set by the context initiator. Allows the context initiator to bind the channel identification information securely to the security context.

input_token
Specifies the token received from the context acceptor.

The first time the application calls the routine, you specify **GSS_NO_BUFFER**. Subsequent calls require a token from the context acceptor.

Input/Output

context_handle
Specifies the context handle for the new context.

The first time the application calls the routine, you specify **GSS_C_NO_CONTEXT**. Subsequent calls use the value returned by the first call.

Output

actual_mech_type
Returns one of the following values indicating the security mechanism:

- **GSS_C_OID_DCE_KRBV5_DES** for DCE security
- **GSS_C_OID_KRBV5_DES** for Kerberos

output_token
Returns the token to send to the context acceptor.

If the length field of the returned buffer is 0 (zero), no token is sent.

ret_flags
Returns six independent flags, each of which indicates that the context supports a service option. The following symbolic names are provided to correspond to each flag:

GSS_C_DELEG_FLAG

TRUE Credentials were delegated to the context acceptor.

FALSE
No credentials were delegated.

GSS_C_MUTUAL_FLAG

TRUE The context acceptor has been asked to authenticate itself.

FALSE
The context acceptor has not been asked to authenticate itself.

GSS_C_REPLAY_FLAG

TRUE Replayed signed or sealed messages will be detected.

FALSE
Replayed messages will not be detected.

GSS_C_SEQUENCE_FLAG

TRUE Out-of-sequence signed or sealed messages will be detected.

FALSE
Out-of-sequence signed or sealed messages will not be detected.

GSS_C_CONF_FLAG

TRUE Confidentiality service can be invoked by calling the **gss_seal()** routine.

FALSE
No confidentiality service is available. (Confidentiality can be provided using the **gss_seal()** routine, which provides only message encapsulation, data-origin authentication, and integrity services.)

GSS_C_INTEG_FLAG

TRUE Integrity service can be invoked by calling either the **gss_sign()** or **gss_seal()** routine.

FALSE
Integrity service for individual messages is unavailable.

time_rec
Returns the number of seconds for which the context will be valid. If the mechanism does not support credential expiration, the routine returns the value **GSS_C_INDEFINITE**. If the credential expiration time is not required, specify NULL.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_init_sec_context()** routine is the first step in the establishment of a security context between the context initiator and the context acceptor. To ensure the portability of the application, use its default credential by supplying **GSS_C_NO_CREDENTIAL** to the *claimant_cred_handle* parameter. Specify an explicit credential when the application needs an additional credential; for example, to use delegation.

The first time the application calls the **gss_init_sec_context()** routine, specify the *input_token* parameter as **GSS_NO_BUFFER**. Calls to the routine can return an *output_token* for transfer to the context acceptor. The context acceptor presents the token to the **gss_accept_sec_context()** routine.

If the context initiator does not require a token, **gss_init_sec_context()** sets the length field of the *output_token* argument to 0 (zero).

To complete establishing the context, the calling application can require one or more reply tokens from the context acceptor. If the application requires reply tokens, the **gss_init_sec_context()** routine returns a status value of **GSS_S_CONTINUE_NEEDED**. The application calls the routine again when the reply token is received from the context acceptor and passes the token to the **gss_init_sec_context()** routine via the *input_token* parameter.

The values returned by the *ret_flags* and *time_rec* parameters are not defined unless the routine returns the status **GSS_S_COMPLETE**.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_BINDINGS
The *input_token* parameter contains different channel bindings from those specified with the *input_chan_bindings* parameter.

GSS_S_BAD_NAMETYPE
The *target_name* parameter contains an invalid or unsupported name type.

GSS_S_BAD_NAME
The *target_name* parameter was incorrectly formed.

GSS_S_BAD_SIG
Indicates either that the *input_token* parameter contains an invalid signature or that the *input_token* parameter contains a signature that could not be verified.

GSS_S_CONTINUE_NEEDED
To complete the context, the **gss_init_sec_context()** routine must be called again with a token required from the context acceptor.

GSS_S_CREDENTIALS_EXPIRED
The referenced credentials have expired.

GSS_S_DEFECTIVE_CREDENTIAL
Consistency checks performed on the credential failed.

GSS_S_DEFECTIVE_TOKEN
Consistency checks performed on the *input_token* parameter failed.

GSS_S_DUPLICATE_TOKEN
The *input_token* parameter was already processed. This is a fatal error that occurs during context establishment.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

GSS_S_NO_CONTEXT
The supplied context handle did not refer to a valid context.

GSS_S_OLD_TOKEN
The *input_token* parameter was too old. This is a fatal error that occurs during context establishment.

Related Information

Functions: **gss_accept_sec_context(3sec)**, **gss_delete_sec_context(3sec)**.

gss_inquire_cred

Purpose

Provides the calling application information about a credential

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_inquire_cred(
     OM_uint32 *minor_status
     gss_cred_id_t cred_handle
     gss_name_t *name
     OM_uint32 *lifetime
     int *cred_usage
     gss_OID_set *mechs);
```

Parameters

Input

cred_handle
Specifies a handle for the target credential. To get information about the default credential, specify **GSS_C_NO_CREDENTIAL** .

Output

name Returns the principal name asserted by the credential. If the principal name is not required, specify NULL.

lifetime
Returns the number of seconds for which the credential will remain valid.

If the credential expired, the parameter returns a 0 (zero). If there is no credential expiration, the parameter returns the value **GSS_C_INDEFINITE**. If an expiration time is not required, specify NULL.

cred_usage
Returns one of the following values describing how the application can use the credential:

- **GSS_C_INITIATE**
- **GSS_C_ACCEPT**
- **GSS_C_BOTH**

If no usage information is required, specify NULL.

mechs Returns a set of security mechanisms supported by the credential, as follows:

- **GSSDCE_C_OID_DCE_KRBV5_DES** (for DCE security)
- **GSSDCE_C_OID_KRBV5_DES** (for Kerberos)

minor_status
Returns a status code from the security mechanism.

Description

The **gss_inquire_cred()** routine provides information about a credential to the calling application. The calling application must first have called the **gss_acquire_cred()** routine for a handle for the credential.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CREDENTIALS_EXPIRED
The credentials expired. If the *lifetime* parameter was passed as NULL, it is set to 0 (zero).

GSS_S_DEFECTIVE_CREDENTIAL
The credentials were invalid.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CRED
The routine could not access the credentials.

Related Information

Functions: **gss_acquire_cred(3sec)**.

gss_process_context_token

Purpose

Passes a context to the security service

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_process_context_token(
     OM_uint32 *minor_status
     gss_ctx_id_t *context_handle
     gss_buffer_t input_token_buffer);
```

Parameters

Input

context_handle
Specifies the context handle on which the security service processes the token.

input_token_buffer
Specifies an opaque pointer to the first byte of the token to be processed.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gss_process_context_token()** routine passes tokens generated by the **gss_delete_security_context()** routine to the security service.

Usually, tokens are associated with either the context establishment or with per-message security services. If the tokens are associated with the context establishment, they are passed to the **gss_init_sec_context()** or **gss_accept_sec_context()** routine. If the tokens are associated with the per-message security service, they are passed to the **gss_verify()** or **gss_unseal()** routine. Tokens generated by the **gss_delete_security_context()** routine are passed by the **gss_process_context_token()** routine to the security service for processing.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_DEFECTIVE_TOKEN
Consistency checks performed on the *input_token* parameter failed.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter return value for more information.

GSS_S_NO_CONTEXT
The supplied context handle did not refer to a valid context.

Related Information

Functions: **gss_delete_security_context(3sec)**.

gss_release_buffer

Purpose

Frees storage associated with a buffer

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_release_buffer(
     OM_uint32 *minor_status
     gss_buffer_t buffer);
```

Parameters

Input

buffer The buffer to delete.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gss_release_buffer()** routine deletes the buffer by freeing the storage associated with it.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. See the *minor_status* parameter for details.

gss_release_cred

Purpose

Marks a credential for deletion

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_release_cred(
     OM_uint32 *minor_status
     gss_cred_id_t *cred_handle);
```

Parameters

Input

cred_handle
Specifies the buffer containing the opaque credential handle. This information is optional. To release the default credential, specify **GSS_C_NO_CREDENTIAL**.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gss_release_cred()** routine informs the GSSAPI that a credential is no longer required and marks it for deletion.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CRED
The credentials could not be accessed.

gss_release_name

Purpose

Frees storage associated with an internal name that was allocated by a GSSAPI routine.

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_release_name(
     OM_uint32 *minor_status
     gss_name_t *name);
```

Parameters

Input

name The name to delete.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gss_release_name()** routine deletes the internal name by freeing the storage associated with that internal name.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_NAME
The *name* parameter did not contain a valid name.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_compare_name(3sec)**, **gss_display_name(3sec)**, **gss_import_name(3sec)**.

gss_release_oid_set

Purpose

Frees storage associated with a gss_OID_set object

Synopsis

```
#include <dce/gssapi.h>

OM_uint32  gss_release_oid_set  (
     OM_uint32 *minor_status
     gss_OID_set set);
```

Parameters

Input

set The OID set to delete.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gss_release_oid_set()** routine frees storage that is associated with the **gss_OID_set** parameter and was allocated by a GSSAPI routine.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

gss_seal

Purpose

Cryptographically signs and optionally encrypts a message

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_seal(
     OM_uint32 *minor_status,
     gss_ctx_id_t context_handle,
     int conf_req_flag,
     int qop_req,
     gss_buffer_t input_message_buffer,
     int *conf_state,
     gss_buffer_t output_message_buffer);
```

Parameters

Input

context_handle
Specifies the context on which the message is sent.

conf_req_flag
Specifies the requested level of confidentiality and integrity services, as follows:

TRUE Both confidentiality and integrity services are requested.

FALSE
Only integrity services are requested.

qop_req
Specifies the cryptographic algorithm, or quality of protection. Specify **GSS_C_QOP_DEFAULT**. The DCE GSSAPI supports only one quality of protection.

input_message_buffer
Specifies the message to seal.

Output

conf_state
Returns the requested level of confidentiality and integrity services, as follows:

TRUE Confidentiality, data origin, authentification, and integrity services have been applied.

FALSE
Only integrity and data origin services have been applied.

output_message_buffer
Returns the buffer to receive the sealed message.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_seal()** routine cryptographically signs and optionally encrypts a message. The *output_message* parameter contains both the signature and the message.

Although the *qop_req* parameter enables a choice between several qualities of protection, DCE GSSAPI supports only one quality of protection. If you specify an unsupported protection, the **gss_seal()** routine returns a status of **GSS_S_FAILURE**.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CONTEXT_EXPIRED
The context has already expired.

GSS_S_CREDENTIALS_EXPIRED
The context is recognized but the associated credentials have expired.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CONTEXT
The context identified in the *context_handle* parameter was not valid.

gss_sign

Purpose

Generates a cryptographic signature for a message.

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_sign(
     OM_uint32 *minor_status,
     gss_ctx_id_t context_handle,
     int qop_req,
     gss_buffer_t message_buffer,
     gss_buffer_t msg_token);
```

Parameters

Input

context_handle
Specifies the context on which the message is sent.

qop_req
Specifies the cryptographic algorithm, or quality of protection. Specify **GSS_C_QOP_DEFAULT**. DCE GSSAPI supports only one quality of protection.

message_buffer
Specifies the message to send.

Output

msg_token
Returns the buffer to receive the signature token to transfer to the context acceptor.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_sign()** routine generates an encrypted signature for a message. It places the signature in a token for transfer to the context acceptor.

Although the *qop_req* parameter enables a choice between several qualities of protection, DCE GSSAPI supports only one quality of protection. If you specify an unsupported protection, the **gss_sign()** routine returns a status of **GSS_S_FAILURE**.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CONTEXT_EXPIRED
The context has already expired.

GSS_S_CREDENTIALS_EXPIRED
The context is recognized but the associated credentials have expired.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CONTEXT
The context identified in the *context_handle* parameter was not valid.

gss_unseal

Purpose

Converts a sealed message into a usable form and verifies the embedded signature

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_unseal(
     OM_uint32 *minor_status,
     gss_ctx_id_t context_handle,
     gss_buffer_t input_message_buffer,
     gss_buffer_t output_message_buffer,
     int *conf_state,
     int *qop_state);
```

Parameters

Input

context_handle
Specifies the context on which the message arrived.

input_message_buffer
Specifies the sealed message.

output_message_buffer
Specifies the buffer to receive the unsealed message.

Output

conf_state
Returns the requested level of confidentiality and integrity services, as follows:

TRUE Both confidentiality and integrity services are requested.

FALSE
Only integrity services are requested.

qop_state
Returns the cryptographic algorithm, or quality of protection.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_unseal()** routine converts a sealed message to a usable form and verifies the embedded signature. The *conf_state* parameter indicates whether the message was encrypted. The *qop_state* parameter indicates the quality of protection.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_BAD_SIG
The signature was incorrect.

GSS_S_CONTEXT_EXPIRED
The context has already expired.

GSS_S_CREDENTIALS_EXPIRED
The context is recognized but the associated credentials have expired.

GSS_S_DEFECTIVE_TOKEN
The token failed consistency checks.

GSS_S_DUPLICATE_TOKEN
The token was valid and contained the correct signature but it had already been processed.

GSS_S_FAILURE
The routine failed. The context specified in the *context_handle* parameter was not valid.

GSS_S_NO_CONTEXT
The context identified in the *context_handle* parameter was not valid.

GSS_S_OLD_TOKEN
The token was valid and contained the correct signature but it is too old.

GSS_S_UNSEQ_TOKEN
The token was valid and contained the correct signature but it has been verified out of sequence. An earlier token signed or sealed by the remote application has not been processed locally.

Related Information

Functions: **gss_seal(3sec)**, **gss_sign(3sec)**.

gss_verify

Purpose

Checks that the cryptographic signature fits the supplied message

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_verify(
     OM_uint32 *minor_status,
     gss_ctx_id_t context_handle,
     gss_buffer_t message_buffer,
     gss_buffer_t token_buffer,
     int *qop_state);
```

Parameters

Input

context_handle
Specifies the context on which the message arrived.

message_buffer
Specifies the message to be verified.

token_buffer
Specifies the signature token to be associated with the message.

Output

qop_state
Returns the cryptographic algorithm, or quality of protection, from the signature.

minor_status
Returns a status code from the security mechanism.

Description

The **gss_verify()** routine checks that an encrypted signature, in the *token_buffer* parameter, fits the message in the *message_buffer* buffer. The application receiving the message can use the *qop_state* parameter to check the message's protection.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CONTEXT_EXPIRED
The context has already expired.

GSS_S_CREDENTIALS_EXPIRED
The context is recognized but the associated credentials have expired.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CONTEXT
The context identified in the *context_handle* parameter was not valid.

Related Information

Functions: **gss_seal(3sec)**, **gss_sign(3sec)**.

gssdce_add_oid_set_member

Purpose

Adds an OID to an OID set

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_add_oid_set_member(
     OM_uint32* minor_status
     gss_OID* member_OID
     gss_OID_set* OID_set);
```

Parameters

Input

member_OID
Specifies the OID you want to add to the OID set.

OID_set
Specifies an OID set.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_add_oid_set_member()** routine adds a new OID to an OID set. If an OID set does not exist, you can create a new, empty OID set with the **gssdce_create_empty_oid_set()** routine.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gssdce_create_empty_oid_set(3sec)**.

gssdce_create_empty_oid_set

Purpose

Creates a new, empty OID set to which members can be added by calling **gssdce_add_oid_set_member()**

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_create_empty_oid_set(
     OM_uint32 *minor_status
     gss_OID_set *OID_set);
```

Parameters

Input

OID_set
Specifies the OID set you want to create.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_create_empty_oid_set()** routine creates a new, empty OID set to which the context initiator can add members. Use the **gssdce_add_oid_set_member()** routine to add members to the OID set.

Use the **gssdce_create_empty_oid_set()** routine to specify a set of security mechanisms with which you can use an acquired credential. To create a credential that can accept a security context using DCE security, Kerberos, or a combination of the two, use the **gss_acquire_cred()** routine.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gssdce_add_oid_set_member(3sec)**.

gssdce_cred_to_login_context

Purpose

Obtains the DCE login context associated with a GSSAPI credential

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_cred_to_login_context(
     OM_uint32 *minor_status
     cred_id_t *cred_handle
     sec_login_handle_t login_context);
```

Parameters

Input

cred_handle
Specifies the credential handle.

Output

login_context
Returns the DCE login context associated with the credential.

minor_status
Returns a status code from the security mechanism.

Description

Using the **gssdce_cred_to_login_context()** routine, an application can obtain the DCE login context associated with a GSSAPI credential. Only credentials with usage-types **INIT** or **BOTH** have associated login contexts.

Use this routine in the following situations:

- If you want to add delegation notes to a login context
- To use an **INITIATE** or **BOTH** credential to initiate an authenticated RPC call

The application must delete the login context when it no longer needs the credentials or the login context.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_CREDENTIALS_EXPIRED
The credentials have expired.

GSS_S_DEFECTIVE_CREDENTIAL
The credential is defective in some way.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CRED
The routine requested the default login context, but no default login context was available.

Related Information

Functions: **gssdce_login_context_to_cred(3sec)**, **sec_login_purge_contexts(3sec)**, **sec_login_release_context(3sec)**.

gssdce_extract_creds_from_sec_context

Purpose

Extracts a DCE credential from a GSSAPI security context

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_extract_creds_from_sec_context(
     OM_uint32 *minor_status
     gss_ctx_id_t context_handle
     rpc_authz_cred_handle_t output_cred);
```

Parameters

Input

context_handle
Specifies the handle of the security context containing the DCE credential.

Output

output_cred
Returns the DCE credential.

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_extract_creds_from_sec_context()** routine extracts the context initiator's DCE credential from a context acceptor's security context. Use this routine if the underlying mechanism type is DCE security (**GSSDCE_C_OID_DCE_KRBV5_DES**).

The context acceptor calls the **gssdce_extract_creds_from_sec_context()** routine to get the DCE credential containing the privilege attributes of the context initiator. DCE credentials are used by DCE access control list (ACL) managers to determine whether the initiator has the right to access the object to which an ACL refers.

The principal contained in the DCE credential may not be the same as the *src_name* parameter value from the **gss_accept_sec_context()** routine. The principal in the DCE credential may be a compound principal.

If the context was established by calling the **gss_init_set_context()** routine and specifying **GSSDCE_C_OID_KRBV5_DES** to use Kerberos (instead of DCE security), the **gssdce_extract_creds_from_sec_context()** routine returns a major status of O and a minor status of O.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

GSS_S_NO_CONTEXT
The routine could not access the security context.

Related Information

Functions: **gss_init_sec_context(3sec)**.

gssdce_login_context_to_cred

Purpose

Creates a GSSAPI credential handle for a context initiator or context acceptor from a DCE login context

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_login_context_to_cred(
     OM_uint32 *minor_status
     sec_login_handle_t login_context
     OM_uint32 lifetime_req
     OID_set desired_mechs
     cred_id_t *output_cred_handle
     OID_set *actual_mechs
     OM_uint32 lifetime_rec);
```

Parameters

Input

login_context
Specifies the DCE login context handle. To use the default login context handle, specify NULL.

lifetime_req
Specifies the number of seconds that the credential should remain valid.

desired_mechs
Specifies the object identifier (OID) set for the security mechanism to use with the credential, as follows:

DCE security
Specify **GSS_C_NULL_OID_SET**.

Kerberos
Specify **GSSDCE_C_OID_KRBV5_DES**.

Both DCE security and Kerberos
Specify **GSSDCE_C_OID_DCE_KRBV5_DES** and **GSSDCE_C_OID_KRBV5_DES**.

To help ensure portability of your application, use the default security mechanism by specifying **GSS_C_NULL_OID_SET**.

Output

output_cred_handle
Returns the credential handle.

actual_mechs
Returns the set specifying the security mechanisms with which the credential can be used. The set can contain one or both of the following:

- **GSSDCE_C_OID_DCE_KRBV5_DES** (for DCE security)
- **GSSDCE_C_OID_KRBV5_DES** (for Kerberos)

lifetime_rec
Returns the number of seconds that the credential will remain valid.

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_login_context_to_cred()** routine creates a generic security service application programming interface (GSSAPI) credential handle for the context initiator or context acceptor from a DCE login context. The routine creates a credential that can be used to initiate or acquire a security context. Use this routine if you need to create a GSSAPI credential for delegation.

Status Codes

The following describes a partial list of codes (messages) that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all messages. The following status codes can be returned:

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_DEFECTIVE_CREDENTIAL
The credential is defective in some way.

GSS_S_NO_CRED
The routine requested the default login context, but no default login context was available.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gssdce_cred_to_login_context(3sec)**.

gssdce_register_acceptor_identity

Purpose

Registers a context acceptor's identity

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gss_register_acceptor_indentity(
     OM_uint32 *minor_status
     unsigned_char_t *acceptor_principal_name
     rpc_auth_key_retrieval_fn_t get_key_fn
     void *arg);
```

Parameters

Input

acceptor_principal_name
Specifies the principal name to use for the context acceptor.

get_key_fn
Specifies either the DCE default key-retrieval routine or the address of a routine that returns encryption keys.

arg Specifies an argument to pass to the *get_key_fn* key acquisition routine. To specify the DCE default, use NULL.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_register_acceptor_identity()** routine registers the server principal name as an identity claimed by the context acceptor and informs DCE security where to find the key table containing the principal's key information.

The **gssdce_register_acceptor_identity()** routine uses the *get_key_fn* and *arg* parameters of the **rpc_server_register_auth_info()** routine to find the key for the token for the context acceptor's principal name. The following table lists the values for the parameters and which key tables they point to:

Retrieval Routine	Key Table	Explanation
NULL	NULL	Uses the default DCE retrieval routine to get the key from the DCE key table. This is accomplished via the default key table, **/krb/v5srvtab**.

NULL	*string= key_table_name*	Uses the default DCE retrieval routine to get the key from the a key table whose name you specify using the argument string.
routine_address	*user_written_routine*	Uses a user-written retrieval routine to get the key from a key table specified in the routine.

For more information on registering a server with DCE, refer to the **rpc_server_register_auth_info(3rpc)** reference page.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the minor status for details.

Related Information

Functions: **gss_accept_sec_context(3sec)**, **rpc_server_register_auth_info(3rpc)**.

gssdce_set_cred_context_ownership

Purpose

Changes the ownership of a DCE credential's login context

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_set_cred_context_ownership(
     OM_uint32 *minor_status
     gss_cred_id_t credential_handle
     int ownership);
```

Parameters

Input

credential_handle
Specifies the handle of the DCE credential to be modified.

ownership
Specifies the owner of the DCE credential. Specify one of the following:

GSSDCE_C_OWNWERSHIP_GSSAPI
Specifies that the credential's login context is owned by the generic security service application programming interface (GSSAPI).

GSSDCE_C_OWNERSHIP_APPLICATION
Specifies that the credential's login context is owned by the application.

Output

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_set_cred_context_ownership()** routine modifies the ownership of a DCE credential's login context. **INIT** type and **BOTH** type credentials have DCE login contexts. Normally, these internal login contexts are deleted when the credential is released (when the application calls the **gss_release_cred()** routine). However, for credentials created by the **gssdce_cred_to_login_context()** and credentials passsed to the **gsscdce_cred_to_login_context()**routine, the application may have an external reference to the credential's login context and may still be using the login context. The GSSAPI will not delete internal login contexts of these credentials when they are released.

This routine allows the application to modify the ownership of a credential's login context. If ownership is changed to **GSSDCE_C_OWNERSHIP_GSSAPI**, the login context is deleted when GSSAPI releases the credential. If ownership is changed to **GSSDCE_C_OWNERSHIP_APPLICATION**, the application is responsible for deleting the login context. DCE credential login contexts that are owned by an application must not be deleted until the credential is released since the GSSAPI may still need to access the credential's login context.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gss_release_buffer(3sec)**, **gssdce_cred_to_login_context(3sec)**.

gssdce_test_oid_set_member

Purpose

Checks an OID set to see if a specified OID is in the set

Synopsis

```
#include <dce/gssapi.h>

OM_uint32   gssdce_test_oid_set_member(
     OM_uint32 *minor_status
     gss_OID member_OID
     gss_OID_set set
     int* is_present);
```

Parameters

Input

member_OID
Specifies the OID to search for in the OID set.

set Specifies the OID set to check.

Output

is_present
Returns one of the following values to indicate whether the OID is a member of the OID set:

Returns...	If...
1	The OID is present as a member of the OID set
0	The OID is absent, not a member of the OID set

minor_status
Returns a status code from the security mechanism.

Description

The **gssdce_test_oid_set_member()** routine checks an OID set to see if the specified OID is a member of the set. To add a member to an OID set, use the **gssdce_add_oid_set_member()** routine.

The **gssdce_test_oid_set_member()** routine uses the value of the *actual_mechs* output parameter from the **gss_acquire_cred()** routine to get the list of OIDs. It checks this list to see if any of the OIDs are members of the OID set.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

GSS_S_COMPLETE
The routine was completed successfully.

GSS_S_FAILURE
The routine failed. Check the *minor_status* parameter for details.

Related Information

Functions: **gss_acquire_cred(3sec)**, **gss_indicate_mechs(3sec)**, **gssdce_add_oid_set_member(3sec)**.

pkc_add_trusted_key

Purpose

Adds a key to specified trust list

Synopsis

```
#include <pkc_certs.h>

 pkc_add_trusted_key(
     pkc_trust_list_t * trust_list
     const pkc_trusted_key_t & key);
```

Parameters

Input

trust_list
Specifies trust list to which key should be added.

key Specifies key to add.

Description

pkc_add_trusted_key(3sec) adds a specified key to a specified trust list.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**,

pkc_pending_revocation.class(3sec), **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_append_to_trustlist

Purpose

Appends one or more items to trust list

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_append_to_trustlist(
     trustlist_t ** tr_list
     trustitem_t * tr_item
     size_t no_of_tr_items);
```

Parameters

Input

tr_list Specifies trust list to which item(s) are to be appended.

tr_item
Specifies item(s) to append to trust list.

no_of_tr_items
Specifies number of items to append.

Description

pkc_append_to_trustlist(3sec) appends one or more items to a trust list (a **pkc_trust_list_t**, pointed to by **(*tr_list)->handle**).

If the trust list is invalid, **pkc_s_bad_param** is returned.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_ca_key_usage.class

Purpose

A class that expresses key usage

Member Functions

Public

- **pkc_ca_key_usage_t & operator = (unsigned long c)**
- **pkc_ca_key_usage_t(unsigned long c = 0xffffffffu)**

Description

pkc_ca_key_usage_t expresses the usage of a key.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Parent Class

This class is derived from **pkc_generic_key_usage_t**.

Related Information

Classes: **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_check_cert_against_trustlist

Purpose

Checks specified certificate against specified list of trusted keys

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_check_cert_against_trustlist(
     pkc_trust_list_t * trust_list
     const Certificate * cert
     int revoked_certs_permitted);
```

Parameters

Input

trust_list
Specifies list of trusted keys to check certificate against.

cert Specifies certificate to check.

revoked_certs_permitted
Specifies whether revoked certificates should still be trusted for dates prior to their revocation date.

Description

pkc_check_cert_against_trustlist(3sec) checks the specified certificate against the specified list of trusted keys. If the certificate is valid and can be verified from the trust list, its content is added to the trust list. *revoked_certs_permitted* is a flag that specifies whether revoked certificates should still be trusted for dates prior to their revocation date.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**,

pkc_revoke_certificates(3sec). Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_constraints.class

Purpose

A class that expresses constraints on names

Member Data

Public

- **unsigned path_length**

  The maximum path length that can be certified by the key (if the entity can act as a certifying authority). **0xffffu** means ''unlimited''.
- **pkc_name_subord_constraints_t subord_constraints**
- **pkc_name_subtree_constraints_t subtree_constraints**

Member Functions

Public

- **pkc_constraints_t & operator = (const pkc_constraints_t & o)**
- **pkc_constraints_t(void)**
- **unsigned32 constrain()**

  Adds the specified constraints. Takes the following argument:
  - **const pkc_constraints_t & o**
- **char is_permitted() const**

  Takes the following arguments:
  - **const x500name & ca_name**
  - **const x500name & subject_name**
- **void get_next_link_constraint() const**

  Generates a new name constraint that will be applicable to a certificate issued by the subject of this constraint. Takes the following argument:
  - **pkc_constraints_t ** new_constraints**

Description

pkc_constraints_t is a class that expresses constraints on the names that can be certified by a given key. Three types of constraint can be checked: total path length, name subordination, and subtree constraints.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**,

pkc_constraints.class(3sec)

pkc_revocation_list.class(3sec), **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_copy_trustlist

Purpose

Copies a trustlist

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_copy_trustlist(
     const pkc_trust_list_t * input_trust_list
     pkc_trust_list_t  * output_trust_list);
```

Parameters

Input

input_trust_list
The trust list to be copied.

Output

output_trust_list
The copied trust list.

Description

pkc_copy_trustlist(3sec) creates a functionally equivalent copy of a trust list. The key ids within the newly created trust list will be different from those in the original trust list, but the keys and the trust relationships between them will be the same.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_delete_trustlist(3sec)**, **pkc_display_trustlist(3sec)**, **pkc_lookup_element_in_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**,

pkc_key_policies.class(3sec), **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_crypto_generate_keypair

Purpose

Generates a pair of public and private keys

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_crypto_generate_keypair(
     gss_OID algorithm
     unsigned32 size
     void * alg_info
     sec_pk_data_t * private_key
     sec_pk_data_t * public_key);
```

Parameters

Input

algorithm
Specifies the crypto module.

size Specifies the key size.

alg_info
Specifies algorithm-specific information, if any.

Output

private_key
The generated private key.

public_key
The generated public key.

Description

pkc_crypto_generate_keypair generates a pair of public and private keys. The **(*generate_keypair)()** routine of the crypto module specified by *algorithm* is called to do this (but note that crypto modules are not required to provide a **(*generate_keypair)()** function).

The *size* parameter will be used by the routine to determine the key size in some way defined by the algorithm; for the RSA algorithm, for example, it should be treated as the number of bits in the key modulus. The *private_key* and *public_key* parameters should be expected to return BER-encoded **PrivateKeyInfo** and **SubjectPublicKeyInfo** data objects respectively.

The *alg_info* parameter can be used for algorithm-specific information to modify the key generation process; NULL can be specified.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_intro(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

pkc_crypto_get_registered_algorithms

Purpose

Returns algorithm implementations

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_crypto_reg.h>

pkc_crypto_get_registered_algorithms(
     gss_OID_set *oid_set);
```

Parameters

Output

oid_set
: A pointer to an OID set describing the currently registered algorithm implementations.

Description

pkc_crypto_get_registered_algorithms(3sec) returns an OID set describing the currently registered algorithm implementations.

pkc_crypto_lookup_algorithm(3sec) may be called to obtain details about a particular algorithm.

Return Values

pkc_s_success
: Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

pkc_crypto_lookup_algorithm

Purpose

Returns cryptographic module details

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_crypto_reg.h>

unsigned32 pkc_crypto_lookup_algorithm(
     gss_OID oid
     pkc_signature_algorithm_t *details);
```

Parameters

Input

oid An OID identifying the algorithm about which details are desired.

Output

details A pointer to an algorithm implementation descriptor block for the specified algorithm.

Description

pkc_crypto_lookup_algorithm(3sec) returns a pointer to an algorithm implementation descriptor block for the specified cryptographic algorithm, and leaves the algorithm list unlocked. Calling this routine is the recommended way of obtaining information about a registered algorithm implementation.

The complete list of registered algorithms may be obtained by calling **pkc_crypto_get_registered_algorithms(3sec)**.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

pkc_crypto_register_signature_alg

Purpose

Registers a signature algorithm module

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_crypto_reg.h>

unsigned32 pkc_crypto_register_signature_alg(
     pkc_signature_algorithm_t * alg
     int replacement_policy);
```

Parameters

Input

alg A pointer to the signature algorithm module structure to be registered.

replacement_policy
Specifies how the registration is to be handled if an implementation of the algorithm is already registered. There are three possible values:

PKC_REPLACE_NONE
Specifies that an error should be returned if an implementation of the algorithm is already registered.

PKC_REPLACE_ENTRYPOINTS
Specifies that only entry points that the original implementation (if any) did not provide should be replaced. (Note that this value is not currently supported.)

PKC_REPLACE_ALL
Specifies that the new implementation should replace the existing one, if any.

Description

pkc_crypto_register_signature_alg(3sec) registers a signature algorithm module, in the form of a properly declared **pkc_signature_algorithm_t** data structure, which contains identifying information about the module as well as entry points to all of the module's functions.

Calling this routine will cause the module passed to it to be registered as a cryptographic module; it can then be accessed by other applications via the high level certification API routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_sign(3sec)**, **pkc_crypto_verify_signature(3sec)**.

pkc_crypto_sign

Purpose

Signs data with private key

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_crypto_reg.h>

pkc_crypto_sign(
     gss_OID algorithm
     sec_pk_gen_data_t data
     sec_pk_data_t private_key
     sec_pk_data_t *signature);
```

Parameters

Input

algorithm
An OID identifying the cryptographic algorithm to be used in signing the data.

data The data to be signed.

private_key
The private key (i.e., private member of a public-private key pair) to be used to sign the data.

Output

signature
The signature generated by the algorithm on the data passed.

Description

pkc_crypto_sign(3sec) searches the list of registered algorithms for an implementation of the specified algorithm. If found, the implementation is opened, if necessary, and its **(sign)()** function invoked to sign the *data*. The *signature* is returned to the caller.

Using this routine, an application can get data signed in one simple call. The alternative is to lookup the desired cryptographic module by calling **pkc_crypto_lookup_algorithm(3sec)**, then explicitly call the module's **(sign)()** routine.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_verify_signature(3sec)**.

pkc_crypto_verify_signature

Purpose

Verifies a signature

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_crypto_reg.h>

pkc_crypto_verify_signature(
     gss_OID algorithm
     sec_pk_gen_data_t data
     sec_pk_data_t public_key
     sec_pk_data_t signature);
```

Parameters

Input

algorithm
An OID identifying the cryptographic algorithm to be used in verifying the data.

data The signed data whose signature is to be verified.

public_key
The public key (i.e., public member of a public-private key pair) to be used to verify the signed data.

signature
The signature to be verified.

Description

pkc_crypto_verify_signature(3sec) searches the list of registered algorithms for an implementation of the specified algorithm. If found, the implementation is opened, if necessary, and its **(verify)()** function invoked to verify the data and signature passed by the caller.

The routine returns 0 for a correct signature, **pkc_invalid_signature** for an incorrect signature, or another DCE-defined error status to indicate any other errors.

Using this routine, an application can verify signed data in one simple call. The alternative is to lookup the desired cryptographic module by calling **pkc_crypto_lookup_algorithm(3sec)**, then explicitly call the module's **(verify)()** routine.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_crypto_generate_keypair(3sec)**, **pkc_crypto_get_registered_algorithms(3sec)**, **pkc_crypto_lookup_algorithm(3sec)**, **pkc_crypto_register_signature_alg(3sec)**, **pkc_crypto_sign(3sec)**.

pkc_delete_trustlist

Purpose

Deletes a trust list

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_delete_trustlist(
     pkc_trust_list_t * trust_list);
```

Parameters

Input

trust_list
The trust list to be deleted.

Description

pkc_delete_trustlist(3sec) deletes a trust list and all keys within it.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_copy_trustlist(3sec)**, **pkc_display_trustlist(3sec)**, **pkc_lookup_element_in_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**,

pkc_trust_list_element.class(3sec), **pkc_trusted_key.class(3sec)**.

pkc_display_trustlist

Purpose

Displays information about a trust list

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_display_trustlist(
     const pkc_trust_list_t * input_trust_list);
```

Parameters

Input

input_trust_list
The trust list to be displayed.

Description

pkc_display_trustlist(3sec) is a debugging routine, intended for use when developing a policy module. It prints information about a trust list (all the keys, the trust relationships between keys, etc.) to standard output. It can be used to verify that the trust chains that a policy module implementor expects are actually being built within the trust list.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_copy_trustlist(3sec)**, **pkc_delete_trustlist(3sec)**, **pkc_lookup_element_in_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**,

pkc_name_subtree_constraint.class(3sec),
pkc_name_subtree_constraints.class(3sec),
pkc_pending_revocation.class(3sec), **pkc_revocation.class(3sec)**,
pkc_revocation_list.class(3sec), **pkc_trust_list.class(3sec)**,
pkc_trust_list_element.class(3sec), **pkc_trusted_key.class(3sec)**.

pkc_free

Purpose

Frees storage allocated by certification routines

Synopsis

```
#include <pkc_api.h>

extern void pkc_free(
     void *);
```

Parameters

Input

storage
The storage to be freed.

Description

pkc_free(3sec) frees any storage that was allocated by any of the **pkc**_ routines called by an application. This routine should be used to release contiguous storage returned by any of the **pkc**_ routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_free_keyinfo

Purpose

Frees key information storage

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_free_keyinfo(
     keyinfo_t ** keybase);
```

Parameters

Input

keybase
Pointer to the **keyinfo_t** structure(s) to be freed.

Description

pkc_free_keyinfo(3sec) frees storage allocated by **pkc_retrieve_keyinfo(3sec)** for a **keyinfo_t** structure.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_free_trustbase

Purpose

Frees a trust base's storage

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_free_trustbase(
     trustbase_t ** base);
```

Parameters

Input

base Specifies trust base whose storage is to be freed.

Description

pkc_free_trustbase(3sec) frees the allocated storage for a trust base.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_free_trustlist

Purpose

Frees a trust list's storage

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_free_trustlist(
     trustlist_t ** tr_list);
```

Parameters

Input

tr_list Specifies trust list whose storage is to be freed.

Description

pkc_free_trustlist(3sec) frees the allocated storage for a trust list.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_generic_key_usage.class

Purpose

A class that expresses generic key usage

Member Data

Public

- **unsigned long permitted**

Member Functions

Public

- **pkc_generic_key_usage_t()**
  Takes the following argument:
  - **unsigned long permit_bits = 0xffffffffu**
- **char is_permitted() const**
  Takes the following argument:
  - **unsigned long check**
- **char is_permitted() const**
  Takes the following argument:
  - **const pkc_generic_key_usage_t & check**
- **void constrain()**
  Takes the following argument:
  - **unsigned long constraint**
- **void constrain()**
  Takes the following argument:
  - **const pkc_generic_key_usage_t & constraint**
- **void set()**
  Takes the following argument:
  - **unsigned long constraints**
- **pkc_generic_key_usage_t & operator = (unsigned long c)**

Description

pkc_generic_key_usage_t expresses various generic aspects of a key's usage.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**,

pkc_generic_key_usage.class(3sec)

pkc_pending_revocation.class(3sec), **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_get_key_certifier_count

Purpose

Returns number of key's certifying authorities

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_key_certifier_count(
     keyinfo_t  * keyinfobase
     unsigned key_index
     size_t * ca_count);
```

Parameters

Input

keyinfobase
A **keyinfo_t** structure containing information about the key.

key_index
The index of the key (ranging from 0 to *key_count* - 1).

Output

ca_count
Number of certifying authorities for the key.

Description

pkc_get_key_certifier_count(3sec) returns the number of certifying authorities in the certification path of the specified key.

The desired information is extracted by the routine from the **keyinfo_t** structure, which must first be obtained by the caller by a call to the **pkc_retrieve_keyinfo(3sec)** routine.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**,

pkc_init_trustlist(3sec), **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_get_key_certifier_info

Purpose

Returns information about a certifier

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_key_certifier_info(
     keyinfo_t * keyinfobase
     unsigned key_index
     unsigned certifier_index
     char ** certifier_name
     utc_t * certification_start
     utc_t * certification_expiration
     char * crl_valid
     utc_t * crl_last_seen
     utc_t * next_crl_expected);
```

Parameters

Input

keyinfobase
Information about the key (returned by **pkc_retrieve_keyinfo(3sec)**).

key_index
Index of the key.

certifier_index
Index of the certifier about whom information is desired.

Output

certifier_name
The name of the certifier.

certification_start
Time at which certification by this certifier starts.

certification_expiration
Time at which certification by this certifier ends.

crl_valid
If TRUE, there is a certificate revocation list for this certifier.

crl_last_seen
Time at which certificate revocation list was last seen.

next_crl_expected
Time at which next certificate revocation list is expected.

Description

pkc_get_key_certifier_info(3sec) returns information about a specific certifier from a key's certification path. Certifier 0 is the CA that vouched for the key; certifier 1 is the CA that vouched for certifier 0, etc. The total number of certifiers for a given key is returned by **pkc_get_key_certifier_count(3sec)**.

The desired information is extracted by the routine from the **keyinfo_t** structure, which must first be obtained by the caller by a call to the **pkc_retrieve_keyinfo(3sec)** routine.

Any of the return parameters may be passed as NULL if the corresponding information is not required.

Upon successful return, the *certifier_name* parameter will contain allocated storage which must be released with **pkc_free(3sec)** when the application has finished with it.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_get_key_count

Purpose

Returns number of keys for a principal

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_key_count(
     keyinfo_t * keyinfobase
     size_t * key_count);
```

Parameters

Input

keyinfobase
Key information returned by **pkc_retrieve_keyinfo(3sec)**.

Output

key_count
Number of keys.

Description

pkc_get_key_count(3sec) returns the number of keys within a **keyinfo_t** structure.

The desired information is extracted from the **keyinfo_t** structure, which must first be obtained by the caller by a call to the **pkc_retrieve_keyinfo(3sec)** routine.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_get_key_data

Purpose

Returns a public key

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_key_data(
     keyinfo_t * keyinfobase
     unsigned key_index
     unsigned char ** key_data
     size_t * key_length);
```

Parameters

Input

keyinfobase
Key information for the principal, returned by **pkc_retrieve_keyinfo(3sec)**.

key_index
Index (ranging from 0 to *key_count* - 1) of the key desired.

Output

key_data
The encoded public key.

key_length
Length of the key data returned.

Description

pkc_get_key_data(3sec) extracts an encoded public key from a **keyinfo_t** structure. *key_index* is the index of the key (ranging from 0 to *key_count* - 1).

The returned **key_data** is encoded as an ASN.1 BER **SubjectPublicKeyInfo** object (as defined in X.509).

The desired information is extracted by from the **keyinfo_t** structure, which must first be obtained by the caller by a call to the **pkc_retrieve_keyinfo(3sec)** routine.

Upon successful return, *key_data* will contain PKC-allocated storage which must be released with **pkc_free(3sec)** when the application has finished with it.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_get_key_trust_info

Purpose

Returns information about key trust

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_key_trust_info(
     keyinfo_t  * keyinfobase
     unsigned key_index
     certification_flags_t * flags
     uuid_t * key_domain
     unsigned long * key_usages);
```

Parameters

Input

keyinfobase
Key information, returned by **pkc_retrieve_keyinfo(3sec)**.

key_index
Index of the key, ranging from 0 to **keycount** - 1.

Output

flags Information about the trust that can be placed in the key (see below).

key_domain
Indicates domain of retrieved key. A value of **sec_pk_domain_unspecified** or **NULL** means that the policy does not distinguish keys by domain.

key_usages
Indicates usage key is intended for.

Description

pkc_get_key_trust_info(3sec) returns a set of flags describing the trust that can be placed in the key.

The desired information is extracted by the routine from the **keyinfo_t** structure, which must first be obtained by the caller by a call to the **pkc_retrieve_keyinfo(3sec)** routine.

The returned **certification_flags_t** structure describes the trust that can be placed in a returned key. It contains the following fields:

- **trust_type**
  A **trust_type_t** value, which will be one of the following:
  - **UNTRUSTED**
    No trust (e.g., unauthenticated).
  - **DIRECT_TRUST**
    Direct trust via third party (e.g., authenticated registry).
  - **CERTIFIED_TRUST**

Trust certified by caller's trust base.

- **missing_crls**

  A **char**; its value is TRUE (not 0) if one or more CRLs are missing.

- **revoked**

  A **char** whose value is TRUE (not 0) if any certificate has been revoked (even if it was still valid at the retrieval time).

If **key_domain** and **key_usages** are passed as non-**NULL** pointers, upon successful return these parameters will describe the domain and permitted usage(s) of the specified key. Policies that do not distinguish keys according to domain will indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage will indicate all usages are permitted.

The returned **key_usages** is a bit mask which describes the usage(s), if any, which the key is restricted to. The value is formed by AND-ing together one or more of the following constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A returned **key_usages** value of **NULL** (or a value with all bits set) means that the key is suitable for any usage.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_get_registered_policies

Purpose

Returns all registered trust policies

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_get_registered_policies(
     gss_OID_set * oid_set);
```

Parameters

Output

oid_set
A set of OIDs which represent all installed policies.

Description

pkc_get_registered_policies(3sec) returns a set of OIDs, which point to all currently installed policies (that is, all pre-loaded policies, plus any policies that have been installed via the policy registration API).

An application will call this routine once during its lifetime. After successfully making the call, the application can choose to use the returned OIDs in a call to **pkc_init_trustbase(3sec)**, etc.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_init_trustbase

Purpose

Initializes a trust base

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_init_trustbase(
     trustlist_t ** tr_list
     gss_OID policy_oid
     utc_t time
     selection_t * sel
     trustbase_t ** base);
```

Parameters

Input

tr_list Specifies trust list on the basis of which the trust base is to be initialized.

policy_oid
Specifies policy to use.

time Specifies time at which the public key is to be valid. Can be 0.

sel Must be set to 0.

Output

base Initialized trust base.

Description

pkc_init_trustbase(3sec) initializes the initial trust base to include all the certificates initially trusted, given the initial set of trusted certificates. This routine will also store the cross-certificate pair certificates found during the creation of the trust base.

Upon successful return, *base* will contain a PKC-allocated trust base structure, which should be released with **pkc_free_trustbase(3sec)** when the application has finished with it.

Users will normally call the **pkc**_ routines in the following order:

1. **pkc_get_registered_policies(3sec)**
   Called once for the lifetime of the application.
2. **pkc_init_trustlist(3sec)**
3. **pkc_append_to_trustlist(3sec)**
   Called one or more times.
   Note that steps 2 and 3 together build up an initial trust list.
4. **pkc_init_trustbase(3sec)**
   A trust base is computed, given an initial trust list.
5. **pkc_retrieve_keylist(3sec)**

Called one or more times, for each individual's public key that needs to be looked up.

6. **pkc_free_trustlist(3sec)**
7. **pkc_free_trustbase(3sec)**

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_init_trustlist

Purpose

Creates an empty trust list

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_init_trustlist(
     trustlist_t ** tr_list);
```

Parameters

Input

tr_list A PKC-allocated data structure which contains the initialized trust list.

Description

pkc_init_trustlist(3sec) creates an empty trust list. If *tr_list* is empty, returns **pkc_s_asn_bad_param**; if memory cannot be allocated, returns **pkc_s_nomem**; otherwise, returns **pkc_s_success**.

Upon successful return, *tr_list* will contain a PKC-allocated data structure which must be released with **pkc_free_trustlist(3sec)** when the application has finished with it.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_retrieve_keyinfo(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_key_policies.class

Purpose

A class that expresses policy rules and operations

Member Functions

Public

- **pkc_key_policies_t(void)**
  Initializes to ″all policies OK″.
- **unsigned32 set()**
  Takes the following argument:
  – **const pkc_key_policy_t & pol**
- **unsigned32 set()**
  Adds an allowed policy. Takes the following argument:
  – **const gss_OID pol**
- **unsigned32 set_none(void)**
  Sets ″no policies permitted″.
- **unsigned32 set_all(void)**
  Sets ″all policies permitted″.
- **unsigned32 constrain()**
  Takes the following argument:
  – **const pkc_key_policies_t & pol**
- **pkc_key_policies_t & operator = (const pkc_key_policies_t & pol)**

Description

pkc_key_policies_t embodies rules and operations for key policies.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_key_policy.class

Purpose

Key policy class

Member Data

Public

- **gss_OID value**

Member Functions

Public

- **pkc_key_policy_t(void)**
- **virtual ~pkc_key_policy_t()**
- **pkc_key_policy_t & operator = (const pkc_key_policy_t & o)**
- **pkc_key_policy_t & operator = (const gss_OID & o)**

Description

pkc_key_policy_t embodies a key policy.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_key_usage.class

Purpose

A class that expresses key usage rules

Member Functions

Public

- **pkc_key_usage_t & operator = ()**
  Takes the following argument:
  - **unsigned long c**
- **pkc_key_usage_t()**
  Takes the following argument:
  - **unsigned long c = 0xffffffflu**

Description

pkc_key_usage_t contains usage rules for a key.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Parent Class

This class is derived from **pkc_generic_key_usage_t**.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_lookup_element_in_trustlist

Purpose

Retrieves a specified key

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_lookup_element_in_trustlist(
     pkc_trust_list_t * trust_list
     const pkc_trust_list_element_t  ** key
     unsigned long key_id);
```

Parameters

Input

trust_list
Specifies the trust list

key_id Specifies ID of key to return.

Output

key A pointer to the returned key.

Description

pkc_lookup_element_in_trustlist(3sec) takes a trust list and a key id, and returns a pointer to the specified key (if the key actually is in the trust list).

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_copy_trustlist(3sec)**, **pkc_delete_trustlist(3sec)**, **pkc_display_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**,

pkc_constraints.class(3sec), **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_lookup_key_in_trustlist

Purpose

Searches a trust list for the specified key

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_lookup_key_in_trustlist(
     pkc_trust_list_t * trust_list
     const pkc_trusted_key_t ** key
     unsigned long key_id);
```

Parameters

Input

trust_list
Specifies the trust list to search.

key_id Specifies ID of key to return.

Output

key The returned key.

Description

pkc_lookup_key_in_trustlist(3sec) searches the specified trust list for the specified key. In the returned key, the caller will find the following fields.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Fields from the Certificate

- **start_date**
  A **utc_t**
- **end_date**
  A **utc_t**
- **ca_usages**
  A **pkc_ca_key_usage_t**
- **user_usages**
  A **pkc_key_usage_t**
- **policies**
  A **pkc_key_policies_t**
- **constraints**
  A **pkc_constraints_t**

Flags:

- **trusted** (A **char**)
  Expresses whether this entry is trusted (*a priori*).
- **certified** (A **char**)
  Expresses whether this key is certified by another entry.
- **certified_by** (**x500name**)
  Name of the CA that certified this key.
- **serial_number** (**asn_integer**)
  Serial number of certifying certificate.

The following fields are copied from the certifying key entry:

- **certified_start_date** (**utc_t**)
- **certified_end_date** (**utc_t**)
- **certified_usages** (**pkc_ca_key_usage_t**)
- **certified_policies** (**pkc_key_policies_t**)
- **certified_constraints** (**pkc_constraints_t**)
- **revoked** (**char**)
  Non-zero if the certifying certificate has been revoked.
- **revocation_date** (**utc_t**)
  Date from which certifier revocation is effective.
- *key_id* (**unsigned long**)
  An ID identifying this key entry.
- **ca_key_id** (**unsigned long**)
  The ID of the key that certified this one. 0 means direct trust.
- **old_key_id** (**unsigned long**)
  Temporary storage for use while copying
- **old_ca_key_id** (**unsigned long**)

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**,

pkc_trust_list_element.class(3sec), **pkc_trusted_key.class(3sec)**.

pkc_lookup_keys_in_trustlist

Purpose

Searches trust list for keys

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_lookup_keys_in_trustlist(
     pkc_trust_list_t * trust_list
     const pkc_trusted_key_t ** key
     size_t * key_count
     const x500name & owner
     utc_t * key_time
     const pkc_generic_key_usage_t * usages );
```

Parameters

Input

trust_list
Specifies trust list to search.

owner Specifies principal whose keys are to be searched for.

key_time
Specifies time of ownership to search for.

usages
Specifies usage to search for.

Output

key Array of pointers to keys found.

key_count
Number of keys found.

Description

pkc_lookup_keys_in_trustlist(3sec) searches the specified trust list for keys owned by the specified principal at the specified time for the specified usage. The keys are returned in an array of pointers to **pkc_trusted_key_t** objects, which is allocated on the heap. The pointers point to elements within the trust list; thus the caller should copy into allocated storage if they are expected to remain valid after the deletion of the trust list.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_name_subord_constraint.class

Purpose

Constraint rules and operations class

Member Data

Public

- **pkc_name_subord_constraint_t * next**
- **pkc_name_subord_constraint_t * prev**
- **unsigned long constraint**
- **unsigned skipCerts**

Member Functions

Public

- **void unlink(void)**
- **~pkc_name_subord_constraint_t()**
- **pkc_name_subord_constraint_t()**

  Takes the following argument:

  – **pkc_name_subord_constraints_t * theRoot**
- **pkc_name_subord_constraint_t(void)**
- **void set_constraint()**

  Takes the following argument:

  – **unsigned long c**
- **void set_skipCerts()**

  Takes the following argument:

  – **unsigned c**
- **void get_next_link_constraint() const**

  Generates a new name subordination constraint that will be applicable to a certificate issued by the subject of this constraint. Takes the following argument:

  – **pkc_name_subord_constraint_t ** new_constraint**
- **char is_permitted() const**

  Takes the following arguments:

  – **const x500name & issuer_name**

  – **const x500name & subject_name**

  Return values have following meanings:

  1 Permitted

  0 Forbidden

  −1 Not relevant

  −2 Relevant, but explicit permission is required from another subordination constraint.

pkc_name_subord_constraint.class(3sec)

Description

pkc_name_subord_constraint_t contains name-subordinate constraint rules and operations for a certificate.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_name_subord_constraints.class

Purpose

A class that expresses subordinate constraints on a name

Member Data

Public

- **pkc_name_subord_constraint_t * first**
- **pkc_name_subord_constraint_t * last**

Member Functions

Public

- **pkc_name_subord_constraints_t & operator = (const pkc_name_subord_constraints_t & o)**
- **pkc_name_subord_constraint_t * first**
- **pkc_name_subord_constraint_t * last**
- **pkc_name_subord_constraints_t(void)**
- **~pkc_name_subord_constraints_t()**
- **char is_permitted() const**

  Takes the following arguments:

  – **const x500name & ca_name**
  – **const x500name & subject_name**
- **void get_next_link_constraint() const**

  Takes the following argument:

  – **pkc_name_subord_constraints_t ** new_constraints**

Description

pkc_name_subord_constraints_t embodies a set of subordinate constraints on a name.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_name_subtree_constraint.class

Purpose

A class that expresses a subtree constraint on a name

Member Data

Public

- **pkc_name_subtree_constraint_t * next**
- **pkc_name_subtree_constraint_t * prev**
- **x500name * base**
- **x500name * chopBefore**
- **x500name * chopAfter**
- **unsigned minimum**
- **unsigned maximum**

Member Functions

Public

- **void unlink(void)**
- **virtual ~pkc_name_subtree_constraint_t()**
- **pkc_name_subtree_constraint_t()**
  Takes the following argument:
  – **pkc_name_subtree_constraints_t * theRoot**
- **void set_base()**
  Takes the following argument:
  – **const x500name & n**
- **void set_chopBefore()**
  Takes the following argument:
  – **const x500name & n**
- **void set_chopAfter()**
  Takes the following argument:
  – **const x500name & n**
- **void set_minimum()**
  Takes the following argument:
  – **unsigned n**
- **void set_maximum()**
  Takes the following argument:
  – **unsigned n**
- **char is_permitted() const**
  Takes the following arguments:
  – **const x500name & issuer_name**
  – **const x500name & subject_name**

  Return values have the following meanings:

  1 Permitted.

0 Forbidden.

−1 Not relevant.

−2 Relevant, but explicit permission is required from another subtree constraint.

- **void get_next_link_constraint() const**

  Generates a new name subtree constraint that will be applicable to a certificate issued by the subject of this constraint. Takes the following argument:

  – **pkc_name_subtree_constraint_t ** new_constraint**

Description

pkc_name_subtree_constraint_t embodies a subtree constraint on a name.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_name_subtree_constraints.class

Purpose

A class that expresses a set of subtree constraints on a name

Member Data

Public

- **pkc_name_subtree_constraint_t * first**
- **pkc_name_subtree_constraint_t * last**

Member Functions

Public

- **pkc_name_subtree_constraints_t & operator = (const pkc_name_subtree_constraints_t & o)**
- **pkc_name_subtree_constraint_t * first**
- **pkc_name_subtree_constraint_t * last**
- **pkc_name_subtree_constraints_t(void)**
- **virtual ~pkc_name_subtree_constraints_t()**
- **char is_permitted() const**

  Takes the following arguments:

  – **const x500name & ca_name**

  – **const x500name & subject_name**
- **void get_next_link_constraint() const**

  Takes the following argument:

  – **pkc_name_subtree_constraints_t ** new_constraints**

Description

pkc_name_subtree_constraints_t embodies a set of subtree constraints on a name.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_pending_revocation.class

Purpose

Class of certificates awaiting revocation

Member Data

Public

- **SignedCertificateList crl**

Member Functions

Public

- **pkc_pending_revocation_t()**

  Takes the following arguments:

  – **const SignedCertificateList & crl**

  – **pkc_revocation_list_t * the_root = NULL**
- **pkc_pending_revocation_t & operator = (const pkc_pending_revocation_t & o)**
- **virtual void unlink(void)**
- **virtual ~pkc_pending_revocation_t()**

Description

pkc_pending_revocation_t contains certificates awaiting revocation. Has the **friend** class **pkc_revocation_list_t**.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_plcy_delete_keyinfo

Purpose

Frees public key storage

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_delete_keyinfo(
     gss_OID policy
     void ** keys_handle);
```

Parameters

Input

policy Specifies policy module.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

Description

pkc_plcy_delete_keyinfo(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*delete_keyinfo)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_delete_trustbase

Purpose

Frees trust base storage

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_delete_trustbase(
     gss_OID policy
     void ** trust_base_handle);
```

Parameters

Input

policy Specifies policy module.

trust_base_handle
Specifies trust base to be deleted.

Description

pkc_plcy_delete_trustbase(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*delete_trustbase)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_establish_trustbase

Purpose

Establishes a trust base

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_establish_trustbase(
     gss_OID policy
     const pkc_trust_list_t & initial_trust
     const utc_t * date
     char initial_explicit_policy_required
     void ** trust_base_handle);
```

Parameters

Input

policy Specifies policy to use.

initial_trust
Specifies the initial set of trusted keys.

date Specifies time for which information is to be returned.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

trust_base_handle
The initialized trust base.

Description

pkc_plcy_establish_trustbase(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(establish_trustbase()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

This is a one-time call made by an application to initialize a trust base. It returns an extended trust list. After this call is made, the application can call **pkc_retrieve_keyinfo(3sec)** to obtain the public keys of any particular principal.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_key_certifier_count

Purpose

Returns number of a key's certifying authorities

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_get_key_certifier_count(
     gss_OID policy
     void * keys_handle
     unsigned key_index
     size_t * ca_count);
```

Parameters

Input

policy Specifies policy desired.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

key_index
Specifies the key whose number of certifying authorities is to be returned.

Output

ca_count
Number of certifying authorities for the key.

Description

pkc_plcy_get_key_certifier_count(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*get_key_certifier_count)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**,

pkc_plcy_get_registered_policies(3sec), **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_key_certifier_info

Purpose

Returns information about a key's certifier

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_get_key_certifier_info(
     gss_OID policy
     void * keys_handle
     unsigned key_index
     unsigned ca_index
     char ** certifier_name
     utc_t * certification_start
     utc_t * certification_expiration
     char  * is_crl_valid
     utc_t * last_crl_seen
     utc_t * next_crl_expected);
```

Parameters

Input

policy The policy desired.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

key_index
Index of the key.

ca_index
Index of the certifier about whom information is desired.

Output

certifier_name
The name of the certifier.

certification_start
Time at which certification by this certifier starts.

certification_expiration
Time at which certification by this certifier ends.

is_crl_valid
If TRUE, there is a certificate revocation list for this certifier.

last_crl_seen
Time at which certificate revocation list was last seen.

next_crl_expected
Time at which next certificate revocation list is expected.

Description

pkc_plcy_get_key_certifier_info(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*get_key_certifier_info)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_key_count

Purpose

Returns number of keys for a principal

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_get_key_count(
     gss_OID policy
     void * keys_handle
     size_t * key_count);
```

Parameters

Input

policy Specifies policy desired.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

Output

key_count
The number of principal's keys.

Description

pkc_plcy_get_key_count(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*get_key_count)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_key_data

Purpose

Returns a public key

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_get_key_data(
     gss_OID policy
     void * keys_handle
     unsigned key_index
     unsigned char ** key_data
     size_t * key_length);
```

Parameters

Input

policy Policy desired.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

key_index
Specifies index of key desired.

Output

key_data
The public key requested.

key_length
Length of *key_data*.

Description

pkc_plcy_get_key_data(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*get_key_data)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_key_trust

Purpose

Returns information about trust in a key

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_get_key_trust(
     gss_OID policy
     void * keys_handle
     unsigned key_index
     certification_flags_t * flags
     uuid_t * key_domain
     unsigned long * key_usages);
```

Parameters

Input

policy Specifies policy.

keys_handle
A policy specific structure, obtained from a call to **pkc_plcy_retrieve_keyinfo(3sec)**.

key_index
Specifies key about which trust information is requested.

Output

flags Information about the trust that can be placed in the key (see below).

key_domain
Indicates domain of retrieved key. A value of **sec_pk_domain_unspecified** or **NULL** means that the policy does not distinguish keys by domain.

key_usages
Indicates usage key is intended for.

Description

pkc_plcy_get_key_trust(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(*get_key_data)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

The returned **certification_flags_t** structure describes the trust that can be placed in the key. It contains the following fields:

- **trust_type**
  A **trust_type_t** value, which will be one of the following:
  – **UNTRUSTED**
    No trust (e.g., unauthenticated).
  – **DIRECT_TRUST**
    Direct trust via third party (e.g., authenticated registry).

– **CERTIFIED_TRUST**
  Trust certified by caller's trust base.

If **key_domain** and **key_usages** are passed as non-**NULL** pointers, upon successful return these parameters will describe the domain and permitted usage(s) of the specified key. Policies that do not distinguish keys according to domain will indicate a domain of **sec_pk_domain_unspecified**; policies that do not distinguish keys according to usage will indicate all usages are permitted.

The returned **key_usages** is a bit mask which describes the usage(s), if any, which the key is restricted to. The value is formed by AND-ing together one or more of the following constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A returned **key_usages** value of **NULL** (or a value with all bits set) means that the key is suitable for any usage.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_registered_policies(3sec)**,

pkc_plcy_lookup_policy(3sec), **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_get_registered_policies

Purpose

Returns OID set describing registered policies

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_plcy_reg.h>

unsigned32 pkc_plcy_get_registered_policies(
     gss_OID_set * oid_set);
```

Parameters

Output

oid_set
: A pointer to an OID set describing the currently registered policy implementations.

Description

pkc_plcy_get_registered_policies(3sec) returns an OID set describing the currently registered policy implementations.

pkc_plcy_lookup_policy(3sec) can be called to return details about a specific policy implementation.

Policy modules are identified by OIDs (object identifiers). A policy module is accessed by passing its identifying OID to **pkc_plcy_lookup_policy(3sec)**.

Return Values

pkc_s_success
: Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_lookup_policy

Purpose

Returns a policy module descriptor block

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_plcy_reg.h>

unsigned32 pkc_plcy_lookup_policy(
     gss_OID oid
     pkc_policy_t * details);
```

Parameters

Input

oid An OID identifying a currently registered policy module.

Output

details A pointer to a policy module descriptor block.

Description

pkc_plcy_lookup_policy(3sec) returns a policy module descriptor block for the specified policy, and leaves the policy list unlocked. Calling this routine is the preferred way of obtaining information about a registered policy implementation.

The complete list of registered policies may be obtained by calling **pkc_get_registered_policies(3sec)**.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**, **pkc_register_policy(3sec)**.

pkc_plcy_register_policy

Purpose

Registers a policy module

Synopsis

```
#include <dce/pkc_base.h>
#include <dce/pkc_plcy_reg.h>

unsigned32 pkc_plcy_register_policy(
     pkc_policy_t * plcy
     int replacement_policy);
```

Parameters

Input

plcy A pointer to the policy module structure to be registered.

replacement_policy
Specifies how the registration is to be handled if an implementation of the policy is already registered. There are three possible values:

PKC_REPLACE_NONE
Specifies that an error should be returned if an implementation of the policy is already registered.

PKC_REPLACE_ENTRYPOINTS
Specifies that only entrypoints that the original implementation (if any) did not provide should be replaced. (Note that this value is not currently supported.)

PKC_REPLACE_ALL
Specifies that the new implementation should replace the existing one, if any.

Description

pkc_plcy_register_policy(3sec) registers a policy module, in the form of a properly declared **pkc_policy_t** data structure, which contains identifying information about the module as well as entry points to all of the module's functions.

Calling this routine will cause the module passed to it to be registered among the system's policy modules; it can then be accessed by other applications via the high level certification routines.

C++ must be used to perform policy registration.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_plcy_retrieve_keyinfo(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_plcy_retrieve_keyinfo

Purpose

Retrieves keys for specified principal

Synopsis

```
#include <pkc_plcy.h>

unsigned32 pkc_plcy_retrieve_keyinfo(
     gss_OID policy
     const void * trust_base_handle
     const x500name & subjectName
     const utc_t * date
     const uuid_t & desired_domain
     pkc_key_usage_t & desired_usage
     char initial_explicit_policy_required
     void ** keys_handle);
```

Parameters

Input

policy Specifies the policy being interrogated.

trust_base_handle
Expresses the caller's initial trust.

subjectName
Specifies the desired subject name (principal name).

date Specifies time for which information is to be returned.

desired_domain
Specifies particular domain to which the key-search operation should be restricted. Specify **sec_pk_domain_unspecified** or **NULL** to indicate that keys for any domain should be retrieved.

desired_usage
Allows the user to restrict the key-search operation to keys intended for one or more specific usages.

initial_explicit_policy_required
Specifies whether the initial certificate must explicitly contain the active policy in its policies field.

Output

keys_handle
The returned key information.

Description

pkc_plcy_retrieve_keyinfo(3sec) searches the list of registered policies for implementations of the specified policy. If found, the implementation is opened, if necessary, and its **(retrieve_key_info)()** function is invoked. Necessary mutex protection around non-thread safe policy implementations is provided.

The **desired_usage** parameter consists of a bit mask, formed by AND-ing together one or more of the constants:

PKC_KEY_USAGE_AUTHENTICATION
Specifies keys that can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
Specifies keys that can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
Specifies keys that can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
Specifies keys that can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
Specifies keys that can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
Specifies keys that can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
Specifies keys that can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
Specifies keys that can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
Specifies keys that can be used to sign transactions

A **NULL** can be specified for **desired_usage** to indicate that keys for any usage should be retrieved.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_plcy_intro(3sec)**, **pkc_plcy_delete_keyinfo(3sec)**, **pkc_plcy_delete_trustbase(3sec)**, **pkc_plcy_establish_trustbase(3sec)**, **pkc_plcy_get_key_certifier_count(3sec)**, **pkc_plcy_get_key_certifier_info(3sec)**, **pkc_plcy_get_key_count(3sec)**, **pkc_plcy_get_key_data(3sec)**, **pkc_plcy_get_key_trust(3sec)**, **pkc_plcy_get_registered_policies(3sec)**, **pkc_plcy_lookup_policy(3sec)**, **pkc_plcy_retrieve_key(3sec)**, **pkc_register_policy(3sec)**.

pkc_retrieve_keyinfo

Purpose

Returns information about a key

Synopsis

```
#include <pkc_api.h>

unsigned32 pkc_retrieve_keyinfo(
     trustbase_t * base
     char * name
     utc_t * key_date
     uuid_t * key_domain
     unsigned long * key_usages
     selection_t * sel
     keyinfo_t ** keyinfobase);
```

Parameters

Input

base The trust base, returned by **pkc_init_trustbase(3sec)**.

name Principal name.

key_date
Specifies time for which information is to be returned.

key_domain
Allows the user to restrict the key-search operation to keys for a particular domain. Specify **sec_pk_domain_unspecified** or **NULL** to indicate that keys for any domain should be retrieved.

key_usages
Allows the user to restrict the key-search operation to keys intended for one or more specific usages.

sel Must be 0 (currently ignored).

Output

keyinfobase
The returned key information.

Description

pkc_retrieve_keyinfo(3sec) returns a **keyinfo_t** structure describing the set of trusted keys that are valid for the specified principal at the specified date, under any additional constraints specified in *sel*.

The **key_usages** parameter consists of a bit mask, formed by AND-ing together one or more of the constants:

PKC_KEY_USAGE_AUTHENTICATION
The key can be used to authenticate a user

PKC_KEY_USAGE_INTEGRITY
The key can be used to provide integrity protection

PKC_KEY_USAGE_KEY_ENCIPHERMENT
The key can be used to encrypt user keys

PKC_KEY_USAGE_DATA_ENCIPHERMENT
The key can be used to encrypt user data

PKC_KEY_USAGE_KEY_AGREEMENT
The key can be used for key-exchange

PKC_KEY_USAGE_NONREPUDIATION
The key can be used for non-repudiation

PKC_CAKEY_USAGE_KEY_CERT_SIGN
The key can be used to sign key certificates

PKC_CAKEY_USAGE_OFFLINE_CRL_SIGN
The key can be used to sign CRLs

PKC_CAKEY_USAGE_TRANSACTION_SIGN
The key can be used to sign transactions

A **NULL** can be specified for **key_usages** to indicate that keys for any usage should be retrieved.

This routine must be called before any of the following routines can be called:

- **pkc_get_key_count(3sec)**
- **pkc_get_key_data(3sec)**
- **pkc_get_key_trust_info(3sec)**
- **pkc_get_key_certifier_count(3sec)**
- **pkc_get_key_certifier_info(3sec)**

Upon successful return, *keyinfobase* will contain a **keyinfo_t** structure which must be passed in calls to the above routines, which then extract and return the requested information.

The **keyinfo_t** structure must be released by a call to **pkc_free_keyinfo(3sec)** when the application has finished with it.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keylist(3sec)**.

pkc_retrieve_keylist

Purpose

Retrieves all keys for a principal

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_retrieve_keylist(
     trustbase_t * base
     char * name
     trusted_key_t ** out_keys
     size_t * no_of_keys);
```

Parameters

Input

base Specifies trust base from which to retrieve keys.

name Specifies principal whose keys are to be retrieved.

Output

out_keys
Keys retrieved.

no_of_keys
Number of keys retrieved.

Description

Given an initialized trust base, **pkc_retrieve_keylist(3sec)** returns all public keys for the principal specified.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_intro(3sec)**, **pkc_append_to_trustlist(3sec)**, **pkc_free(3sec)**, **pkc_free_keyinfo(3sec)**, **pkc_free_trustbase(3sec)**, **pkc_free_trustlist(3sec)**, **pkc_get_key_certifier_count(3sec)**, **pkc_get_key_certifier_info(3sec)**, **pkc_get_key_count(3sec)**, **pkc_get_key_data(3sec)**, **pkc_get_key_trust_info(3sec)**, **pkc_get_registered_policies(3sec)**, **pkc_init_trustbase(3sec)**, **pkc_init_trustlist(3sec)**, **pkc_retrieve_keyinfo(3sec)**.

pkc_revocation.class

Purpose

A class that expresses certificate revocation operations

Member Data

Public

- **x500name certIssuer**
- **asn_integer certSerialNumber**
- **utc_t certRevocationDate**

Member Functions

Public

- **pkc_revocation_t()**

  Takes the following arguments:
  - **const x500name & issuer**
  - **const asn_integer & serialNumber**
  - **utc_t revocationDate**
  - **pkc_revocation_list_t * the_root = NULL**
- **pkc_revocation_t & operator = (const pkc_revocation_t & o)**
- **virtual void unlink(void)**
- **virtual ~pkc_revocation_t()**

Description

pkc_revocation_t embodies certificate revocation operations. Has **friend** class **pkc_revocation_list_t**.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_revocation_list.class

Purpose

Revoked certificates list class

Member Data

Public

- **pkc_revocation_t * first**
- **pkc_revocation_t * last**
- **pkc_pending_revocation_t * first_pending**
- **pkc_pending_revocation_t * last_pending**

Member Functions

Public

- **unsigned32 get_revocation_date() const**
  Takes the following arguments:
  - **const pkc_revocation_t & o**
  - **utc_t * revocationDate**
- **unsigned32 get_revocation_date() const**
  Takes the following arguments:
  - **const x500name & issuer**
  - **const asn_integer & serialNumber**
  - **utc_t * revocationDate**
- **unsigned32 add_revocation()**
  Takes the following argument:
  - **const pkc_revocation_t & o**
- **unsigned32 add_revocation()**
  Takes the following arguments:
  - **const x500name & issuer**
  - **const asn_integer & serialNumber**
  - **const utc_t * revocationDate**
- **unsigned32 add_crl()**
  Takes the following argument:
  - **const SignedCertificateList & crl**
- **unsigned32 add_key()**
  Takes the following arguments:
  - **pkc_trust_list_t * trust_list**
  - **const SubjectPublicKeyInfo & key**
  - **const x500name & subject**
  - **const utc_t & start_date**
  - **const utc_t & end_date**
  - **const pkc_ca_key_usage_t * usages = NULL**
- **pkc_revocation_list_t(void)**

- **virtual ~pkc_revocation_list_t()**
- **void empty(void)**

Description

pkc_revocation_list_t embodies a list of revoked certificates and their dates.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_revoke_certificate

Purpose

Revokes key and dependents from specified trust list

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_revoke_certificate(
     pkc_trust_list_t * trust_list
     const x500name & issued_by
     const asn_integer & serial_no
     utc_t * invalidate_from );
```

Parameters

Input

trust_list
Specifies trust list from which to revoke keys.

issued_by
Specifies issuer whose keys are to be revoked.

serial_no
Specifies serial number of key to revoke.

invalidate_from
Specifies time after which keys will be invalid.

Description

pkc_revoke_certificate(3sec) applies the specified revocation to the specified trust list (i.e. revokes a key and all dependent keys). If *invalidate_from* is NULL, the key is completely revoked; if a valid UTC time is provided, the key is revoked from that time on. The revocation is stored within the trust list, and any subsequent attempts to add the certificate will be rejected.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificates(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_revoke_certificates

Purpose

Revokes a key and all dependent keys

Synopsis

```
#include <pkc_certs.h>

unsigned32 pkc_revoke_certificates(
     pkc_trust_list_t * trust_list
     const SignedCertificateList * crl);
```

Parameters

Input

trust_list
Specifies list from which keys are to be revoked.

crl Specifies keys to revoke.

Description

pkc_revoke_certificates(3sec) applies the specified revocations to the specified trust list (i.e. revokes a key and all dependent keys). The revocations are stored within the trust list, and any subsequent attempts to add a revoked certificate will be rejected.

This routine is a C++ interface. C++ must be used to perform direct certificate manipulation.

See also the contents of the **asn.h** and **x509.h** header files, which define some of the basic types used by the low-level certificate manipulation routines.

Return Values

pkc_s_success
Operation successfully completed.

Errors

Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Related Information

Functions: **pkc_add_trusted_key(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**. Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**,

pkc_name_subtree_constraint.class(3sec),
pkc_name_subtree_constraints.class(3sec),
pkc_pending_revocation.class(3sec), **pkc_revocation.class(3sec)**,
pkc_revocation_list.class(3sec), **pkc_trust_list.class(3sec)**,
pkc_trust_list_element.class(3sec), **pkc_trusted_key.class(3sec)**.

pkc_trust_list.class

Purpose

A class that expresses certificate trust list operations

Member Data

Public

- **pkc_trust_list_element_t * first**
- **pkc_trust_list_element_t * last**
- **pkc_revocation_list_t revocation_list**
  List of revocations

Member Functions

Public

- **pkc_trust_list_t(void)**
- **void empty(void)**
- **virtual ~pkc_trust_list_t()**
- **pkc_trust_list_t & operator = (const pkc_trust_list_t & o)**
- **unsigned32 fixup_links(void)**
- **unsigned32 find_certified_key()**
  Returns the first key entry that was created from the specified certificate. Call **find_next_certified_key()** to return the next such key. Takes the following arguments:
  – **const x500name &** *certifier*
  – **const asn_integer &** *certifying_serial_no*
  – **pkc_trust_list_element_t **** *key*
- **unsigned32 find_next_certified_key()**
  Returns the next key entry that was created from the same certificate as the current entry. *key* is both an input and an output. Takes the following argument:
  – **pkc_trust_list_element_t **** *key*
- **unsigned32 find_certified_key_by_id()**
  Returns the first key entry that was certified by the specified key id. Call **find_next_certified_key_by_id()** to return the next such key. Takes the following arguments:
  – **unsigned long** *ca_key_id*
  – **pkc_trust_list_element_t **** *key*
- **unsigned32 find_next_certified_key_by_id()**
  Returns the next key entry that was certified by the same key as the current entry. *key* is both an input and an output. Takes the following argument:
  – **pkc_trust_list_element_t **** *key*

Description

pkc_trust_list_t embodies rules and operations for a certificate trust list. This class has the **friend** class **pkc_trust_list_element_t**.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_trust_list_element.class

Purpose

Public key class

Member Data

Public

- **pkc_trust_list_element_t * next**
- **pkc_trust_list_element_t * prev**

Member Functions

Public

- **void unlink(void)**
- **pkc_trust_list_element_t()**

  Takes the following argument:
  - **pkc_trust_list_t &** *the_root*
- **pkc_trust_list_element_t()**

  Takes the following arguments:
  - **pkc_trust_list_t &** *the_root*
  - **utc_t** *startDate*
  - **utc_t** *endDate*
  - **pkc_ca_key_usage_t** *caUsages*
  - **pkc_key_usage_t** *userUsages*
  - **pkc_key_policies_t** *keyPolicies*
  - **pkc_constraints_t** *keyConstraints*
- **virtual ~pkc_trust_list_element_t()**
- **unsigned32 apply_revocation()**

  Apply a revocation to this key, starting at the specified date. If revocation_date is NULL, the key is completely revoked: this key, and all keys dependent on it will be revoked. Takes the following argument:
  - **utc_t *** *revocation_date*

Description

pkc_trust_list_element_t defines a key.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Parent Class

This class is derived from the **pkc_trusted_key_t** class. It has as **friend** class **pkc_trust_list_t** (a list of trusted keys).

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trusted_key.class(3sec)**.

pkc_trusted_key.class

Purpose

Trusted public key class

Member Data

Public

- **SubjectPublicKeyInfo value**
- **x500name owner**

Member Functions

Public

- **pkc_trusted_key_t(void)**
- **pkc_trusted_key_t()**
  Takes following arguments:
  - **utc_t** *startDate*
  - **utc_t** *endDate*
  - **pkc_ca_key_usage_t** *caUsages*
  - **pkc_key_usage_t** *userUsages*
  - **pkc_key_policies_t** *keyPolicies*
  - **pkc_constraints_t** *keyConstraints*
- **unsigned32 get_start_date() const**
  Takes the following argument:
  - **utc_t *** *start_date*
- **unsigned32 get_end_date() const**
  Takes the following argument:
  - **utc_t *** *end_date*
- **unsigned32 get_usages() const**
  Takes the following argument:
  - **pkc_key_usage_t *** *user_usages*
- **unsigned32 get_ca_usages() const**
  Takes the following argument:
  - **pkc_ca_key_usage_t *** *ca_usages*
- **unsigned32 get_key_policies() const**
  Takes the following argument:
  - **pkc_key_policies_t *** *policies*
- **unsigned32 get_constraints() const**
  Takes the following argument:
  - **pkc_constraints_t *** *constraints*
- **unsigned32 get_certifier() const**
  Takes the following argument:
  - **x500name &** *name*
- **unsigned32 get_certifier() const**

Takes the following argument:
  - **pkc_trusted_key_t **** *ca*
- **char valid_at() const**
  Takes the following argument:
  - **utc_t *** *time*
- **pkc_trusted_key_t & operator = (const pkc_trusted_key_t & o)**
- **char may_certify() const**
  Takes the following arguments:
  - **const x500name &** *subject*
  - **unsigned long** *usage* **= PKC_CAKEY_USAGE_KEY_CERT_SIGN**
- **char may_certify() const**
  Takes the following arguments:
  - **const x500name & subject**
  - **const pkc_ca_key_usage_t & usage**

Description

pkc_trusted_key_t is a class that expresses trust in a public key. It is very much like a certificate, but with trust pre-established, rather than based on a signature.

This class has the **friend** class **pkc_trust_list_t**.

The certificate manipulation routines are a C++ interface. C++ must be used to perform direct certificate manipulation.

Related Information

Classes: **pkc_ca_key_usage.class(3sec)**, **pkc_constraints.class(3sec)**, **pkc_generic_key_usage.class(3sec)**, **pkc_key_policies.class(3sec)**, **pkc_key_policy.class(3sec)**, **pkc_key_usage.class(3sec)**, **pkc_name_subord_constraint.class(3sec)**, **pkc_name_subord_constraints.class(3sec)**, **pkc_name_subtree_constraint.class(3sec)**, **pkc_name_subtree_constraints.class(3sec)**, **pkc_pending_revocation.class(3sec)**, **pkc_revocation.class(3sec)**, **pkc_revocation_list.class(3sec)**, **pkc_trust_list.class(3sec)**, **pkc_trust_list_element.class(3sec)**.

Functions: **pkc_add_trusted_key(3sec)**, **pkc_lookup_keys_in_trustlist(3sec)**, **pkc_lookup_key_in_trustlist(3sec)**, **pkc_check_cert_against_trustlist(3sec)**, **pkc_revoke_certificate(3sec)**, **pkc_revoke_certificates(3sec)**, **pkc_delete_trustlist(3sec)**, **pkc_copy_trustlist(3sec)**.

rdacl_get_access

Purpose

Reads a privilege attribute certificate

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_get_access(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_acl_permset_t *net_rights
     error_status_t *status);
```

Parameters

Input

h A handle referring to the object whose ACL is to be accessed.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

Output

net_rights
The output list of access rights, in **sec_acl_permset_t** form. This is a 32-bit set of permission flags supported by the manager type.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_get_access()** routine determines the complete extent of access to the specified object by the calling process. Although the **rdacl_test_access()** routines are the preferred method of testing access, this routine is useful for implementing operations like the conventional UNIX access function.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_manager_type
The manager type is not valid.

sec_acl_invalid_acl_type
The ACL type is not valid.

sec_acl_not_authorized
The requested operation is not allowed.

sec_acl_object_not_found
The requested object could not be found.

error_status_ok
The call was successful.

Related Information

Functions: **rdacl_test_access(3sec)**, **sec_intro(3sec)**.

rdacl_get_manager_types

Purpose

Lists the types of ACLs protecting an object

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_get_manager_types(
     handle_t h
     sec_acl_component_name_t component_name
     sec_acl_type_t sec_acl_type
     unsigned32 size_avail
     unsigned32 *size_used
     unsigned32 *num_types
     uuid_t manager_types[ ]
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

size_avail
An unsigned 32-bit integer containing the allocated length of the *manager_types[ ]* array.

Output

size_used
An unsigned 32-bit integer containing the number of output entries returned in the *manager_types[ ]* array.

num_types
An unsigned 32-bit integer containing the number of types returned in the *manager_types[ ]* array. This is always equal to *size_used*.

manager_types[ ]
An array of length *size_avail* to contain UUIDs (of type **uuid_t**) identifying the different 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 **rdacl_get_manager_types()** routine returns a list of the types of ACLs protecting an object. For example, in addition to the regular file system ACL, a file representing the stable storage of some database could have an ACL manager that supported permissions allowing database updates only on certain days of the week.

ACL editors and browsers can use this operation to determine the ACL manager types that a particular reference monitor is using to protect a selected entity. Then, using the **rdacl_get_printstring()** routine, they can determine how to format for display the permissions supported by a specific manager.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
: The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
: The call was successful.

Related Information

Functions: **rdacl_get_printstring(3sec)**, **sec_intro(3sec)**.

rdacl_get_mgr_types_semantics

Purpose

Lists the ACL manager types protecting an object and the POSIX semantics supported by each manager type

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_get_mgr_types_semantics(
     handle_t h
     sec_acl_component_name_t component_name
     sec_acl_type_t sec_acl_type
     unsigned32 size_avail
     unsigned32 *size_used
     unsigned32 *num_types
     uuid_t manager_types[ ]
     sec_acl_posix_semantics_t posix_semantics[ ]
     error_status_t *status);
```

Parameters

Input

h
A handle referring to the target object.

component_name
A character string containing the name of the target object.

sec_acl_type
The ACL type used to limit the function's output to ACL managers that control the specified types of ACLs. The possible values are as follows:

- **sec_acl_type_object**

  Object ACL, the ACL controlling access to an object.
- **sec_acl_type_default_object**

  Initial Object ACL, the default ACL for objects created in a container object.
- **sec_acl_type_default_container**

  Initial Container ACL, the default ACL for containers created in a container object.

size_avail
An unsigned 32-bit integer containing the allocated length of the *manager_types[ ]* and the *posix_semantics[ ]* arrays.

Output

size_used
An unsigned 32-bit integer containing the number of output entries returned in the *manager_types[ ]* array.

num_types
An unsigned 32-bit integer containing the number of types returned in the *manager_types[ ]* array. This is always equal to *size_used*.

manager_types[ ]
An array of length *size_avail* containing the returned UUIDs (of type **uuid_t**) identifying the different ACL manager types protecting the target object.

posix_semantics[ ]
An array of length *size_avail* containing the POSIX semantics (of type **sec_acl_posix_semantics_t**) that are supported by each returned ACL manager type.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_get_manager_types_semantics()** routine returns a list of the ACL manager types protecting an object and a list of the POSIX semantics supported by those ACL manager types. Access to an object can be controlled by multiple ACL manager types. For example, access to a file representing the stable storage of a database could be controlled by two ACL manager types each with completely different sets of permissions: one to provide standard file system access (read, write, execute, and so on) and one to provide access that allows database updates only on certain days of the week.

ACL editors and browsers can use this operation to determine the ACL manager types that a particular reference monitor is using to protect a selected entity. Then, using the **rdacl_get_printstring()** routine, they can determine how to format for display the permissions supported by a specific manager.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **rdacl_get_printstring(3sec)**, **sec_intro(3sec)**.

rdacl_get_printstring

Purpose

Returns printable ACL strings

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_get_printstring(
     handle_t h
     uuid_t *manager_type
     unsigned32 size_avail
     uuid_t *manager_type_chain
     sec_acl_printstring_t *manager_info
     boolean32 *tokenize
     unsigned32 *total_num_printstrings
     unsigned32 *size_used
     sec_acl_printstring_t printstrings[ ]
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **rdacl_get_manager_types()** to acquire a list of the manager types protecting a given object.

size_avail
An unsigned 32-bit integer containing the allocated length of the *printstrings*[ ] array.

Output

manager_type_chain
If the target object ACL contains more than 32 permission bits, multiple manager types are used, one for each 32-bit wide slice of permissions. The UUID returned in *manager_type_chain* refers to the next ACL manager in the chain. If there are no more ACL managers for this ACL, **uuid_nil** is returned.

manager_info
Provides a name and helpstring for the given ACL manager.

tokenize
When FALSE this variable indicates that the returned permission printstrings are unambiguous and therefore may be concatenated when printed without confusion. When TRUE, however, this property does not hold, and the strings need to be separated when printed or passed.

total_num_printstrings
An unsigned 32-bit integer containing the total number of permission printstrings supported by this ACL manager type.

size_used
An unsigned 32-bit integer containing the number of permission entries returned in the *printstrings*[ ] array.

printstrings[ ]
An array of permission printstrings of type **sec_acl_printstring_t**. Each entry of the array is a structure containing three components:

printstring
A character string of maximum length **sec_acl_printstring_len** containing the printable representation of a specified permission.

helpstring
A character string of maximum length **sec_acl_printstring_help_len** containing some text that can be used to describe the specified permission.

permissions
A **sec_acl_permset_t** permission set describing the permissions that are to be represented with the companion printstring.

The array consists of one such entry for each permission supported by the ACL manager identified by *manager_type*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_get_printstring()** routine returns an array of printable representations (called printstrings) for each permission bit or combination of permission bits the specified ACL manager will support. The ACL manager type specified must be one of the types indicated by the ACL handle.

In addition to returning the printstrings, this routine also returns instructions about how to print the strings. When the *tokenize* variable is set to FALSE, a print string might be **r** or **w**, which could be concatenated in the display as **rw** without any confusion. However, when the *tokenize* variable is TRUE, it implies the printstrings might be of a form like **read** or **write**, which must be displayed separated by spaces or colons or something.

In any list of permission printstrings, there may appear to be some redundancy. ACL managers often define aliases for common permission combinations. By convention, however, simple entries need to appear at the beginning of the *printstrings*[ ] array, and combinations need to appear at the end.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not among those referenced by the input handle.

error_status_ok
The call was successful.

Related Information

Functions: **rdacl_get_manager_types(3sec)**, **sec_acl_bind(3sec)**, **sec_intro(3sec)**.

rdacl_get_referral

Purpose

Gets a referral to an ACL update site

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_get_referral(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_acl_type_t sec_acl_type
     sec_acl_tower_set_t *towers[ ]
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

Output

towers[ ]
A pointer to address information indicating an ACL update site. This information, obtained from the RPC runtime, is used by the client-side code to construct a new ACL binding handle indicating a site that will not return the **sec_acl_site_readonly** error.

The **sec_acl_tower_set_t** structure contains an array of towers (called *towers[ ]*) and an unsigned 32-bit integer indicating the number of array elements (called *count*). This type enables the client to pass in an unallocated array of towers and have the server allocate the correct amount.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_get_referral()** routine obtains a referral to an ACL update site. This function is used when the current ACL site yields a **sec_acl_site_readonly** error. Some replication managers will require all updates for a given object to be directed to a given replica. If clients of the generic ACL interface know they are dealing with an object that is replicated in this way, this function allows them to recover from the problem and rebind to the proper update site. The DCE network registry, for example, is replicated this way.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

rdacl_lookup

Purpose

Returns the ACL for an object

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_lookup(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_acl_type_t sec_acl_type
     sec_acl_result_t *result);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

Output

result A pointer to a tagged union of type **sec_acl_result_t**. The tag is the completion status, **result.st**. If **result.st** is equal to **error_status_ok**, the union contains an ACL. Otherwise, the completion status indicates an error, and the union is empty.

If the call returned successfully, the **result.tagged_union.sec_acl_list_t** structure contains a **sec_acl_list_t**. This data type is an array of pointers to **sec_acl_t**s that define ACLs. If the permission set of the returned ACL is 32 bits or smaller, **sec_acl_list_t** points to only one **sec_acl_t**. If the permission set of the returned ACL is larger than 32 bits, multiple **sec_acl_t**s are used to hold them, and the **sec_acl_list_t** points to multiple **sec_acl_t**s.

Description

The **rdacl_lookup()** routine loads into memory a copy of an object's ACL corresponding to the specified manager type. The routine returns a pointer to the ACL. This routine is only used by ACL editors and browsers; an application would use **sec_acl_test_access()** or **sec_acl_test_access_on_behalf()** to process the contents of an ACL.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
: The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
: The manager type selected is not an available option.

sec_acl_cant_allocate_memory
: The requested operation requires more memory than is available.

error_status_ok
: The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_test_access(3sec)**, **sec_acl_test_access_on_behalf(3sec)**, **sec_intro(3sec)**.

rdacl_replace

Purpose

Replaces an ACL

Synopsis

```
#include <dce/rdaclif.h>

void  rdacl_replace(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_acl_type_t sec_acl_type
     sec_acl_list_t *sec_acl_list
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

sec_acl_list
The new ACL to use for the target object. This is represented by a pointer to the **sec_acl_list_t** structure containing the complete access control list. An ACL contains a list of ACL entries, the UUID of the default cell where authentication takes place (foreign entries in the ACL contain the name of their parent cell), and the UUID of the ACL manager to interpret the list.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_replace()** routine replaces the ACL indicated by the input handle with the information in the *sec_acl_list* parameter. ACLs are thought of as immutable, and in

order to modify them, an editing application must read an entire ACL (using the **sec_acl_lookup()** routine), modify it as needed, and replace it using this routine.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
: The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
: The manager type selected is not an available option.

error_status_ok
: The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_lookup(3sec)**, **sec_intro(3sec)**.

rdacl_test_access

Purpose

Tests access to an object

Synopsis

```
#include <dce/rdaclif.h>

boolean32  rdacl_test_access(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_acl_permset_t desired_permset
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

desired_permset
A permission set in **sec_acl_permset_t** form containing the desired privileges. This is a 32-bit set of permission flags supported by the manager type.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_test_access()** routine determines if the specified ACL contains entries granting privileges to the calling process matching those in *desired_permset*. An application generally only inquires after the minimum set of privileges needed to accomplish a specific task.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

error_status_ok
The call was successful.

Related Information

Functions: **rdacl_test_access_on_behalf(3sec)**, **sec_intro(3sec)**.

rdacl_test_access_on_behalf

Purpose

Tests access to an object on behalf of another process

Synopsis

```
#include <dce/rdaclif.h>

boolean  rdacl_test_access_on_behalf(
     handle_t h
     sec_acl_component_name_t component_name
     uuid_t *manager_type
     sec_id_pac_t *subject
     sec_acl_permset_t desired_permset
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object.

component_name
A character string containing the name of the target object.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

subject
A privilege attribute certificate (PAC) for the subject process. The PAC contains the name and UUID of the principal and parent cell of the subject process, as well as a list of any groups to which it belongs. The PAC also contains a flag (named **authenticated**). When set, it indicates that the certificate was obtained from an authenticated source. When not set, the certificate must not be trusted.

The field is FALSE when it was obtained from the **rpc_auth** layer and the protect level was set to **rpc_c_protect_level_none**. This indicates that no authentication protocol was actually used in the remote procedure call; the identity was simply transmitted from the caller to the callee. If an authentication protocol was used, then the flag is set to TRUE. A server uses **rpc_binding_inq_auth_client()** to acquire a certificate for the client process.

desired_permset
A permission set in **sec_acl_permset_t** form containing the desired privileges. This is a 32-bit set of permission flags supported by the manager type.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rdacl_test_access_on_behalf()** routine determines if the specified ACL contains entries granting privileges to the subject, a process besides the calling process, matching those in *desired_permset*. This routine succeeds only if the access is available to both the caller process as well as the subject identified in the call. An application will generally only inquire after the minimum set of privileges needed to accomplish a specific task.

Notes

This call is not intended to be used by application programs. The *sec_acl* application programming interface (API) provides all the functionality necessary to use the ACL facility. This reference page is provided for programmers who wish to write an ACL manager. In order to write an ACL manager, a programmer must implement the entire **rdacl** interface.

This network interface is called on the client side via the *sec_acl* local interface. Developers are responsible for implementing the server side of this interface. Test server code is included as a sample implementation.

Files

/usr/include/dce/rdaclif.idl
: The **idl** file from which **dce/rdaclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
: The manager type selected is not an available option.

error_status_ok
: The call was successful.

Related Information

Functions: **rdacl_test_access(3sec)**, **rpc_binding_inq_auth_client(3rpc)**, **sec_intro(3sec)**.

rsec_pwd_mgmt_gen_pwd

Purpose

Generates a set of passwords

Synopsis

```
#include <dce/rsec_pwd_mgmt.h>

void  rsec_pwd_mgmt_gen_pwd(
     handle_t pwd_mgmt_svr_h
     sec_rgy_name_t princ_name
     unsigned32 plcy_args
     sec_attr_t plcy[ ]
     sec_bytes_t gen_info_in
     unsigned32 num_pwds
     unsigned32 *num_returned
     sec_passwd_rec_t gen_pwd_set[ ]
     sec_bytes_t *gen_info_out
     error_status_t *stp);
```

Parameters

Input

pwd_mgmt_svr_h
An RPC binding handle to the password management server exporting this operation.

princ_name
The name of the principal requesting the generated passwords.

plcy_args
The size of the *plcy[ ]* array.

plcy[ ] An array of extended registry attributes, each specifying a password management policy of some sort. The contents of this array are as follows:

plcy[0]
Effective registry password minimum length for the principal.

plcy[1]
Effective registry password policy flags for the principal, describing limitations on password characters.

gen_info_in
An NDR pickle containing additional information needed to generate the passwords. There are currently no encoding types defined.

num_pwds
The number of generated passwords requested.

Output

num_returned
The number of generated passwords returned.

gen_pwd_set[ ]
An array of generated passwords, each stored in a **sec_passwd_rec_t** structure.

gen_info_out
An NDR pickle containing additional information returned by the password management server. There are currently no encoding types defined.

stp A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rsec_pwd_mgmt_gen_pwd()** routine returns a set of generated passwords.

Notes

This function is not intended to be called by application programmers. The **sec_pwd_mgmt()** API provides all the functionality necessary to retrieve generated passwords. This reference page is provided for programmers who want to write their own password management servers.

This network interface is called on the client side via the **sec_pwd_mgmt_gen_pwd()** operation. Developers are responsible for implementing the server side of this interface. (**pwd_strengthd(8sec)** is provided as a sample implementation.)

The **plcy** [ ] parameter is intended to be expandable to allow administrators to attach new password policy ERAs to a principal. This feature is, however, currently unsupported, and the **plcy** [ ] parameter consists only of the entries described in this reference page.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The **idl** file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_pwd_mgmt_not_authorized
The user is not authorized to call this API.

sec_pwd_mgmt_svr_error
Password management server generic error. Additional information is usually logged by the password management server.

error_status_ok
The call was successful.

Related Information

Functions: **pwd_strengthd(8sec)**, **rsec_pwd_mgmt_str_chk(3sec)**, **sec_intro(3sec)**, **sec_pwd_mgmt_gen_pwd(3sec)**.

rsec_pwd_mgmt_str_chk

Purpose

Strength-checks a password

Synopsis

```
#include <dce/rsec_pwd_mgmt.h>

boolean32  rsec_pwd_mgmt_str_chk(
     handle_t handle
     sec_rgy_name_t princ
     sec_passwd_rec_t *pwd
     signed32 pwd_val_type
     unsigned32 plcy_args
     sec_attr_t plcy[ ]
     sec_bytes_t str_info_in
     sec_bytes_t *str_info_out
     error_status_t *stp);
```

Parameters

Input

handle An RPC binding handle to the password management server exporting this operation.

princ The name of the principal requesting the generated passwords.

pwd A pointer to the password to be strength checked.

pwd_val_type
The value of the user's password validation type (as stored in the *pwd_val_type* ERA).

plcy_args
The size of the *plcy[ ]* array.

plcy[ ] An array of extended registry attributes, each specifying a password management policy of some sort. The contents of this array are as follows:

plcy[0]
Effective registry password minimum length for the principal.

plcy[1]
Effective registry password policy flags for the principal, describing limitations on password characters.

str_info_in
An NDR pickle containing additional information needed to strength check the password. There are currently no encoding types defined.

Output

str_info_out
An NDR pickle containing additional information returned by the password management server. There are currently no encoding types defined.

stp A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **rsec_pwd_mgmt_str_chk()** routine strength checks a password.

Notes

This function is not intended to be called by application programmers. The registry server provides all the functionality necessary to strength check passwords. This reference page is provided for programmers who wish to write their own password management servers.

This network interface is called on the client side via **secd(8)**. Developers are responsible for implementing the server side of this interface. (**pwd_strengthd(8sec)** is provided as a sample implementation.)

The **plcy** [ ] parameter is intended to be expandable to allow administrators to attach new password policy ERAs to a principal. This feature is, however, currently unsupported, and the **plcy** [ ] parameter consists only of the entries described in this reference page.

Return Value

The **rsec_pwd_mgmt_str_chk()** routine returns TRUE if the user's password passes the server's strength checking algorithm and FALSE if it does not.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The **idl** file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_pwd_mgmt_str_check_failed
The password failed the server's strength checking algorithm.

sec_pwd_mgmt_not_authorized
The user is not authorized to call this API.

sec_pwd_mgmt_svr_error
Password management server generic error. Additional information is usually logged by the password management server.

error_status_ok
The call was successful

Related Information

Functions: **pwd_strengthd(8sec)**, **rsec_pwd_mgmt_gen_pwd(3sec)**, **sec_intro(3sec)**.

sec_acl_bind

Purpose

Returns a handle for an object's ACL

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_bind(
     unsigned char *entry_name
     boolean32 bind_to_entry
     sec_acl_handle_t *h
     error_status_t *status);
```

Parameters

Input

entry_name
: The name of the target object. Subsequent ACL operations using the returned handle will affect the ACL of this object.

bind_to_entry
: Bind indicator, for use when *entry_name* identifies both an entry in the global namespace and an actual object. A TRUE value binds the handle to the entry in the namespace, while FALSE binds the handle to the actual object.

Output

h
: A pointer to the **sec_acl_handle_t** variable to receive the returned ACL handle. The other *sec_acl* routines use this handle to refer to the ACL for the object specified with *entry_name*.

status
: A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_bind()** routine returns a handle bound to the indicated object's ACL. This routine is central to all the other *sec_acl* routines, each of which requires this handle to identify the ACL on which to operate.

Notes

If the specified name is both an actual object, and an entry in the global namespace, there are two ACLs associated with it. For example, in addition to the ACL normally attached to file system objects, the root directory of a file system has an ACL corresponding to its entry in the global namespace. This controls access by outsiders to the entire file system, whereas the resident ACL for the root directory only controls access to the directory and, by inheritance, its subdirectories. The ambiguity must be resolved with the *bind_to_entry* parameter.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_object_not_found
The requested object could not be found.

sec_acl_no_acl_found
There is no ACL associated with the specified object.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_acl_bind_auth

Purpose

Returns an opaque handle to an object's ACL

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_bind(
     unsigned char *entry_name
     boolean32 bind_to_entry
     sec_acl_bind_auth_info_t *auth_info
     sec_acl_handle_t *h
     error_status_t *status);
```

Parameters

Input

entry_name
The name of the target object. Subsequent access control list (ACL) operations using the returned handle will affect the ACL of this object.

bind_to_entry
A bind indicator, for use when *entry_name* identifies both an entry in the global namespace and an actual object. A TRUE value binds the handle to the entry in the namespace, while FALSE binds the handle to the actual object.

auth_info
A pointer to the **sec_acl_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info(3rpc)** reference page for more information on authorization.) If this argument is not supplied, default authorization information is used as it is in the **sec_acl_bind()** routine.

Output

h A pointer to the **sec_acl_handle_t** variable to receive the returned ACL handle. The other *sec_acl* routines use this handle to refer to the ACL for the object specified with *entry_name*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_bind_auth()** routine returns a handle bound to the indicated object's ACL. This routine and the **sec_acl_bind()** routine provide the handle that identifies the ACL on which other *sec_acl* routines operate. Use this routine instead of the **sec_acl_bind()** routine to specify authorization information explicitly instead of using the default authorization information.

Note:

If the specified name is both an actual object, and an entry in the global namespace, there are two ACLs associated with it. For example, in addition to the ACL normally attached to file system objects, the root directory of a file system has an ACL corresponding to its entry in the global namespace. This controls access by outsiders to the entire file system, whereas the resident ACL for the root directory only controls access to the directory and, by inheritance, its subdirectories. The ambiguity must be resolved with the *bind_to_entry* parameter.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_object_not_found
The requested object could not be found.

sec_acl_no_acl_found
There is no ACL associated with the specified object.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_acl_bind(3sec)**.

sec_acl_bind_to_addr

Purpose

Returns a handle to an object identified by its network address

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_bind_to_addr(
     unsigned char *site_addr
     sec_acl_component_name_t component_name
     sec_acl_handle_t *h
     error_status_t *status);
```

Parameters

Input

site_addr
An RPC string binding to the fully qualified network address of the target object.

component_name
The name of the target object. Subsequent ACL operations using the returned handle will affect the ACL of this object.

Output

h A pointer to the **sec_acl_handle_t** variable to receive the returned ACL handle. The other *sec_acl* routines use this handle to refer to the ACL for the object specified with *entry_name*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_bind_to_addr()** routine returns a handle bound to the indicated object's ACL manager. This routine and the **sec_acl_bind()** routine are central to all the other *sec_acl* routines, each of which requires a handle to identify the ACL on which to operate.

This routine differs from **sec_acl_bind()** in that it binds to the network address of the target object, rather than to a cell namespace entry. Therefore, unlike **sec_acl_bind()**, it is possible to pass **sec_acl_bind_to_addr()** a null string as a component name and to bind with a nonexistent name. The purpose of this call is to eliminate the necessity of looking up an object's name. To validate the name, use **sec_acl_bind()**.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_object_not_found
The requested object could not be found.

sec_acl_no_acl_found
There is no ACL associated with the specified object.

sec_acl_unable_to_authenticate
The call could not authenticate to the server that manages the target object's ACL.

sec_acl_bind_error
The call could not bind to the requested site.

sec_acl_invalid_site_name
The *site_addr* parameter is invalid.

sec_acl_cant_allocate_memory
Memory allocation failure.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_acl_calc_mask

Purpose

Returns the sec_acl_type_mask_obj entry for the specified ACL list

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_calc_mask(
     sec_acl_list_t *sec_acl_list
     error_status_t *status);
```

Parameters

Input/Output

sec_acl_list
A pointer to a **sec_acl_type_t** the specifies the number of ACLs of each ACL type. The **sec_acl_type_t** data type distinguishes between the various types of ACLs an object can possess for a given manager. In the file system, for example, most objects have only one ACL, controlling the access to that object, but objects that control the creation of other objects (sometimes referred to as *containers*) may have more. A directory, for example, can have ACLs to be used as initial values when member objects are created.

Do not confuse ACL types with the permissions corresponding to different ACL manager types or with the ACL manager types themselves.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_calc_mask()** routine calculates and sets the **sec_acl_e_type_mask_obj** entry of the specified ACL list. The value of the **sec_acl_e_type_mask_obj** entry is the union of the permissions of all ACL entries that refer to members of the file group class.

This operation is performed locally, within the client. The function does not check to determine if the manager to which the specified ACL list will be submitted supports the **sec_acl_e_type_mask_obj** entry type. The calling application must determine whether to call this routine, after obtaining the required, if any, POSIX semantics, via the **sec_acl_get_mgr_types_semantics()** routine.

Notes

This call is provided in source code form.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_cant_allocate_memory
Requested operation requires more memory than is available.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_acl_get_access

Purpose

Lists the access (permission set) that the caller has for an object

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_get_access(
     sec_acl_handle_t h
     uuid_t *manager_type
     sec_acl_permset_t *net_rights
     error_status_t *status);
```

Parameters

Input

h A handle referring to the object whose ACL is to be accessed. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the manager type of the ACL in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

Output

net_rights
The output list of access rights in **sec_acl_permset_t** form. This is a 32-bit set of permission flags supported by the manager type.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_get_access()** routine determines the complete extent of access to the specified object by the calling process. Although the **sec_acl_test_access()** and **sec_acl_test_access_on_behalf()** routines are the preferred method of testing access, this routine is useful for implementing operations like the conventional UNIX access function.

Permissions Required

The **sec_acl_get_access()** routine requires at least one permission of any kind on the object for which the access is to be returned.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_test_access(3sec)**, **sec_acl_test_access_on_behalf(3sec)**.

sec_acl_get_error_info

Purpose

Returns error information from an ACL handle

Synopsis

```
#include <dce/daclif.h>

error_status_t  sec_acl_get_error_info(
     sec_acl_handle_t h);
```

Parameters

Input

h A handle referring to the target ACL. The handle is bound to the ACL with the **sec_acl_bind()** routine, which also specifies the name of the object to which the target ACL belongs.

Description

The **sec_acl_get_error_info()** routine returns error information from the specified ACL handle.

During a call to a routine in the *sec_acl* application programming interface (API), error codes received from the RPC runtime or other APIs are saved in the ACL handle and a corresponding error code from the *sec_acl* set is passed back by the ACL API. The **sec_acl_get_error_info()** routine returns the last error code stored in the ACL handle for those clients who need to know exactly what went wrong.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Return Values

This routine returns a value of type **error_status_t**, indicating the cause of the last error issued by the RPC runtime.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_invalid_handle
The ACL handle specified by **sec_acl_handle_t** is invalid.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_lookup(3sec)**, **sec_intro(3sec)**.

sec_acl_get_manager_types

Purpose

Lists the manager types of the ACLs protecting an object

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_get_manager_types(
     sec_acl_handle_t h
     sec_acl_type_t sec_acl_type
     unsigned32 size_avail
     unsigned32 *size_used
     unsigned32 *num_types
     uuid_t manager_types[ ]
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

size_avail
An unsigned 32-bit integer containing the allocated length of the *manager_types[ ]* array.

Output

size_used
An unsigned 32-bit integer containing the number of output entries returned in the *manager_types[ ]* array.

num_types
An unsigned 32-bit integer containing the number of types returned in the *manager_types[ ]* array. This may be greater than *size_used* if there was not enough space allocated in the *manager_types[ ]* array for all the manager types.

manager_types[ ]
An array of length *size_avail* to contain UUIDs (of type **uuid_t**) identifying the different 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 **sec_acl_get_manager_types()** routine returns a list of the manager types of ACLs of type *sec_acl_type* that are protecting the object identified by *h*. For example, in addition to the regular file system ACL, a file representing the stable storage of some database could have an ACL manager that supported permissions allowing database updates only on certain days of the week.

ACL editors and browsers can use this operation to determine the ACL manager types that a particular reference monitor is using to protect a selected entity. Then, using the **sec_acl_get_printstring()** routine, they can determine how to format for display the permissions supported by a specific manager.

Permissions Required

The **sec_acl_get_manager_types()** routine requires at least one permission of any kind on the object for which the ACL manager types are to be returned.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_get_printstring(3sec)**, **sec_intro(3sec)**.

sec_acl_get_mgr_types_semantics

Purpose

Lists the manager types of the ACLs protecting an object

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_get_mgr_types_semantics(
     sec_acl_handle_t h
     sec_acl_type_t sec_acl_type
     unsigned32 size_avail
     unsigned32 *size_used
     unsigned32 *num_types
     uuid_t manager_types[ ]
     sec_acl_posix_semantics_t posix_semantics[ ]
     error_status_t *status);
```

Parameters

Input

h
A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

size_avail
An unsigned 32-bit integer containing the allocated length of the *manager_types[ ]* array.

Output

size_used
An unsigned 32-bit integer containing the number of output entries returned in the *manager_types[ ]* array.

num_types
An unsigned 32-bit integer containing the number of types returned in the *manager_types[ ]* array. This may be greater than *size_used* if there was not enough space allocated in the *manager_types[ ]* array for all the manager types.

manager_types[ ]
An array of length *size_avail* to contain UUIDs (of type **uuid_t**) identifying the different types of ACL managers protecting the target object.

posix_semantics[ ]
An array of POSIX semantics supported by each manager type with entries of type **sec_acl_posix_semantics_t**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_get_mgr_types_semantics()** routine returns a list of the manager types of ACLs of type *sec_acl_type* that are protecting the object identified by *h*. For example, in addition to the regular file system ACL, a file representing the stable storage of some database could have an ACL manager that supported permissions allowing database updates only on certain days of the week.

ACL editors and browsers can use this operation to determine the ACL manager types that a particular reference monitor is using to protect a selected entity. Then, using the **sec_acl_get_printstring()** routine, they can determine how to format for display the permissions supported by a specific manager.

Permissions Required

The **sec_acl_get_mgr_types_semantics()** routine requires at least one permission of any kind on the object for which the ACL manager types are to be returned.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_get_printstring(3sec)**, **sec_intro(3sec)**.

sec_acl_get_printstring

Purpose

Returns printable ACL strings

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_get_printstring(
     sec_acl_handle_t h
     uuid_t *manager_type
     unsigned32 size_avail
     uuid_t *manager_type_chain
     sec_acl_printstring_t *manager_info
     boolean32 *tokenize
     unsigned32 *total_num_printstrings
     unsigned32 *size_used
     sec_acl_printstring_t printstrings[ ]
     error_status_t *status);
```

Parameters

Input

h
A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

size_avail
An unsigned 32-bit integer containing the allocated length of the *printstrings[ ]* array.

Output

manager_type_chain
If the target object ACL contains more than 32 permission bits, multiple manager types are used, one for each 32-bit wide ″slice″ of permissions. The UUID returned in *manager_type_chain* refers to the next ACL manager in the chain. If there are no more ACL managers for this ACL, **uuid_nil** is returned.

manager_info
Provides a name and help string for the given ACL manager.

tokenize
When FALSE, this variable indicates that the returned permission printstrings are unambiguous and therefore may be concatenated when printed without confusion. When TRUE, however, this property does not hold, and the strings need to be separated when printed or passed.

total_num_printstrings
An unsigned 32-bit integer containing the total number of permission printstrings supported by this ACL manager type.

size_used
An unsigned 32-bit integer containing the number of permission entries returned in the *printstrings[ ]* array.

printstrings[ ]
An array of permission printstrings of type **sec_acl_printstring_t**. Each entry of the array is a structure containing the following three components:

printstring
A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.

helpstring
A character string of maximum length **sec_acl_printstring_help_len** containing some text that can be used to describe the specified permission.

permissions
A **sec_acl_permset_t** permission set describing the permissions that are represented with the companion printstring.

The array consists of one such entry for each permission supported by the ACL manager identified by *manager_type*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_get_printstring()** routine returns an array of printable representations (called *printstrings*) for each permission bit or combination of permission bits the specified ACL manager supports. The ACL manager type specified must be one of the types protecting the object indicated by *h*.

In addition to returning the printstrings, this routine also returns instructions about how to print the strings. When the *tokenize* variable is set to FALSE, a printstring might be **r** or **w**, which could be concatenated in the display as **rw** without any confusion. However, when the *tokenize* variable is TRUE, it implies the printstrings might be of a form like **read** or **write**, which must be displayed separated by spaces or colons or something.

In any list of permission printstrings, there may appear to be some redundancy. ACL managers often define aliases for common permission combinations. By convention, however, simple entries should appear at the beginning of the *printstrings[ ]* array, and combinations should appear at the end.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not among those referenced by the input handle.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_get_manager_types(3sec)**, **sec_intro(3sec)**.

sec_acl_lookup

Purpose

Returns the ACL for an object

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_lookup(
     sec_acl_handle_t h
     uuid_t *manager_type
     sec_acl_type_t sec_acl_type
     sec_acl_list_t *sec_acl_list
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

Output

sec_acl_list
A pointer to the **sec_acl_list_t** structure to receive the complete access control list. An ACL contains a list of ACL entries, the UUID of the default cell where authentication takes place (foreign entries in the ACL contain the name of their home cell), and the UUID of the ACL manager to interpret the list.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_lookup()** routine loads into memory a copy of an object's ACL corresponding to the specified manager type. The routine returns a pointer to the

ACL. This routine is only used by ACL editors and browsers; an application would use **sec_acl_test_access()** or **sec_acl_test_access_on_behalf()** to process the contents of an ACL.

Permissions Required

The **sec_acl_lookup()** routine requires at least one permission of any kind on the object for which the ACL is to be returned.

Notes

The memory containing the **sec_acl_t** structure for each ACL is dynamically allocated. Use the **sec_acl_release()** routine to return each ACL's memory block to the pool when an application is finished with the ACLs.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

sec_acl_cant_allocate_memory
The requested operation requires more memory than is available.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_test_access(3sec)**, **sec_acl_test_access_on_behalf(3sec)**, **sec_intro(3sec)**.

sec_acl_release

Purpose

Releases ACL storage

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_release(
     sec_acl_handle_t h
     sec_acl_t *sec_acl
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

sec_acl
A pointer to the complete ACL associated with the target object.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_release()** routine releases any local storage associated with the ACL object, returning it to the pool. This is strictly a local operation (since the storage in question is local), and has no effect on the remote object or its ACL. The ACL handle is in the argument list only for consistency with other *sec_acl* routines.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_lookup(3sec)**, **sec_intro(3sec)**.

sec_acl_release_handle

Purpose

Removes an ACL handle

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_release_handle(
     sec_acl_handle_t *h
     error_status_t *status);
```

Parameters

Input

h The handle to be removed. The handle is bound to the object to which the ACL belongs with the **sec_acl_bind()** routine.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_release_handle()** routine removes the specified handle. This is strictly a local operation, and has no effect on the remote object or its ACL.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_intro(3sec)**.

sec_acl_replace

Purpose

Replaces an ACL

Synopsis

```
#include <dce/daclif.h>

void  sec_acl_replace(
     sec_acl_handle_t h
     uuid_t *manager_type
     sec_acl_type_t sec_acl_type
     sec_acl_list_t *sec_acl_list
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

sec_acl_type
The ACL type. The **sec_acl_type_t** data type distinguishes the various types of ACLs an object can possess for a given manager type. The possible values are as follows:

- **sec_acl_type_object**
- **sec_acl_type_default_object**
- **sec_acl_type_default_container**

sec_acl_list
The new ACL to use for the target object. This is represented by a pointer to the **sec_acl_list_t** structure containing the complete access control list. An ACL contains a list of ACL entries, the UUID of the default cell where authentication will take place (foreign entries in the ACL contain the name of their parent cell), and the UUID of the ACL manager to interpret the list.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_replace()** routine replaces the ACL indicated by the input handle with the information in the *sec_acl_list* parameter. ACLs are thought of as immutable,

and in order to modify them, an editing application must read an entire ACL (using the **sec_acl_lookup()** routine), modify it as needed, and replace it using this routine.

Permissions Required

The **sec_acl_replace()** routine requires the **c** (control) permission on the object for which the ACL is to be replaced.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_lookup(3sec)**, **sec_intro(3sec)**.

sec_acl_test_access

Purpose

Tests access to an object

Synopsis

```
#include <dce/daclif.h>

boolean32  sec_acl_test_access(
     sec_acl_handle_t h
     uuid_t *manager_type
     sec_acl_permset_t desired_permset
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

desired_permset
A permission set in **sec_acl_permset_t** form containing the desired privileges. This is a 32-bit set of permission flags supported by the manager type.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_test_access()** routine determines if the specified ACL contains entries granting privileges to the calling process matching those in *desired_permset*. An application generally only inquires after the minimum set of privileges needed to accomplish a specific task.

Permissions Required

The **sec_acl_test_access()** routine requires at least one permission of any kind on the object for which the privileges are to be tested.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Return Values

The routine returns TRUE if the calling application program is authorized to access the target object with the privileges in *desired_permset*.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

error_status_ok
The call was successful.

Related Information

Functions: **sec_acl_bind(3sec)**, **sec_acl_test_access_on_behalf(3sec)**, **sec_intro(3sec)**.

sec_acl_test_access_on_behalf

Purpose

Tests access to an object on behalf of another process

Synopsis

```
#include <dce/daclif.h>

boolean32  sec_acl_test_access_on_behalf(
     sec_acl_handle_t h
     uuid_t *manager_type
     sec_id_pac_t *subject
     sec_acl_permset_t desired_permset
     error_status_t *status);
```

Parameters

Input

h A handle referring to the target object. Use **sec_acl_bind()** to create this handle.

manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_acl_get_manager_types()** to acquire a list of the manager types protecting a given object.

subject
A privilege attribute certificate (PAC) for the subject process. The PAC contains the name and UUID of the principal and cell of the subject process, as well as a list of any groups to which it belongs. The PAC also contains a flag (named **authenticated**). When set, it indicates that the certificate was obtained from an authenticated source. When not set, the certificate must not be trusted. (The field is FALSE when it was obtained from the **rpc_auth(3rpc)** layer and the protect level was set to **rpc_c_protect_level_none**. This indicates that no authentication protocol was actually used in the remote procedure call; the identity was simply transmitted from the caller to the callee. If an authentication protocol was used, then the flag is set to TRUE.)

If a null PAC is passed, the subject is treated as an anonymous user, matching only the **any_other** and **unauthenticated** entries (if they exist) on the ACL.

A server uses **rpc_binding_inq_auth_client()** to acquire a certificate for the client process.

desired_permset
A permission set in **sec_acl_permset_t** form containing the desired privileges. This is a 32-bit set of permission flags supported by the manager type.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_acl_test_access_on_behalf()** routine determines if the specified ACL contains entries that grant the privileges specified in *desired_permset* to the subject process. An application generally inquires about only the minimum set of privileges needed to accomplish a specific task.

Permissions Required

The **sec_acl_test_access_on_behalf()** routine requires at least one permission of any kind on the object for which the privileges are to be tested. Both the calling process and the identified subject must have permission on the object.

Note: This operation is obsolete, but is documented for backward compatibility. **sec_id_pac_t** is no longer the data structure used for identities (for further information, see the **sec_cred_*(3sec)** routines), and delegation subsumes the functionality that the **sec_acl_test_access_on_behalf()** routine was originally intended to provide. ACL managers do not have to implement the server side of this functionality to be DCE compliant, and therefore clients should not rely on its being available in servers.

Files

/usr/include/dce/daclif.idl
The **idl** file from which **dce/daclif.h** was derived.

Return Values

If the routine completes successfully (with a completion status of **error_status_ok**) it returns a value of

- TRUE, if the caller has any access (at least one permission of any kind), and the subject has the *desired_permset* privileges.
- FALSE, if both the caller and the subject have any access, but the subject does not have the *desired_permset* privileges.

If the routine does not complete successfully, it returns a bad completion status code and a return value of FALSE.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_acl_unknown_manager_type
The manager type selected is not an available option.

error_status_ok
The call was successful.

sec_acl_not_implemented
Requested operation is not implemented in this version of DCE.

Related Information

Functions: **rpc_binding_inq_auth_client(3rpc)**, **sec_acl_bind(3sec)**, **sec_acl_test_access(3sec)**, **sec_intro(3sec)**.

sec_attr_trig_query

Purpose

Reads attributes coded with an attribute trigger type of query

Synopsis

```
#include <dce/sec_attr_trig.h>

void sec_attr_trig_query  (
     handle_t h
     sec_attr_component_name_t cell_name
     sec_attr_component_name_t component_name
     sec_attr_trig_cursor_t *cursor
     unsigned32 num_attr_keys
     unsigned32 space_avail
     sec_attr_t attr_keys[ ]
     unsigned32 *num_returned
     sec_attr_t attrs[ ]
     sec_attr_trig_timeval_sec_t time_to_live[ ]
     unsigned32 *num_left
     error_status_t *status);
```

Parameters

Input

h
A handle referring to the trigger server to be accessed. Use the trigger binding information specified in the attribute encoding to acquire a bound handle.

cell_name
A value of **sec_attr_component_name_t** that identifies the cell in which the object whose attribute is to be accessed resides. Supply a NULL *cell_name* to specify the local cell (***/.:***).

component_name
A value of **sec_attr_component_name_t** that identifies the name of the object whose attribute is to be accessed. If *cell_name* specifies a foreign cell, *component_name* is interpreted as a UUID in string format since the caller of this interface knows only the UUID, not the name, of the foreign principal.

num_attr_keys
An unsigned 32-bit integer that specifies the number of elements in the *attr_keys[ ]* array. This integer must be greater than 0 (zero).

space_avail
An unsigned 32-bit integer that specifies the size of the *attr_keys[ ]* array.

attr_keys[ ]
An array of values of type **sec_attr_t**. For each attribute instance, the **sec_attr_t** array contains an *attr_id* (a UUID of type **uuid_t**) to identify the attribute to be queried and an *attr_value*. *attr_value* can be used to pass in optional information required by the attribute trigger query. If no additional information is to be passed, set *attr_value* to **sec_attr_enc_void**. This is actually accomplished by setting the **sec_attr_encoding_t** data type to **sec_attr_enc_void**.

The size of the *attr_keys[ ]* array is determined by *num_attr_keys*.

Input/Output

cursor A pointer to a cursor of type **sec_attr_trig_cursor_t**. As an input parameter, *cursor* can be initialized (by the server) or uninitialized. If the cursor is uninitialized, the cursor begins processing the query at the first attribute that satisfies the search criteria. As an output parameter, *cursor* is positioned past the attributes returned in this call.

Output

num_returned
A pointer to an unsigned 32-bit integer that specifies the number of attribute instances returned in the *attr_keys[ ]* array.

attrs[ ]
An array of values of type **sec_attr_t**. The size of this array is determined by the *space_avail* parameter and the length by the *num_returned* parameter.

*time_to_live***[ ]**
An array of values of type **sec_attr_trig_timeval_sec_t**. For each attribute in the *attrs*[ ] array, The *time_to_live[ ]* array specifies the time in seconds that the attribute can be safely cached.

num_left
A pointer to an unsigned 32-bit integer that supplies the number of attributes found but not returned because of space constraints in the *attrs*[ ] buffer.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_attr_trig_query()** routine reads attributes coded with a attribute trigger type of query.

The **sec_attr_trig_query()** routine is called by the DCE attribute lookup code for all schema entries that specify a query attribute trigger (**sec_attr_trig_type_query** specified with the **sec_attr_trig_type_flags_t** data type). The attribute query code passes the **sec_attr_trig_query()** input parameters to a user-written query attribute trigger server and receives the output parameters back from the server. Although generally this routine is not called directly, this reference page is provided for users who are writing the attribute trigger servers that will receive **sec_attr_trig_query()** input and supply its output.

Multivalued attributes are returned as independent attribute instances sharing the same attribute UUID. A read of an attribute set returns all instances of members of the set; the attribute set instance is not returned.

For objects in the local cell, set the *cell_name* parameter to **null**, and the *component_name* parameter to specify the object's name.

For objects in a foreign cell, set the *cell_name* parameter to identify the name of the foreign cell, and the *component_name* parameter to the UUID in string format that identifies the object in the foreign cell.

The *num_left* parameter contains the number of attributes that were found but could not be returned because of space constraints of the *attrs[ ]* array. (Note that this number may be inaccurate if the target server allows updates between successive queries.) To obtain all of the remaining attributes, set the size of the *attrs[ ]* array so that it is large enough to hold the number of attributes listed in *num_left*.

Files

/usr/include/dce/sec_attr_trig.idl
The **idl** file from which **dce/sec_attr_trig.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

not_all_available

unauthorized

error_status_ok

Related Information

Functions: **sec_attr_trig_cursor_init** , **sec_attr_trig_update(3sec)**, **sec_intro(3sec)**.

sec_attr_trig_update

Purpose

For attributes coded with an attribute trigger type of update, passes attribute updates to an update attribute trigger server for evaluation

Synopsis

```
#include <dce/sec_attr_trig.h>

void sec_attr_trig_update  (
     handle_t h
     sec_attr_component_name_t cell_name
     sec_attr_component_name_t component_name
     unsigned32 num_to_write
     unsigned32 space_avail
     sec_attr_t in_attrs[ ]
     unsigned32 *num_returned
     sec_attr_t out_attrs[ ]
     unsigned32 *num_left
     signed32 *failure_index
     error_status_t *status);
```

Parameters

Input

h
A handle referring to the trigger server to be accessed. Use the trigger binding information specified in the attribute encoding to acquire a bound handle.

cell_name
A value of **sec_attr_component_name_t** that identifies the cell in which the object whose attribute is to be accessed resides. Supply a NULL *cell_name* to specify the local cell (**/.:**).

component_name
A value of **sec_attr_component_name_t** that identifies the name of the object whose attribute is to be accessed. If *cell_name* specifies a foreign cell, *component_name* is interpreted as a UUID in string format since the caller of this interface knows only the UUID, not the name, of the foreign principal.

num_to_write
An unsigned 32-bit integer that specifies the number of elements in the *in_attrs* array. This integer must be greater than 0 (zero).

space_avail
An unsigned 32-bit integer that specifies the size of the *out_attrs* array.

in_attrs[ ]
An array of values of type **sec_attr_t** that specifies the attribute instances to be written. The size of *in_attrs[ ]* is determined by *num_to_write*.

Output

num_returned
A pointer to an unsigned 32-bit integer that specifies the number of attribute instances returned in the *out_attrs[ ]* array.

out_attrs[ ]
An array of values of type **sec_attr_t**. These values, supplied by the update attribute trigger server, are in a form suitable for storage in the registry database.

num_left
A pointer to an unsigned 32-bit integer that supplies the number of attributes that were found but not returned because of space constraints in the *out_attrs[ ]* buffer.

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *in_attrs[ ]* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is −1.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_attr_trig_update()** routine passes attributes coded with an attribute trigger type of update to a user-written update attribute trigger server for evaluation before the updates are made to the registry.

Although generally this routine it is not called directly, this reference page is provided for users who are writing the attribute trigger servers that will receive **sec_attr_trig_update()** input and supply its output.

The **sec_attr_trig_update()** routine is called by the DCE attribute update code for all schema entries that specify an update attribute trigger (**sec_attr_trig_type_update** specified with the **sec_attr_trig_type_flags_t** data type). The attribute update code passes the **sec_attr_trig_update()** input parameters to a user-written update attribute trigger server and receives the output parameters back from the server. The attribute trigger server is responsible for evaluating the semantics of the entry in order to reject or accept it, and the attribute trigger server may even make changes in the output it sends back to the update code to ensure the entry adheres to the semantics. The output received from the attribute trigger server is in a form to be stored in the registry. (Note that update attribute trigger servers do not store attribute values. Attribute values are stored in the registry database.)

This is an atomic operation: if the update of any attribute in the array fails to pass the evaluation, all updates are aborted. The attribute causing the update to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains −1.

For objects in the local cell, set the *cell_name* parameter to **null**, and the *component_name* parameter to specify the object's name.

For objects in a foreign cell, set the *cell_name* parameter the the name of the foreign cells, and the *component_name* parameter to specify the UUID in string format that identifies the object in the foreign cell.

Files

/usr/include/dce/sec_attr_trig.idl
The **idl** file from which **dce/sec_attr_trig.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

database read only

server unavailable

invalid/unsupported attribute type

invalid encoding type

value not unique

site read only

unauthorized

error_status_ok

Related Information

Functions: **sec_attr_trig_query(3sec)**, **sec_intro(3sec)**.

sec_attr_util_alloc_copy

Purpose

Allocates the necessary subfields of the destination **sec_attr_t** and copies the corresponding data from the source **sec_attr_t**

Synopsis

```
#include <dce/sec_attr_util.h>

void sec_attr_util_alloc_copy  (
     void *(*allocate) (unsigned32 size)
     sec_attr_t *from
     sec_attr_t *to
     error_status_t *status);
```

Parameters

Input

(allocate)* **(unsigned32** *size***)**
A caller-specified allocate routine (such as **rpc_ss_allocate()**) used to allocate resources for the output *to* parameter. Set to NULL to use the default **malloc()** routine.

***from** A pointer to a **sec_attr_t** that is the source to be copied from.

Output

***to** A pointer to the target **sec_attr_t** that contains subfields allocated, if necessary, by the caller-specified allocate routine and data copied from the source **sec_attr_t** specified by *from*.

***status**
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_attr_util_alloc_copy()** routine allocates memory for the subfields of the target **sec_attr_t**, if necessary, and copies data from the source **sec_attr_t** to the target **sec_attr_t**.

Use the **sec_attr_util_free()** routine to free the memory allocated by this routine. If a nonnull allocate routine was input to **sec_attr_util_alloc_copy()**, then a corresponding free routine must be input to the **sec_attr_util_free()** routine.

Files

/usr/include/dce/sec_attr_util.idl
The **idl** file from which **dce/sec_attr_util.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_implemented

error_status_ok

Related Information

Functions: **sec_attr_util_free(3sec)**, **sec_attr_util_inst_free_ptrs(3sec)**, **sec_attr_util_inst_free(3sec)**.

sec_attr_util_free

Purpose

Frees nonnull pointers in a **sec_attr_t** with an input deallocate routine

Synopsis

```
#include <dce/sec_attr_util.h>

void   sec_attr_util_free(
     void (*deallocate) (void *ptr)
     sec_attr_t *attr);
```

Parameters

Input/Output

(**deallocate***)(void** **ptr***)**
A caller-specified memory deallocate routine. If set to NULL, the default **free()** is used.

***attr** As input, a pointer to a **sec_attr_t** for which memory should be deallocated. As output, a pointer to the **sec_attr_t** with subfields, if any, deallocated and set to NULL.

Description

The **sec_attr_util_free()** routine uses the input *deallocate* routine to free memory allocated to a **sec_attr_t** by **sec_attr_util_alloc_copy()**. With an input value of NULL for *deallocate*, the **sec_attr_util_free** routine behaves identically to **sec_attr_util_inst_free_ptrs**.

Files

/usr/include/dce/sec_attr_util.idl
The **idl** file from which **dce/sec_attr_util.h** was derived.

Related Information

Functions: **sec_attr_util_alloc_copy(3sec)**, **sec_attr_util_inst_free_ptrs(3sec)**, **sec_attr_util_inst_free(3sec)**.

sec_attr_util_inst_free

Purpose

Frees nonnull pointers in a **sec_attr_t** and the pointer to the **sec_attr_t** itself

Synopsis

```
#include <dce/sec_attr_util.h>

void sec_attr_util_inst_free  (
     sec_attr_t **sec_attr_p);
```

Parameters

Input/Output

***sec_attr_p*
: As input, the address of an allocated pointer to a potentially initialized **sec_attr_t**. As output, the address of a deallocated pointer that has been set to NULL.

Description

The **sec_attr_util_inst_free()** routine frees each nonnull pointer in a **sec_attr_t** pointed to by **sec_attr_p*. The **sec_attr_p* itself is also freed and set to NULL. A partially initialized **sec_attr_t** is handled correctly .

The **sec_attr_util_inst_free()** routine is useful for freeing the resources of dynamically allocated **sec_attr_t**s and their subfields.

Note that most DCE client application programming interfaces (APIs) that return **sec_attr_t**s allocate only subfields, and not the **sec_attr_t** itself. Use **sec_attr_util_inst_free_ptrs** instead of **sec_attr_util_inst_free** to free attribute resources allocated by such APIs.

Files

/usr/include/dce/sec_attr_util.idl
: The **idl** file from which **dce/sec_attr_util.h** was derived.

Related Information

Functions: **sec_attr_util_inst_free_ptrs(3sec)**.

sec_attr_util_inst_free_ptrs

Purpose

Frees nonnull pointers in a **sec_attr_t**

Synopsis

```
#include <dce/sec_attr_util.h>

void sec_attr_util_inst_free_ptrs  (
     sec_attr_t *sec_attr_p);
```

Parameters

Input/Output

sec_attr_p
: As input, a pointer to an allocated and potentially initialized **sec_attr_t**. As output, a pointer to a **sec_attr_t** with internal pointers freed and set to NULL. The **sec_attr_t** itself is not freed.

Description

The **sec_attr_util_inst_free_ptrs()** routine frees and sets to NULL each nonnull pointer in a **sec_attr_t** pointed to by *sec_attr_p*. The **sec_attr_t** itself is not freed. The **sec_attr_t** may have been only partially initialized.

Files

/usr/include/dce/sec_attr_util.idl
: The **idl** file from which **dce/sec_attr_util.h** was derived.

Related Information

Functions: **sec_attr_util_inst_free(3sec)**.

sec_attr_util_sch_ent_free

Purpose

Frees nonnull pointers in a **sec_attr_schema_entry_t** and the pointer to the **sec_attr_schema_entry_t** itself

Synopsis

```
#include <dce/sec_attr_util.h>

void sec_attr_util_sch_ent_free  (
     sec_attr_schema_entry_t **sec_sch_entry_p);
```

Parameters

Input/Output

******_sec_sch_entry_p_
: As input, the address of an allocated pointer to a potentially initialized **sec_attr_schema_entry_t**. As output, the address of a deallocated pointer that has been set to NULL.

Description

The **sec_attr_util_sch_ent_free()** routine frees each nonnull pointer in a **sec_attr_schema_entry_t** pointed to by a **sec_sch_entry_p*. The **sec_sch_entry_p* itself is also freed and set to NULL. A partially initialized **sec_attr_schema_entry_t** is handled correctly

Files

/usr/include/dce/sec_attr_util.idl
: The **idl** file from which **dce/sec_attr_util.h** was derived.

Related Information

Functions: **sec_attr_util_sch_ent_free_ptrs(3sec)**.

sec_attr_util_sch_ent_free_ptrs

Purpose

Frees nonnull pointers in a **sec_attr_schema_entry_t**

Synopsis

```
#include <dce/sec_attr_util.h>

void sec_attr_util_sch_ent_free_ptrs  (
     sec_attr_schema_entry_t *sec_sch_entry_p);
```

Parameters

Input/Output

sec_sch_entry_p
As input, a pointer to an allocated and potentially initialized **sec_attr_schema_entry_t**. As output, a pointer to a **sec_attr_schema_entry_t** with internal pointers freed and set to NULL.

Description

The **sec_attr_util_sch_ent_free_ptrs()** routine frees and sets to NULL each nonnull pointer in a **sec_attr_schema_entry_t** pointed to by *sec_sch_entry_p*. The *sec_sch_entry_p* itself is not freed. A partially initialized **sec_attr_schem_entry_t** is handled correctly.

Files

/usr/include/dce/sec_attr_util.idl
The **idl** file from which **dce/sec_attr_util.h** was derived.

Related Information

Functions: **sec_attr_util_sch_ent_free(3sec)**.

sec_cred_free_attr_cursor

Purpose

Frees the local resources allocated to a **sec_attr_cursor_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_cred_free_attr_cursor  (
     sec_cred_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor As input, a pointer to a **sec_cred_attr_cursor_t** whose resources are to be freed. As output a pointer to an initialized **sec_cred_attr_cursor_t** with allocated resources freed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_free_attr_cursor()** routine frees the resources assoicated with a cursor of type **sec_cred_attr_cursor_t** used by the **sec_cred_get_extended_attrs()** call.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

Related Information

Functions: **sec_cred_get_extended_attrs(3sec)**, **sec_cred_initialize_attr_cursor(3sec)**, **sec_intro(3sec)**.

sec_cred_free_cursor

Purpose

Releases local resources allocated to a **sec_cred_cursor_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_cred_free_cursor  (
     sec_cred_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor As input, a **sec_cred_cursor_t** whose resources are to be freed. As output, a **sec_cred_cursor_t** whose resources are freed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_free_cursor()** routine releases local resources allocated to a **sec_cred_cursor_t** used by the **sec_cred_get_delegate()** call.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_memory

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_initialize_cursor(3sec)**, **sec_intro(3sec)**.

sec_cred_free_pa_handle

Purpose

Frees the local resources allocated to a privilege attribute handle of type **sec_cred_pa_handle_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_cred_free_pa_handle  (
     sec_cred_pa_handle__t *pa_handle
     error_status_t *status);
```

Parameters

Input/Output

pa_handle
: As input, a pointer to a **sec_cred_pa_handle_t** whose resources are to be freed. As output a pointer to a **sec_cred_pa_handle_t** with allocated resources freed.

Output

status
: A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_free_pa_handle()** routine frees the resources assoicated with a privilege attribute handle of type **sec_cred_pa_handle_t** used by the **sec_cred_get_initiator()** and **sec_cred_get_delegate()** calls.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_authz_session_info

Purpose

Returns session-specific information that represents an authenticated client's credentials

Synopsis

```
#include <dce/sec_cred.h>

void  sec_cred_get_authz_session_info(
     rpc_authz_cred_handle_t callers_identity
     uuid_t *session_id
     sec_timeval_t *session_expiration
     error_status_t *status);
```

Parameters

Input

callers_identity
A credential handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the **rpc_binding_inq_auth_caller()** call.

Output

session_ID
A pointer to a **uuid_t** that identifies the client's DCE authorization session.

session_expiration
A pointer to a **sec_timeval_t** that specifies the expiration time of the authenticated client's credentials.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_get_authz_session_info()** routine retrieves session-specific information that represents the credentials of authenticated client specified by *callers_identity*. If the client is a member of a delegation chain, the information represents the credentials of all members of the chain.

The information can aid application servers in the construction of identity-based caches. For example, it could be used as a key into a cache of previously allocated delegation contexts and thus avoid the overhead of allocating a new login context on every remote operation. It could also be used as a key into a table of previously computed authorization decisions.

Before you execute this call, you must execute an **rpc_binding_inq_auth_caller()** call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_authz_cannot_comply

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_caller(3rpc)**, **sec_intro(3sec)**.

sec_cred_get_client_princ_name

Purpose

Returns the principal name associated with a credential handle

Synopsis

```
#include <dce/sec_cred.h>

void  sec_cred_get_client_princ_name(
     rpc_authz_cred_handle_t callers_identity
     unsigned_char_p_t *client_princ_name
     error_status_t *status);
```

Parameters

Input

callers_identity
A handle of type **rpc_authz_cred_handle_t** to the credentials for which to return the principal name. This handle is supplied as output of the **rpc_binding_inq_auth_caller()** call.

Output

client_princ_name
A pointer to the principal name of the server's RPC client.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_get_client_princ_name()** routine extracts the principal name associated with the credentials identified by *callers_pas*.

Before you execute **sec_cred_get_client_princ_name()**, you must execute an **rpc_binding_inq_auth_caller()** call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_authz_cannot_comply

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_caller(3sec)**, **sec_intro(3sec)**.

sec_cred_get_deleg_restrictions

Purpose

Returns delegate restrictions from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_restriction_set_t *sec_cred_get_deleg_restrictions(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A value of type **sec_cred_pa_handle_t** that provides a handle to a principal's privilege attributes. This handle is supplied as output of the **sec_cred_get_initiator()** call, the **sec_cred_get_delegate()** call and the **sec_login_cred** calls.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_deleg_restrictions()** routine extracts delegate restrictions from the privilege attribute handle identified by *callers_pas*. The restrictions are returned in a **sec_id_restriction_set_t**.

Before you execute **sec_cred_get_pa_data()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_delegate

Purpose

Returns a handle to the privilege attributes of an intermediary in a delegation chain

Synopsis

```
#include <dce/sec_cred.h>

sec_cred_pa_handle_t  sec_cred_get_delegate(
     rpc_authz_cred_handle_t callers_identity
     sec_cred_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input

callers_identity
A handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the **rpc_binding_inq_auth_caller()** call.

Input/Output

cursor As input, a pointer to a cursor of type **sec_cred_cursor_t** that has been initialized by the **sec_cred_initialize_cursor()** call. As an output parameter, *cursor* is a pointer to a cursor of type **sec_attr_srch_cursor_t** that is positioned past the principal whose privilege attributes have been returned in this call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_delegate()** routine returns a handle to the privilege attributes of an intermediary in a delegation chain that performed an authenticated RPC operation.

This call is used by servers. Clients use the **sec_login_cred_get_delegate()** routine to return the privilege attribute handle of an intermediary in a delegation chain.

The credential handle identified by *callers_identity* contains authentication and authorization information for all delegates in the chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of one of the delegates in the binding handle. The **sec_cred_pa_handle_t** returned by this call is used in other **sec_cred_get_** * calls to obtain privilege attribute information for a single delegate.

To obtain the privilege attributes of each delegate in the credential handle identified by *callers_identity*, execute this call until the message **sec_cred_s_no_more_entries** is returned.

Before you execute **sec_cred_get_delegate()**, you must execute

- An **rpc_binding_inq_auth_caller()** call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* parameter.
- A **sec_cred_initialize_cursor()** call to initialize a cursor of type **sec_cred_cursor_t**.

Use the **sec_cred_free_pa_handle()** all to free the resources associated with the **sec_cred_pa_handle_t**.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_auth_handle

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_caller(3rpc)**, **sec_cred_free_pa_handle()**, **sec_cred_get_deleg_restrictions(3sec)**, **sec_cred_get_delegation_type(3sec),sec_cred_get_extended_attrs(3sec)**, **sec_cred_get_opt_restrictions(3sec)**, **sec_cred_get_pa_date**, **sec_cred_get_req_restrictions(3sec)**, **sec_cred_get_tgt_restrictions(3sec)**, **sec_cred_get_v1_pac(3sec) sec_cred_initialize_cursor(3sec)**, **sec_intro(3sec)**.

sec_cred_get_delegation_type

Purpose

Returns the delegation type from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_delegation_type_t *sec_cred_get_delegation_type(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A value of type **sec_cred_pa_handle_t** that provides a handle to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_delegation_type ()** routine extracts the delegation type from the privilege attribute handle identified by *callers_pas* and returns it in a **sec_id_delegation_type_t**.

Before you execute **sec_cred_get_delegation_type()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_extended_attrs

Purpose

Returns extended attributes from a privilege handle

Synopsis

```
#include <dce/sec_cred.h>

void  sec_cred_get_extended_attrs(
     sec_cred_pa_handle_t callers_pas
     sec_cred_attr_cursor_t *cursor
     sec_attr_t *attr
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to the caller's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Input/Output

cursor A cursor of type **sec_cred_attr_cursor_t** that has been initialized by the **sec_cred_initialize_attr_cursor()** routine. As input *cursor* must be initialized. As output, *cursor* is positioned at the first attribute after the returned attribute.

Output

attr A pointer to a value of **sec_attr_t** that contains extended registry attributes.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_extended_attrs()** routine extracts extended registry initialized from the privilege attribute handle identified by *callers_pas*.

Before you execute call, you must execute

- A **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.
- A **sec_cred_initialize_attr_cursor()** to initialize a **sec_attr_t**.

To obtain all the extended registry attributes in the privilege attribute handle, repeat **sec_cred_get_extended_attrs()** calls until the status message **no_more_entries_available** is returned.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

Related Information

Functions: **sec_cred_get_initiator(3sec)**, **sec_cred_get_delegate(3sec)**, **sec_cred_initialize_attr_cursor(3sec)**, **sec_intro(3sec)**.

sec_cred_get_initiator

Purpose

Returns the privilege attributes of the initiator of a delegation chain

Synopsis

```
#include <dce/sec_cred.h>

sec_cred_pa_handle_t  sec_cred_get_initiator(
     rpc_authz_cred_handle_t callers_identity
     error_status_t *status);
```

Parameters

Input

callers_identity
A credential handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the **rpc_binding_inq_auth_caller()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_initiator()** routine returns a handle to the privilege attributes of the initiator of a delegation chain that performed an authenticated RPC operation.

The credential handle identified by *callers_identity* contains authentication and authorization information for all delegates in the chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of the client that initiated the delegation chain. The **sec_cred_pa_handle_t** returned by this call is used in other **sec_cred_get...** calls to obtain privilege attribute information for the initiator.

Before you execute **sec_cred_get_initiator()**, you must execute an **rpc_binding_inq_auth_caller()** call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_auth_handle

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **rpc_binding_inq_auth_caller(3rpc)**, **sec_cred_get_deleg_restrictions(3sec)**, **sec_cred_get_delegation_type(3sec)**,

sec_cred_get_extended_attrs(3sec), **sec_cred_get_opt_restrictions(3sec)**, **sec_cred_get_pa_date**, **sec_cred_get_req_restrictions(3sec)**, **sec_cred_get_tgt_restrictions(3sec)**, **sec_cred_get_v1_pac(3sec)**.

sec_cred_get_opt_restrictions

Purpose

Returns optional restrictions from a privilege handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_opt_req_t *sec_cred_get_opt_restrictions(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_opt_restrictions ()** routine extracts optional restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_restriction_set_t**.

Before you execute **sec_cred_get_pa_data()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_pa_data

Purpose

Returns identity information from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_pa_t *sec_cred_get_pa_data(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_pa_data()** routine extracts identity information from the privilege attribute handle specified by *callers_pas* and returns it in a **sec_id_pa_t**. The identity information includes an identifier of the princpal's locall cell and the principal's local and foreign group sets.

Before you execute **sec_cred_get_pa_data()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_req_restrictions

Purpose

Returns required restrictions from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_opt_req_t *sec_cred_get_req_restrictions(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_req_restrictions()** routine extracts required restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_opt_req_t**.

Before you execute **sec_cred_get_req_restrictions()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_tgt_restrictions

Purpose

Returns target restrictions from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_restriction_set_t *sec_cred_get_tgt_restrictions(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_tgt_restrictions()** routine extracts target restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_restriction_set_t**.

Before you execute **sec_cred_get_tgt_restrictions()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_get_v1_pac

Purpose

Returns pre-1.1 PAC from a privilege attribute handle

Synopsis

```
#include <dce/sec_cred.h>

sec_id_pac_t *sec_cred_get_v1_pac(
     sec_cred_pa_handle_t callers_pas
     error_status_t *status);
```

Parameters

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to the principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator()** call or **sec_cred_get_delegate()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

Description

The **sec_cred_get_v1_pac()** routine extracts the privilege attributes from a pre-1.1 PAC for the privilege attribute handle specified by *callers_pas* and returns them in a **sec_id_pa_t**.

Before you execute **sec_cred_get_v1_pac()**, you must execute a **sec_cred_get_initiator()** or **sec_cred_get_delegate()** call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* parameter.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_pa_handle

error_status_ok

Related Information

Functions: **sec_cred_get_delegate(3sec)**, **sec_cred_get_initiator(3sec)**, **sec_intro(3sec)**.

sec_cred_initialize_attr_cursor

Purpose

Initializes a **sec_attr_cursor_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_cred_initialize_attr_cursor  (
     sec_cred_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor As input, a pointer to a **sec_cred_attr_cursor_t** to be initialized. As output a pointer to an initialized **sec_cred_attr_cursor_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 **sec_cred_initialize_attr_cursor()** routine allocates and initializes a cursor of type **sec_cred_attr_cursor_t** for use with the **sec_cred_get_extended_attrs()** call. Use the **sec_cred_free_attr_cursor()** call to free the resources allocated to *cursor*.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_memory

error_status_ok

Related Information

Functions: **sec_cred_free_attr_cursor()**, **sec_cred_get_extended_attrs(3sec)**, **sec_intro(3sec)**.

sec_cred_initialize_cursor

Purpose

Initializes a **sec_cred_cursor_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_cred_initialize_cursor  (
     sec_cred_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor As input, a **sec_cred_cursor_t** to be initialized. As output, an initialized **sec_cred_cursor_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 **sec_cred_initialize_cursor()** routine initializes a cursor of type **sec_cursor_t** for use with the **sec_cred_get_delegate()** call. Use the **sec_cred_free_cursor()** call to free the resources allocated to *cursor*.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_memory

error_status_ok

Related Information

Functions: **sec_cred_free_cursor(3sec)**, **sec_cred_get_delegate(3sec)**, **sec_intro(3sec)**.

sec_cred_is_authenticated

Purpose

Returns TRUE if the supplied credentials are authenticated, and FALSE if they are not

Synopsis

```
#include <dce/sec_cred.h>

boolean32  sec_cred_is_authenticated(
     rpc_authz_cred_handle_t callers_identity
     error_status_t *status);
```

Parameters

Input

callers_identity
A handle of type **rpc_authz_cred_handle_t** to the credentials to check for authentication. This handle is supplied as output of the **rpc_binding_inq_auth_caller()** call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_cred_is_authenticated()** routine returns TRUE if the credentials identified by *callers_identity* are authenticated or FALSE if they are not.

Before you execute this call, you must execute an **rpc_binding_inq_auth_caller()** call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* parameter.

Files

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

Return Values

The routine returns **true** if the credentials are authenticated; **false** if they are not.

Related Information

Functions: **rpc_binding_inq_auth_caller(3rpc)**, **sec_intro(3sec)**.

sec_id_gen_group

Purpose

Generates a global name from cell and group UUIDs

Synopsis

```
#include <dce/secidmap.h>

void  sec_id_gen_group(
     sec_rgy_handle_t context
     uuid_t *cell_idp
     uuid_t *group_idp
     sec_rgy_name_t global_name
     sec_rgy_name_t cell_namep
     sec_rgy_name_t group_namep
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

cell_idp
A pointer to the UUID of the home cell of the group whose name is in question.

group_idp
A pointer to the UUID of the group whose name is in question.

Output

global_name
The global (full) name of the group in **sec_rgy_name_t** form.

cell_namep
The name of the group's home cell in **sec_rgy_name_t** form.

group_namep
The local (with respect to the home cell) name of the group in **sec_rgy_name_t** form.

status A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_id_gen_group()** routine generates a global name from input cell and group UUIDs. For example, given a UUID specifying the cell **/.../world/hp/brazil**, and a UUID specifying a group resident in that cell named **writers**, the routine would return the global name of that group, in this case, **/.../world/hp/brazil/writers**. It also returns the simple names of the cell and group, translated from the UUIDs.

The routine will not produce translations to any name for which a NULL pointer has been supplied.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_rgy_object_not_found
The registry server could not find the specified group.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_id_gen_name(3sec)**, **sec_id_parse_group(3sec)**, **sec_id_parse_name(3sec)**, **sec_intro(3sec)**.

sec_id_gen_name

Purpose

Generates a global name from cell and principal UUIDs

Synopsis

```
#include <dce/secidmap.h>

void  sec_id_gen_name(
     sec_rgy_handle_t context
     uuid_t *cell_idp
     uuid_t *princ_idp
     sec_rgy_name_t global_name
     sec_rgy_name_t cell_namep
     sec_rgy_name_t princ_namep
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

cell_idp
A pointer to the UUID of the home cell of the principal whose name is in question.

princ_idp
A pointer to the UUID of the principal whose name is in question.

Output

global_name
The global (full) name of the principal in **sec_rgy_name_t** form.

cell_namep
The name of the principal's home cell in **sec_rgy_name_t** form.

princ_namep
The local (with respect to the home cell) name of the principal in **sec_rgy_name_t** form.

status A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_id_gen_name()** routine generates a global name from input cell and principal UUIDs. For example, given a UUID specifying the cell **/.../world/hp/brazil**, and a UUID specifying a principal resident in that cell named **writers/tom**, the routine would return the global name of that principal, in this case, **/.../world/hp/brazil/writers/tom**. It also returns the simple names of the cell and principal, translated from the UUIDs.

The routine will not produce translations to any name for which a NULL pointer has been supplied.

Permissions Required

The **sec_id_gen_name()** routine requires at least one permission of any kind on the account associated with the input cell and principal UUIDs.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_rgy_object_not_found
The registry server could not find the specified principal.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_id_gen_group(3sec)**, **sec_id_parse_group(3sec)**, **sec_id_parse_name(3sec)**, **sec_intro(3sec)**.

sec_id_parse_group

Purpose

Translates a global name into group and cell names and UUIDs

Synopsis

```
#include <dce/secidmap.h>

void  sec_id_parse_group(
     sec_rgy_handle_t context
     sec_rgy_name_t global_name
     sec_rgy_name_t cell_namep
     uuid_t *cell_idp
     sec_rgy_name_t group_namep
     uuid_t *group_idp
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

global_name
The global (full) name of the group in **sec_rgy_name_t** form.

Output

cell_namep
The output name of the group's home cell in **sec_rgy_name_t** form.

cell_idp
A pointer to the UUID of the home cell of the group whose name is in question.

group_namep
The local (with respect to the home cell) name of the group in **sec_rgy_name_t** form.

group_idp
A pointer to the UUID of the group whose name is in question.

status A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_id_parse_group()** routine translates a global group name into a cell name and a cell-relative group name. It also returns the UUIDs associated with the group and its home cell.

The routine will not produce translations to any name for which a NULL pointer has been supplied.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_rgy_object_not_found
The registry server could not find the specified group.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_id_gen_group(3sec)**, **sec_id_gen_name(3sec)**, **sec_id_parse_group(3sec)**, **sec_id_parse_name(3sec)**, **sec_intro(3sec)**.

sec_id_parse_name

Purpose

Translates a global name into principal and cell names and UUIDs

Synopsis

```
#include <dce/secidmap.h>

void  sec_id_parse_name(
     sec_rgy_handle_t context
     sec_rgy_name_t global_name
     sec_rgy_name_t cell_namep
     uuid_t *cell_idp
     sec_rgy_name_t princ_namep
     uuid_t *princ_idp
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

global_name
The global (full) name of the principal in **sec_rgy_name_t** form.

Output

cell_namep
The output name of the principal's home cell in **sec_rgy_name_t** form.

cell_idp
A pointer to the UUID of the home cell of the principal whose name is in question.

princ_namep
The local (with respect to the home cell) name of the principal in **sec_rgy_name_t** form.

princ_idp
A pointer to the UUID of the principal whose name is in question.

status A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_id_parse_name()** routine translates a global principal name into a cell name and a cell-relative principal name. It also returns the UUIDs associated with the principal and its home cell.

The routine will not produce translations to any name for which a NULL pointer has been supplied.

Permissions Required

Only if *princ_idp* is requested as output does the **sec_id_parse_name()** routine require a permission. In this case, the routine requires at least one permission of any kind on the account whose global principal name is to be translated.

Files

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_rgy_object_not_found
The registry server could not find the specified principal.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_id_gen_name(3sec)**, **sec_intro(3sec)**.

sec_key_mgmt_change_key

Purpose

Changes a principal's key

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_change_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     unsigned32 key_vno
     void *keydata
     sec_timeval_period_t *garbage_collect_time
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** function.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key is to be changed.

key_vno
The version number of the new key. If 0 (zero) is specified, the routine will select the next appropriate key version number.

keydata
A pointer to a structure of type **sec_passwd_rec_t**.

Output

garbage_collect_time
The number of seconds that must elapse before all currently valid tickets (which are encoded with the current or previous keys) expire. At that time, all obsolete keys may be ''garbage collected,'' since no valid tickets encoded with those keys will remain outstanding on the network.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_change_key()** routine performs all activities necessary to update a principal's key to the specified value. This includes updating any local storage for the principal's key and also performing any remote operations needed to keep the authentication protocol (or network registry) current. Old keys for the principal are garbage collected if appropriate.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Any error condition will leave the key state unchanged.

sec_key_mgmt_e_key_unavailable
The old key is not present and therefore cannot be used to set a client side authentication context.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_auth_unavailable
The authentication protocol is not available to update the network database or to obtain the necessary network credentials.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_key_mgmt_e_key_unsupported
The key type is not supported.

sec_key_mgmt_e_key_version_ex
A key with this version number already exists.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

sec_rgy_object_not_found
No principal was found with the given name.

sec_login_s_no_memory
A memory allocation error occurred.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_gen_rand_key(3sec)**, **sec_key_mgmt_set_key(3sec)**.

sec_key_mgmt_delete_key

Purpose

Deletes a key from the local storage

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_delete_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     unsigned32 key_vno
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** function.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key is to be deleted.

key_vno
The version number of the desired key.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_delete_key()** routine deletes the specified key from the local key store. If an administrator ever discovers or suspects that the security of a server's key has been compromised, the administrator should delete the key immediately with **sec_key_mgmt_delete_key()**. This routine removes the key from the local key storage, which invalidates all extant tickets encoded with the key. If the compromised key is the current one, the principal should change the key with **sec_key_mgmt_change_key()** before deleting it. It is not an error for a process to delete the current key (as long as it is done *after* the network context has been established), but it may seriously inconvenience legitimate clients of a service.

This routine deletes all key types that have the specified key version number. A key type identifies the data encryption algorithm being used (for example, DES). This routine differs from **sec_key_mgmt_delete_key_type()** in that **sec_key_mgmt_delete_key_type()** deletes only the specified key version of the specified key type from the local key store.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Any error condition will leave the key state unchanged.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_delete_key_type(3sec)**, **sec_key_mgmt_garbage_collect(3sec)**.

sec_key_mgmt_delete_key_type

Purpose

Deletes a key version of a key type from the local key storage

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_delete_key_type(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     void *keytype
     unsigned32 key_vno
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key type is to be deleted.

keytype
A pointer to a value of type **sec_passwd_type_t**. The value identifies the data encryption algorithm that is being used (for example, DES).

key_vno
The version number of the desired key.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_delete_key_type()** routine deletes the specified key version of the specified key type from the local key store. It differs from **sec_key_mgmt_delete_key()** in that **sec_key_mgmt_delete_key()** deletes all key types that have the same key version number.

This routine removes the key from the local key storage, which invalidates all extant tickets encoded with the key. If the key in question is the current one, the principal should change the key with **sec_key_mgmt_change_key()** before deleting it. It is not an error for a process to delete the current key (as long as it is done *after* the network context has been established), but it may seriously inconvenience legitimate clients of a service.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

Any error condition will leave the key state unchanged.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_delete_key(3sec)**, **sec_key_mgmt_garbage_collect(3sec)**.

sec_key_mgmt_free_key

Purpose

Frees the memory used by a key value

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_free_key(
     void *keydata
     error_status_t *status);
```

Parameters

Input

keydata
A pointer to a structure of type **sec_passwd_rec_t**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **sec_key_mgmt_free_key()** routine releases any storage allocated for the indicated key data by **sec_key_mgmt_get_key()**. The storage for the key data returned by **sec_key_mgmt_get_key()** is dynamically allocated.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_get_key(3sec)**.

sec_key_mgmt_garbage_collect

Purpose

Deletes obsolete keys

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_garbage_collect(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key information is to be garbage collected.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_garbage_collect()** routine discards any obsolete key information for this principal. An obsolete key is one that can only decode invalid tickets. As an example, consider a key that was in use on Monday, and was only used to encode tickets whose maximum lifetime was 1 day. If that key was changed

at 8:00 a.m. Tuesday morning, then it would become obsolete by 8:00 a.m. Wednesday morning, at which time there could be no valid tickets outstanding.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_key_mgmt_e_key_unavailable
Requested key not present.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

sec_rgy_object_not_found
No principal was found with the given name.

sec_login_s_no_memory
A memory allocation error occurred.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_delete_key(3sec)**.

sec_key_mgmt_gen_rand_key

Purpose

Generates a new random key of a specified key type

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_gen_rand_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     void *keytype
     unsigned32 key_vno
     void **keydata
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal for whom the key is to be generated.

keytype
A pointer to a value of type **sec_passwd_type_t**. The value identifies the data encryption algorithm to be used for the key (for example, DES).

key_vno
The version number of the new key.

Output

keydata
A pointer to a value of **sec_passwd_rec_t**. The storage for *keydata* is allocated dynamically, so the returned pointer actually indicates a pointer to the key value. The storage for this data may be freed with the **sec_key_mgmt_free_key()** function.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_gen_rand_key()** routine generates a new random key for a specified principal and of a specified key type. The generated key can be used with the **sec_key_mgmt_change_key()** and **sec_key_mgmt_set_key()** routines.

Note that to initialize the random keyseed, the process must first make an authenticated call such as **sec_rgy_site_open()**.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_not_implemented
The specified key type is not supported.

sec_s_no_key_seed
No random key seed has been set.

sec_s_no_memory
Unable to allocate memory.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_change_key(3sec)**, **sec_key_mgmt_set_key(3sec)**.

sec_key_mgmt_get_key

Purpose

Retrieves a key from local storage

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_get_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     unsigned32 key_vno
     void **keydata
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal to whom the key belongs.

key_vno
The version number of the desired key. To return the latest version of the key, set this parameter to **sec_c_key_version_none**.

Output

keydata
A pointer to a value of type **sec_passwd_rec_t**. The storage for *keydata* is allocated dynamically, so the returned pointer actually indicates a pointer to the key value. The storage for this data may be freed with the **sec_key_mgmt_free_key()** routine.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_get_key()** routine extracts the specified key from the local key store.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_s_no_memory
Unable to allocate memory.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_key_mgmt_get_next_key

Purpose

Retrieves successive keys from the local key storage

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_get_next_key(
     void *cursor
     idl_char **principal_name
     unsigned32 *key_vno
     void **keydata
     error_status_t *status);
```

Parameters

Input

cursor A pointer to the current cursor position in the local key storage. The cursor position is set via the routine **sec_key_mgmt_initialize_cursor()**.

Output

principal_name
A pointer to a character string indicating the name of the principal associated with the extracted key. Free the storage for the principal name with the **free()** function.

key_vno
The version number of the extracted key.

keydata
A pointer to a value of type **sec_passwd_rec_t**. The storage for *keydata* is allocated dynamically, so the returned pointer actually indicates a pointer to the key value. The storage for this data may be freed with the **sec_key_mgmt_free_key()** function.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_get_next_key()** routine extracts the key pointed to by the cursor in the local key store and updates the cursor to point to the next key. By repeatedly calling this routine you can scan all the keys in the local store.

Files

/usr/lib/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_s_no_memory
Unable to allocate memory.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_get_key(3sec)**, **sec_key_mgmt_initialize_cursor(3sec)**.

sec_key_mgmt_get_next_kvno

Purpose

Retrieves the next eligible key version number for a key

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_get_next_kvno(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     void *keytype
     unsigned32 *key_vno
     unsigned32 *next_key_vno
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal associated with the key.

keytype
A pointer to a value of type **sec_passwd_type_t**. The value identifies the data encryption algorithm (for example, DES) being used for the key.

Output

key_vno
The current version number of the key. Specify NULL if you do not need this value to be returned.

next_key_vno
The next eligible version number for the key. Specify NULL if you do not need this value to be returned.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_get_next_kvno()** routine returns the current and next eligible version numbers for a key from the registry server (not from the local key table). The key is identified via its associated authentication protocol, principal name, and key type. The *arg* value associated with the key is also specified.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

sec_rgy_object_not_found
No principal was found with the given name.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_key_mgmt_initialize_cursor

Purpose

Repositions the cursor in the local key store

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_initialize_cursor(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     void *keytype
     void **cursor
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key is to be accessed. To access all keys in the local key store, supply NULL for this parameter.

keytype
A pointer to the data encryption algorithm (for example, DES) being used for the key.

Output

cursor The returned cursor value. The storage for the cursor information is allocated dynamically, so the returned pointer actually indicates a pointer to the cursor value. The storage for this data may be freed with the **sec_key_mgmt_release_cursor()** routine.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_initialize_cursor()** routine resets the cursor in the local key store.

Use this routine to reposition the cursor before performing a scan of the local store via **sec_key_mgmt_get_next_key()**. The returned cursor value is supplied as input to **sec_key_mgmt_get_next_key()**.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_s_no_memory
Unable to allocate memory.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_get_next_key(3sec)**, **sec_key_mgmt_release_cursor(3sec)**.

sec_key_mgmt_manage_key

Purpose

Automatically changes a principal's key before it expires

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_manage_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit ktadd** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal whose key is to be managed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_manage_key()** routine changes the specified principal's key on a regular basis, as determined by the local cell's policy. It will run indefinitely, never returning during normal operation, and therefore should be invoked only from a thread that has been devoted to managing keys.

This routine queries the DCE registry to determine the password expiration policy that applies to the named principal. It then idles until a short time before the current key is due to expire and then uses the **sec_key_mgmt_gen_rand_key()** to produce a new random key, updating both the local key store and the DCE registry. This routine also invokes **sec_key_mgmt_garbage_collect()** as needed.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_key_unavailable
The old key is not present and therefore cannot be used to set a client side authentication context.

sec_key_mgmt_e_key_unsupported
The key type is not supported.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

sec_rgy_object_not_found
No principal was found with the given name.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_gen_rand_key(3sec)**, **sec_key_mgmt_garbage_collect(3sec)**.

sec_key_mgmt_release_cursor

Purpose

Releases the memory used by an initialized cursor value

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_release_cursor(
     void **cursor
     error_status_t *status);
```

Parameters

Input

cursor A pointer to the cursor value for which the storage is to be released.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**.

Description

The **sec_key_mgmt_release_cursor()** routine releases any storage allocated for the indicated cursor value by **sec_key_mgmt_initialize_cursor()**. The storage for the cursor value returned by **sec_key_mgmt_initialize_cursor()** is dynamically allocated.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_initialize_cursor(3sec)**.

sec_key_mgmt_set_key

Purpose

Inserts a key value into the local storage

Synopsis

```
#include <dce/keymgmt.h>

void  sec_key_mgmt_set_key(
     sec_key_mgmt_authn_service authn_service
     void *arg
     idl_char *principal_name
     unsigned32 key_vno
     void *keydata
     error_status_t *status);
```

Parameters

Input

authn_service
Identifies the authentication protocol using this key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local key file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default key file (**/krb/v5srvtab**) should be used. A key filename specifies that file should be used as the key file. The filename must begin with **FILE:**. If the filename does not begin with **FILE:**, the code will add it.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

principal_name
A pointer to a character string indicating the name of the principal associated with the key to be set.

key_vno
The version number of the key to be set.

keydata
A pointer to the key value to be set.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_key_mgmt_set_key()** routine performs all local activities necessary to update a principal's key to the specified value. This routine will not update the authentication protocol's value for the principal's key.

In some circumstances, a server may only wish to change its key in the local key storage, and not in the DCE registry. For example, a database system may have several replicas of a master database, managed by servers running on independent machines. Since these servers together represent only one service, they should all share the same key. This way, a user with a ticket to use the database can choose whichever server is least busy. To change the database key, the master server would signal all the replica (slave) servers to change the current key in their local key storage. They would use the **sec_key_mgmt_set_key()** routine, which does not communicate with the DCE registry. Once all the slaves have complied, the master server can then change the registry key and its own local storage.

Files

/usr/include/dce/keymgmt.idl
The **idl** file from which **dce/keymgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_key_mgmt_e_key_unavailable
The old key is not present and therefore cannot be used to set a client side authentication context.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_key_mgmt_e_key_unsupported
The key type is not supported.

sec_key_mgmt_e_key_version_ex
A key with this version number already exists.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_key_mgmt_change_key(3sec)**, **sec_key_mgmt_gen_rand_key(3sec)**.

sec_login_become_delegate

Purpose

Causes an intermediate server to become a delegate in traced delegation chain

Synopsis

```
#include <dce/sec_login.h>

sec_login_handle_t
sec_login_become_delegate(
     rpc_authz_cred_handle_t callers_identity
     sec_login_handle_t my_login_context
     sec_id_delegation_type_t delegation_type_permitted
     sec_id_restriction_set_t *delegate_restrictions
     sec_id_restriction_set_t *target_restrictions
     sec_id_opt_req_t *optional_restrictions
     sec_id_opt_req_t *required_restrictions
     sec_id_compatibility_mode_t compatibility_mode
     error_status_t *status);
```

Parameters

Input

callers_identity
A handle of type **rpc_authz_cred_handle_t** to the authenticated identity of the previous delegate in the delegation chain. The handle is supplied by the **rpc_binding_inq_auth_caller()** call.

my_login_context
A value of **sec_login_handle_t** that provides an opaque handle to the identity of the client that is becoming the intermediate delegate. The **sec_login_handle_t** that specifies the client's identity is supplied as output of the following calls:

- **sec_login_get_current_context()**, if the client inherited the identity of the current context
- The **sec_login_setup_identity()** and the **sec_login_validate_identity()** pair that together establish an authenticated identity if a new identity was established

Note that this identity specified by **sec_login_handle_t** must be a simple login context; it cannot be a compound identity created by a previous **sec_login_become_delegate()** call.

delegation_type_permitted
A value of **sec_id_delegation_type_t** that specifies the type of delegation to be enabled. The types available are as follows:

sec_id_deleg_type_none
No delegation.

sec_id_deleg_type_traced
Traced delegation.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation.

Note that the initiating client sets the type of delegation. If it is set as traced, all delegates must also specify traced delegation; they cannot specify simple delegation. The same is true if the initiating client sets the delegation type as simple; all subsequent delegates must also specify simple delegation. The intermediate delegates can, however, specify no delegation to indicate that the delegation chain can proceed no further.

delegate_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as delegates for the intermediate client identified by *my_login_context*. These servers are added to delegates permitted by the *delegate_restrictions* parameter of the **sec_login_become_initiator** call.

target_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as targets for the intermediate client identified by *my_login_context*. These servers are added to targets specified by the *target_restrictions* parameter of the **sec_login_become_initiator** call.

optional_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined optional restrictions that apply to the intermediate client identified by *my_login_context*. These restrictions are added to the restrictions identified by the *optional_restrictions* parameter of the **sec_login_become_initiator** call.

required_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined required restrictions that apply to the intermediate client identified by *my_login_context*. These restrictions are added to the restrictions identified *required_restrictions* parameter of the **sec_login_become_initiator** call.

compatibility_mode
A value of **sec_id_compatibility_mode_t** that specifies the compatibility mode to be used when the intermediate client operates on pre-1.1 servers. The modes available are as follows:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The pre-1.1 PAC data is extracted from the EPAC of the initiating client.

sec_id_compat_mode_caller
Compatibility mode is on. The pre-1.1 PAC data extracted from the EPAC of the last client in the delegation chain.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_become_delegate()** is used by intermediate servers to become a delegate for the client identified by *callers_identity*. The routine returns a new login context (of type **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate and target restrictions, and any application-defined optional and required restrictions.

The new login context created by this call can then used to to set up authenticated rpc with an intermediate or target server using the **rpc_binding_set_auth_info()** call.

Any delegate, target, required, or optional restrictions specified in this call are added to the restrictions specified by the initiating client and any intermediate clients.

The **sec_login_become_delegate()** call is run only if the initiating client enabled traced delegation by setting the *delegation_type_permitted* parameter in the **sec_login_become_initiator** call to **sec_id_deleg_type_traced**.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

sec_login_s_invalid_context

sec_login_s_compound_delegate

sec_login_s_invalid_deleg_type

sec_login_s_invalid_compat_mode

sec_login_s_deleg_not_enabled

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_caller(3rpc)**, **sec_intro(3sec)**, **sec_login_become_impersonator(3sec)**, **sec_login_become_initiator(3sec)**, **sec_login_get_current_context(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity()**.

sec_login_become_impersonator

Purpose

Used by a server to create a login context and assoicated handle that impersonates the identity of a caller

Synopsis

```
#include <dce/sec_login.h>

sec_login_handle_t
sec_login_become_impersonator(
     rpc_authz_cred_handle_t callers_identity
     sec_login_handle_t my_login_context
     sec_id_delegation_type_t delegation_type_permitted
     sec_id_restriction_set_t *delegate_restrictions
     sec_id_restriction_set_t *target_restrictions
     sec_id_opt_req_t *optional_restrictions
     sec_id_opt_req_t *required_restrictions
     error_status_t *status);
```

Description

The **sec_login_become_impersonator()** is used by intermediate servers to become an impersonator for the client identified by *callers_identity*. The routine returns a new login context (of type **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate, and target restrictions, and any application-defined optional and required restrictions.

The new login context created by this call can then used to to set up authenticated rpc with an intermediate or target server using the **rpc_binding_set_auth_info()** call.

The effective optional and required restrictions are the union of the optional and required restrictions specified in this call and specified by the initiating client and any intermediate clients. The effective target and delegate restrictions are the intersection of the target and delegate restrictions specified in this call and specified by the initiating client and any intermediate clients.

The **sec_login_become_impersonator** call is run only if the initiating client enabled simple delegation by setting the *delegation_type_permitted* parameter in the **sec_login_become_initiator** call to **sec_id_deleg_type_simple**.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

sec_login_s_invalid_deleg_type

sec_login_s_invalid_compat_mode

sec_login_s_deleg_not_enabled

error_status_ok

Related Information

Functions: **rpc_binding_inq_auth_caller(3rpc)**, **sec_intro(3sec)**, **sec_login_become_initiator(3sec)**.

sec_login_become_initiator

Purpose

Constructs a new login context that enables delegation for the calling client

Synopsis

```
#include <dce/sec_login.h>

sec_login_handle_t
sec_login_become_initiator(
     sec_login_handle_t my_login_context
     sec_id_delegation_type_t delegation_type_permitted
     sec_id_restriction_set_t *delegate_restrictions
     sec_id_restriction_set_t *target_restrictions
     sec_id_opt_req_t *optional_restrictions
     sec_id_opt_req_t *required_restrictions
     sec_id_compatibility_mode_t compatibility_mode
     error_status_t *status);
```

Parameters

Input

my_login_context
A value of **sec_login_handle_t** that provides an opaque handle to the identity of the client that is enabling delegation. The **sec_login_handle_t** that specifies the client's identity is supplied as output of the following calls:

- **sec_login_get_current_context()** if the client inherited the identity of the current context
- The **sec_login_setup_identity()** and the **sec_login_validate_identity()** pair that together establish an authentiated identity if a new identity was established

delegation_type_permitted
A value of **sec_id_delegation_type_t** that specifies the type of delegation to be enabled. The types available are as follows:

sec_id_deleg_type_none
No delegation.

sec_id_deleg_type_traced
Traced delegation.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation.

Note each subsequent intermediate delegate of the delegation chain started by the initiating client must set the delegation type to traced if the initiating client set it to traced or to simple if the initiating client set it to simple. Intermediate delegates, however, can set the delegation type to no delegation to indicate that the delegation chain can proceed no further.

delegate_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as delegates for the client initiating delegation.

target_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as targets for the client initiating delegation.

optional_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined optional restrictions that apply to the client initiating delegation.

required_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined required restrictions that apply to the client initiating delegation.

compatibility_mode
A value of **sec_id_compatibility_mode_t** that specifies the compatibility mode to be used when the initiating client interacts with pre-1.1 servers. The modes available are as follows:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The pre-1.1 PAC data is extracted from the EPAC of the initiating client.

sec_id_compat_mode_caller
Compatibility mode is on. The pre-1.1 PAC data extracted from the EPAC of the last client in the delegation chain.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_become_initiator()** enables delegation for the calling client by constructing a new login context (in a **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate, and target restrictions, and any application-defined optional and required restrictions.

The new login context is then used to to set up authenticated rpc with an intermediate server using the **rpc_binding_set_auth_info()** call. The intermediary can continue the delegation chain by calling **sec_login_become_delegate** (if the delegation type is **sec_id_deleg_type_traced**) or **sec_login_become_impersonator** (if the delegation type is **sec_id_deleg_type_impersonation**).

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

error_status_ok

sec_login_s_invalid_compat_mode

sec_login_s_invalid_context

sec_login_s_invalid_deleg_type

Related Information

Functions: **sec_intro(3sec)**, **sec_login_become_delegate(3sec)**, **sec_login_become_impersonator(3sec)**, **sec_login_get_current_context(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity()**.

sec_login_certify_identity

Purpose

Certifies the network authentication service

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_certify_identity(
     sec_login_handle_t login_context
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_certify_identity()** routine certifies that the security server used to set up and validate a login context is legitimate. A legitimate server is one that knows the host machine's secret key. On some systems, this may be a privileged operation.

Information may be retrieved via **sec_login_get_pwent()**, **sec_login_get_groups()**, and **sec_login_get_expiration()** from an uncertified login context, but such information cannot be trusted. All system login programs that use the **sec_login** interface must call **sec_login_certify_identity()** to certify the security server. If they do not, they open the local file system to attacks by imposter Security servers returning suspect local process credentials (UUID and group IDs). This operation updates the local registry with the login context credentials if the certification check succeeds.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Return Values

The routine returns a **boolean32** value that is TRUE if the certification was successful, and FALSE otherwise.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_config
The DCE configuration (**dce_config**) information is not available.

sec_login_s_context_invalid
The input context is invalid.

sec_login_s_default_use
It is an error to try to certify the default context.

error_status_ok
The call was successful.

Examples

Applications wishing to perform a straightforward login can use the **sec_login** package as follows:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags, &login_context,
                             &st)) {
   ... get password from user...

   if (sec_login_validate_identity(login_context, password,
                             &reset_passwd, &auth_src, &st)) {

      if (!sec_login_certify_identity(login_context, &st))
          exit(error_weird_auth_svc);

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      if (reset_passwd) {
         ... get new password from user, reset registry record ...
      };

      sec_login_get_pwent(login_context, &pw_entry, &st);

      if (pw_entry.pw_expire < todays_date) {
         sec_login_purge_context(&login_context, &st);
         exit(0)
      }

      ... any other application specific login valid actions ...
      }

   } else {
      sec_login_purge_context(&login_context, &st);

      ... application specific login failure actions ...
   }
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_get_expiration(3sec)**, **sec_login_get_groups(3sec)**, **sec_login_get_pwent(3sec)**.

sec_login_cred_get_delegate

Purpose

Returns a handle to the privilege attributes of an intermediary in a delegation chain. Used by clients.

Synopsis

```
#include <dce/sec_login.h>

sec_cred_pa_handle_t  sec_login_cred_get_delegate(
     sec_login_handle_t login_context
     sec_cred_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input

login_context
A value of **sec_login_handle_t** that provides an opaque handle to a login context for which delegation has been enabled. The **sec_login_handle_t** that specifies the identity is supplied as output of the **sec_login_become_delegate()** call.

Input/Output

cursor As input, a pointer to a cursor of type **sec_cred_cursor_t** that has been initialized by the **sec_login_cred_init_cursor()** call. As an output parameter, *cursor* is a pointer to a cursor of type **sec_cred_cursor_t** that is positioned past the principal whose privilege attributes have been returned in this call.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_cred_get_delegate()** routine returns a handle of type **sec_login_handle_t** to the the privilege attributes of an intermediary in a delegation chain that performed an authenticated RPC operation.

This call is used by clients. Servers use the **sec_cred_get_delegate()** routine to return the privilege attribute handle of an intermediary in a delegation chain.

The login context identified by *login_context* contains all members in the delegation chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of one of the delegates in the login context. The **sec_cred_pa_handle_t** returned by this call is used in other **sec_cred_get_** * calls to obtain privilege attribute information for a single delegate.

To obtain the privilege attributes of each delegate in the credential handle identified by *callers_identity*, execute this call until the message **sec_cred_s_no_more_entries** is returned.

Before you execute **sec_login_cred_get_delegate()**, you must execute a **sec_login_cred_init_cursor()** call to initialize a cursor of type **sec_cred_cursor_t**.

Use the **sec_cred_free_pa_handle() sec_cred_free_cursor()** calls to free the resources allocated to the **sec_cred_pa_handle_t** and *cursor*.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_cred_get_deleg_restrictions(3sec)**, **sec_cred_get_delegation_type(3sec)**, **sec_cred_get_extended_attrs(3sec)**, **sec_cred_get_opt_restrictions(3sec)**, **sec_cred_get_pa_date(3sec)**, **sec_cred_get_req_restrictions(3sec)**, **sec_cred_get_tgt_restrictions(3sec)**, **sec_cred_get_v1_pac(3sec)**, **sec_login_cred_init_cursor(3sec)**.

sec_login_cred_get_initiator

Purpose

Returns information about the delegation initiator in a specified login context

Synopsis

```
#include <dce/sec_login.h>

sec_cred_pa_handle_t  sec_login_cred_get_initiator(
     sec_login_handle_t login_context
     error_status_t *status);
```

Parameters

Input

login_context
A value of **sec_login_handle_t** that provides an opaque handle to a login context for which delegation has been enabled.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_cred_get_initiator()** routine returns a handle of type **sec_cred_pa_handle_t** to the privilege attributes of the delegation initiator.

The login context identified by *login_context* contains all members in the delegation chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of the initiator. The **sec_cred_pa_handle_t** returned by this call is used in other **sec_cred_get_** * calls to obtain privilege attribute information for the initiator single delegate.

Use the **sec_cred_free_pa_handle()** call to free the resources allocated to the **sec_cred_pa_handle_t** handle.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_invalid_context

error_status_ok

Related Information

Functions: **sec_cred_get_deleg_restrictions(3sec)**, **sec_cred_get_delegation_type(3sec)**, **sec_cred_get_extended_attrs(3sec)**, **sec_cred_get_opt_restrictions(3sec)**, **sec_cred_get_pa_date(3sec)**, **sec_cred_get_req_restrictions(3sec)**, **sec_cred_get_tgt_restrictions(3sec)**, **sec_cred_get_v1_pac(3sec)**, **sec_intro(3sec)**.

sec_login_cred_init_cursor

Purpose

Initializes a **sec_cred_cursor_t**

Synopsis

```
#include <dce/sec_cred.h>

void sec_login_cred_init_cursor  (
     sec_cred_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor As input, a pointer to a **sec_cred_cursor_t** to be initialized. As output, a pointer to an initialized **sec_cred_cursor_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 **sec_login_cred_init_cursor()** routine allocates and initializes a cursor of type **sec_cursor_t** for use with the **sec_login_cred_get_delegate()** call.

Use the **sec_cred_free_cursor()** call to free the resources allocated to *cursor*.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_cred_s_invalid_cursor

sec_login_s_no_memory

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_login_cred_get_delegate(3sec)**.

sec_login_disable_delegation

Purpose

Disables delegation for a specified login context

Synopsis

```
#include <dce/sec_login.h>

sec_logon_handle_t *sec_login_disable_delegation(
     sec_login_handle_t login_context
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context for which delegation has been enabled.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_disable_delegation()** routine disables delegation for a specified login context. It returns a new login context of type **sec_login_handle_t** without any delegation information, thus preventing any further delegation.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_invalid_context

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_login_become_delegate(3sec)**, **sec_login_become_impersonator(3sec)**, **sec_login_become_initiator(3sec)**.

sec_login_export_context

Purpose

Creates an exportable login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_export_context(
     sec_login_handle_t login_context
     unsigned32 buf_len
     idl_byte buf[ ]
     unsigned32 *len_used
     unsigned32 *len_needed
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

buf_len
An unsigned 32-bit integer containing the allocated length (in bytes) of the buffer that is to contain the login context.

Output

*buf***[ ]** An idl_byte array that contains the exportable login context upon return.

len_used
A pointer to an unsigned 32-bit integer indicating the number of bytes needed for the entire login context, up to *buf_len*.

len_needed
If the allocated length of the buffer is too short, an error is issued (**sec_login_s_no_memory**), and on return this pointer indicates the number of bytes necessary to contain the login context.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_export_context()** routine obtains an exportable version of the login context information. This information may be passed to another process running on the same machine.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_memory
Not enough space was allocated for the *buf*[ ] array. The *len_needed* parameter will point to the needed length.

sec_login_s_handle_invalid
The login context handle is invalid.

sec_login_s_context_invalid
The login context specified by the input handle is invalid.

Related Information

Functions: **sec_login_import_context(3sec)**, **sec_intro(3sec)**.

sec_login_free_net_info

Purpose

Frees storage allocated for a principal's network information

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_free_net_info(
     sec_login_net_info_t *net_info);
```

Parameters

Input/Output

net_info
A pointer to the **sec_login_net_info_t** structure to be freed.

Description

The **sec_login_free_net_info()** routine frees any memory allocated for a principal's network information. Network information is returned by a previous successful call to **sec_login_inquire_net_info()**.

Cautions

This routine does not return any completion codes. Make sure that you supply a valid **sec_login_net_info_t** address. The routine simply frees a range of storage beginning at the supplied address, without regard to the actual contents of the storage.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_inquire_net_info(3sec)**.

sec_login_get_current_context

Purpose

Returns a handle to the current login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_get_current_context(
     sec_login_handle_t *login_context
     error_status_t *status);
```

Parameters

Output

login_context
A pointer to an opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_get_current_context()** routine retrieves a handle to the login context for the currently established network identity. The context returned is created from locally cached data so subsequent data extraction operations may return some NULL values.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
There was no current context to retrieve. (See **sec_login_setup_identity(3sec)** for information about how to set up, validate, and implement a login context.)

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_get_current_context()** routine as part of a process to change the groupset:

```
sec_login_get_current_context(&login_context, &st);

sec_login_get_groups(login_context, &num_groups, &groups, &st);

   ...the group IDs have to be converted from the returned UNIX
   numbers into UUIDs (use sec_rgy_pgo_unix_num_to_id(3sec)...

for (i=0; i < num_groups; i++) {
   ... query whether the user wants to discard any of the current
   group memberships.  Copy new groupset to the new_groups array ...
}

if ( !sec_login_newgroups(sec_login_no_flags, num_new_groups,
        new_groups, &login_context, &st)) {
   if (st == sec_login_s_groupset_invalid)
        printf("New groupset invalid);

   ... application specific error handling ...
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_setup_identity(3sec)**.

sec_login_get_expiration

Purpose

Returns the TGT lifetime for an authenticated identity

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_get_expiration(
     sec_login_handle_t login_context
     signed32 *identity_expiration
     error_status_t *status);
```

Parameters

Input

login_context
: An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

Output

identity_expiration
: The lifetime of the ticket-granting ticket (TGT) belonging to the authenticated identity identified by *login_context*. It can be used in the same ways as a UNIX **time_t**.

status A pointer to the completion status.

Description

The **sec_login_get_expiration()** routine extracts the lifetime for the TGT belonging to the authenticated identity contained in the login context. The lifetime value is filled in if available; otherwise, it is set to 0 (zero). This routine allows an application to tell an interactive user how long the user's network login (and authenticated identity) will last before having to be refreshed.

The routine works only on previously certified contexts.

Files

/usr/include/dce/sec_login.idl
: The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
: The login context itself is invalid.

sec_login_s_default_use
There was illegal use of the default login handle.

sec_login_s_not_certified
The login context has not been certified.

sec_login_s_no_current_context
The calling process has no context of its own.

error_status_ok
The call was successful.

Examples

Since the authenticated network identity for a process has a finite lifetime, there is a risk it will expire during some long network operation, preventing the operation from completing. To avoid this situation, an application might, before initiating a long operation, use the **sec_login** package to check the expiration time of its identity and refresh it if there is not enough time remaining to complete the operation. After refreshing the identity, the process must validate it again with **sec_login_validate_identity()**.

```
sec_login_get_expiration(login_context, &expire_time, &st);

if (expire_time < (current_time + operation_duration)) {

     if (!sec_login_refresh_identity(login_context, &st)) {
      if (st == sec_login_s_refresh_ident_bad) {
         ... identity has changed ...
      } else {
         ... login context cannot be renewed ...
         exit(error_context_not_renewable);
      }

     if (sec_login_validate_identity(login_context, password,
                           &reset_passwd, &auth_src, &st)) {
         ... identity validated ...
      } else {
         ... validation failed ...
         exit(error_validation_failure);
      }
   }
}

operation();
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_get_current_context(3sec)**.

sec_login_get_groups

Purpose

Returns the group set from a login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_get_groups(
     sec_login_handle_t login_context
     unsigned32 *num_groups
     signed32 **group_set
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

Output

num_groups
An unsigned 32-bit integer indicating the total number of groups returned in the *group_set* array.

group_set
The list of groups to which the user belongs.

status A pointer to the completion status.

Description

The **sec_login_get_groups()** routine returns the groups contained in the supplied login context. Part of a network identity is a list of the various groups to which the principal belongs. The groups are used to determine a user's access to various objects and services. This routine extracts from the login context a list of the groups for which the user has established network privileges.

The routine works only on previously validated contexts.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
The login context itself is not valid.

sec_login_s_info_not_avail
The login context has no UNIX information.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_not_certified
The login context has not been certified.

sec_login_s_not_certified
The login context is not certified.

sec_rgy_object_not_found
The registry server could not find the specified login context data.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_get_groups()** routine as part of a process to change the groupset:

```
sec_login_get_current_context(&login_context, &st);

sec_login_get_groups(login_context, &num_groups, &groups, &st);

   ...the group IDs have to be converted from the returned UNIX
   numbers into UUIDs (use sec_rgy_pgo_unix_num_to_id(3sec)...

for (i=0; i < num_groups; i++) {
   ... query whether the user wants to discard any of the current
   group memberships.  Copy new groupset to the new_groups array ...
}

if ( !sec_login_newgroups(sec_login_no_flags, num_new_groups,
        new_groups, &login_context, &st)) {
   if (st == sec_login_s_groupset_invalid)
        printf("New groupset invalid);

   ... application specific error handling ...
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_get_projlist(3sec)**.

sec_login_get_pwent

Purpose

Returns a passwd-style entry for a login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_get_pwent(
     sec_login_handle_t login_context
     sec_login_passwd_t *pwent
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and Universal Unique Identifier (UUID), account restrictions, records of group membership, and the process home directory. (See the **sec_intro(3sec)** reference page for more details about the login context.)

Output

pwent A pointer to a pointer to the returned **passwd** -style structure. The particular structure depends on the underlying system. For example, on a system with a **passwd** structure like that supported by 4.4BSD and OSF/1, the structure (found in **/usr/include/pwd.h**) is as follows:

```
struct passwd {
    char   *pw_name;     /* user name */
    char   *pw_passwd;   /* encrypted password */
    int     pw_uid;      /* user uid */
    int     pw_gid;      /* user gid */
    time_t  pw_change;   /* password change time */
    char   *pw_class;    /* user access class */
    char   *pw_gecos;    /* miscellaneous account info */
    char   *pw_dir;      /* home directory */
    char   *pw_shell;    /* default shell */
    time_t  pw_expire;   /* account expiration */
};
```

status A pointer to the completion status. On successful completion, the routine returns one of the following status codes:

error_status_ok
Indicates that the login context has been validated and certified.

sec_login_s_not_certified
Indicates that the login context has been validated, but not certified. Although this code indicates successful completion, it warns you that the context is not validated.

If the call does not complete successfully, it returns an error.

Description

The **sec_login_get_pwent()** routine creates a **passwd** -style structure for the current network login context. This is generally useful for establishing the local operating system context. Applications that require all of the data normally extracted via **getpwnam()** should extract that data from the login context with this call.

This routine works only on explicitly created (not inherited or imported) contexts.

CAUTION:

The returned sec_login_passwd_t structure points to data stored in the structure indicated by the *login_context* pointer, and must be treated as read-only data. Writing to these data objects may cause unexpected failures.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
The login context itself is invalid.

sec_login_s_not_certified
The login context has not been certified.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_info_not_avail
The login context has no UNIX information.

sec_rgy_object_not_found
The registry server could not find the specified login context data.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_get_pwent()** routine:

```
#include <pwd.h>
...
struct passwd *pwd;
...
sec_login_get_pwent(login_context,(sec_login_passwd_t*)&pwd,&status);
...
printf ("%s",pwd->pw_name);
```

Related Information

Functions: **sec_intro(3sec)**.

sec_login_import_context

Purpose

Imports a login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_import_context(
     unsigned32 buf_len
     idl_byte buf[ ]
     sec_login_handle_t *login_context
     error_status_t *status);
```

Parameters

Input

buf_len
The allocated length (in bytes) of the buffer containing the login context.

*buf***[ ]** An idl_byte array containing the importable login context.

Output

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_import_context()** routine imports a context obtained via a call to **sec_login_export_context()** performed on the same machine. To import a login context, users must have the appropriate privileges. Non-privileged users can import only their own login context; privileged users can import the login contexts created by any users.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
The login context itself is not valid.

sec_login_s_default_use
Illegal use of the default login handle occurred.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_export_context(3sec)**.

sec_login_init_first

Purpose

Initializes the default context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_init_first(
     error_status_t *status);
```

Parameters

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_init_first()** routine initializes the default context inheritance mechanism. If the default inheritance mechanism is already initialized, the operation fails. Typically, this routine is called by the initial process at machine boot time to initialize the default context inheritance mechanism for the host machine process hierarchy.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_default_use
The default context is already initialized.

sec_login_s_privileged
An unprivileged process was called in.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_setup_first(3sec)**, **sec_login_validate_first(3sec)**.

sec_login_inquire_net_info

Purpose

Returns a principal's network information

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_inquire_net_info(
     sec_login_handle_t login_context
     sec_login_net_info_t *net_info
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to the login context for the desired principal. (See **sec_intro(3sec)** for more details about the login context.)

Output

net_info
A pointer to the returned **sec_login_net_info_t** data structure that contains the principal's network information. The **sec_login_net_info_t** structure is defined as follows:

```
typedef struct {
   sec_id_pac_t  pac;
   unsigned32    acct_expiration_date;
   unsigned32    passwd_expiration_date;
   unsigned32    identity_expiration_date;
}  sec_login_net_info_t;
};
```

status A pointer to the completion status.

Description

The **sec_login_inquire_net_info()** routine returns network information for the principal identified by the specified login context. The network information consists of the following:

- The privilege attribute certificate (PAC) that describes the identity and group memberships of the principal.
- The expiration date for the principal's account in the DCE registry.
- The expiration date for the principal's password in the DCE registry.
- The lifetime for the principal's authenticated network identity. This is the lifetime of the principal's TGT (see the **sec_login_get_expiration()** routine).

A value of 0 (zero) for an expiration date means there is no expiration date. In other words, the principal's account, password, or authenticated identity is good indefinitely.

To remove the returned *net_info* structure when it is no longer needed, use **sec_login_free_net_info()**.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_not_certified
The login context is not certified.

sec_login_s_context_invalid
The login context is not valid.

sec_login_s_no_current_context
The default context was specified, but none exists.

sec_login_s_auth_local
Operation not valid on local context. The call's identity was not authenticated.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_free_net_info(3sec)**, **sec_login_get_expiration(3sec)**.

sec_login_newgroups

Purpose

Changes the group list for a login context

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_newgroups(
     sec_login_handle_t login_context
     sec_login_flags_t flags
     unsigned32 num_local_groups
     sec_id_t local_groups[ ]
     sec_login_handle_t *restricted_context
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

flags A set of flags of type **sec_login_flags_t**. These contain information about how the new network credentials will be used. Currently, the only flag used is **sec_login_credentials_private**, that, when set, implies that the new context is only to be used by the calling process. If this flag is not set (*flags* **= sec_login_no_flags**), descendants of the calling process may also use the new network credentials.

num_local_groups
An unsigned 32-bit integer containing the number of local group identities to include in the new context.

local_groups[ ]
An array of **sec_id_t** elements. Each element contains the UUID of a local group identity to include in the new context. These identities are local to the cell. Optionally, each element may also contain a pointer to a character string containing the name of the local group.

Output

restricted_context
An opaque handle to the login context containing the changed group list.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_newgroups()** routine changes the group list for the specified login context. Part of a network identity is a list of the various groups to which a principal belongs. The groups are used to determine a user's access to various objects and

services. This routine returns a new login context that contains the changed group list. To remove the new login context when it is no longer needed, use **sec_login_purge_context()**.

This operation does not need to be validated as the user identity does not change. Consequently, knowledge of the password is not needed.

Notes

Currently you can have only groups from the local cell.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Return Values

This routine returns TRUE when the new login context is successfully established.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_auth_local
Operation not valid on local context.

sec_login_s_default_use
It is an error to try to certify the default context.

sec_login_s_groupset_invalid
The input list of group names is invalid. There may be groups to which the caller does not belong, or the list may contain groups that do not exist.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_newgroups()** routine as part of a process to change the groupset:

```
sec_login_get_current_context(&login_context, &st);

sec_login_get_groups(login_context, &num_groups, &groups, &st);

   ...the group IDs have to be converted from the returned UNIX
   numbers into UUIDs (use sec_rgy_pgo_unix_num_to_id(3sec)...

for (i=0; i < num_groups; i++) {
   ... query whether the user wants to discard any of the current
   group memberships.  Copy new groupset to the new_groups array ...
}

if ( !sec_login_newgroups(sec_login_no_flags, num_new_groups,
 new_groups, &login_context, &st)) {
   if (st == sec_login_s_groupset_invalid)
```

```
    printf("New groupset invalid);

      ... application specific error handling ...
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_get_groups(3sec)**, **sec_login_purge_context(3sec)**.

sec_login_purge_context

Purpose

Destroys a login context and frees its storage

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_purge_context(
     sec_login_handle_t *login_context
     error_status_t *status);
```

Parameters

Input

login_context
A pointer to an opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.) Note that a pointer to the handle is submitted, so the handle may be reset to NULL upon successful completion.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_purge_context()** routine frees any storage allocated for the specified login context and destroys the associated network credentials, if any exist.

Cautions

Applications must be cautious when purging the current context as this destroys network credentials for all processes that share the credentials.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_context_invalid
The login context itself is not valid.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_purge_context()** routine as part of a straightforward login process:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags,
            &login_context, &st)) {
   ... get password from user...

   if (sec_login_validate_identity(login_context, password,
                           &reset_passwd, &auth_src, &st)) {

      if (!sec_login_certify_identity(login_context, &st))
          exit(error_wierd_auth_svc);

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      if (reset_passwd) {
         ... get new password from user, reset registry record ...
      };

      sec_login_get_pwent(login_context, &pw_entry, &st);

      if (pw_entry.pw_expire < todays_date) {
         sec_login_purge_context(&login_context, &st);
         exit(0)
      }

      ... any other application specific login valid actions ...
      }

   } else {
      sec_login_purge_context(&login_context, &st);

      ... application specific login failure actions ...
   }
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_set_context(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_refresh_identity

Purpose

Refreshes an authenticated identity for a login context

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_refresh_identity(
     sec_login_handle_t login_context
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_refresh_identity()** routine refreshes a previously established identity. It operates on an existing valid context, and cannot be used to change credentials associated with that identity. The refreshed identity reflects changes that affect ticket lifetimes, but not other changes. For example, the identity will reflect a change to maximum ticket lifetime, but not the addition of the identity as a member to a group. Only a DCE login reflects all administrative changes made since the last login.

The refreshed identity must be validated with **sec_login_validate_identity()** before it can be used.

It is an error to refresh a locally authenticated context.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
The login context itself is not valid.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_no_memory
Not enough memory is available to complete the operation.

error_status_ok
The call was successful.

Examples

Since the authenticated network identity for a process has a finite lifetime, there is a risk it will expire during some long network operation, preventing the operation from completing.

For a server application that must run with an authenticated network identity because they themselves sometimes act as clients of another server, the **sec_login** calls can be used to check the network identity expiration date, run **sec_login_refresh_identity** and **sec_login_validate_identity** before the expiration. This will prevent interruptions in the server's operation due to the restrictions in network access applied to an unauthenticated identity.

```
sec_login_get_expiration(login_context, &expire_time, &st);

if (expire_time < (current_time + operation_duration)) {

   if (!sec_login_refresh_identity(login_context, &st)) {
         ... login context cannot be renewed ...
         ... sleep and try again ....
      }

} else {

     if (sec_login_validate_identity(login_context, password,
                           &reset_passwd, &auth_src, &st)) {
         ... identity validated ...
      } else {
         ... validation failed ...
         exit(error_validation_failure);
      }
   }
}

operation();
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_release_context

Purpose

Frees storage allocated for a login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_release_context(
     sec_login_handle_t *login_context
     error_status_t *status);
```

Parameters

Input/Output

login_context
A pointer to an opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_release_context()** routine frees any memory allocated for a login context. Unlike **sec_login_purge_context()**, it does not destroy the associated network credentials that still reside in the credential cache.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_context_invalid
The login context itself is invalid.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_purge_context(3sec)**.

sec_login_set_context

Purpose

Creates network credentials for a login context

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_set_context(
     sec_login_handle_t login_context
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_set_context()** routine sets the network credentials to those specified by the login context. This context must have been previously validated. Contexts acquired through **sec_login_get_current_context()** or **sec_login_newgroups()** do not need to be validated since those routines return previously validated contexts.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_context_invalid
The login context itself is invalid.

sec_login_s_default_use
Illegal use of the default login handle occurred.

sec_login_s_auth_local
Operation not valid on local context.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_set_context()** routine as part of a straightforward login process:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags,
                          &login_context, &st)) {
   ... get password from user...

   if (sec_login_validate_identity(login_context, password,
                            &reset_passwd, &auth_src, &st)) {

      if (!sec_login_certify_identity(login_context, &st))
          exit(error_weird_auth_svc);

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      if (reset_passwd) {
         ... get new password from user, reset registry record ...
      };

      sec_login_get_pwent(login_context, &pw_entry, &st);

      if (pw_entry.pw_expire < todays_date) {
         sec_login_purge_context(&login_context, &st);
         exit(0)
      }

      ... any other application specific login valid actions ...
      }

   } else {
      sec_login_purge_context(&login_context, &st);

      ... application specific login failure actions ...
   }
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_set_extended_attrs

Purpose

Constructs a new login context that contains extended registry attributes

Synopsis

```
#include <dce/sec_login.h>

sec_login_handle_t
sec_login_set_extended_attrs(
     sec_login_handle_t my_login_context
     unsigned32 num_attributes
     sec_attr_t attributes[ ]
     error_status_t *status);
```

Parameters

Input

my_login_context
A value of **sec_login_handle_t** that provides an opaque handle to the identity of the calling client.

num_attributes
An unsigned 32-bit integer that specifies the number of elements in the *attributes[ ]* array. The number must be greater than 0.

attributes[ ]
An array of values of type **sec_attr_t** that specifies the list of attributes to be set in the new login context.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_set_extended_attrs()** constructs a login context that contains extended registry attributes that have been established for the object identified by *my_login_context*. The attributes themselves must have been established and attached to the object using the extended registry attribute API.

The input *attributes[ ]* array of **sec_attr_t** values should specify the *attr_id* field for each requested attribute. Since the lookup is by attribute type ID only, set the *attribute.attr_value.attr_encoding* field to **sec_attr_enc_void** for each attribute. Note that **sec_attr_t** is an extended registry attribute data type. For more information on extended registry attributes, see the description of the **sec_attr** calls in this document and the *OSF DCE Application Development Guide—Core Components*.

You cannot use this call to add extended registry attributes to a delegation chain. If you pass in a login context that refers to a delegation chain, an invalid context error will be returned.

The routine returns a new login context of type **sec_login_handle_t** that includes the attributes specified in the *attributes[ ]* array.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_invalid_context

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_login_become_impersonator(3sec)**, **sec_login_set_context(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity(3sec)**, **sec_rgy_attr_ *(3sec)** calls.

sec_login_setup_first

Purpose

Sets up the default network context

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_setup_first(
     sec_login_handle_t *init_context
     error_status_t *status);
```

Parameters

Output

init_context
A pointer to an opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. In this call, the context will be that of the host machine initial process. (See **sec_intro(3sec)** for more details about the login context.)

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_setup_first()** routine sets up the default context network identity. If the default context already contains valid credentials, the routine fails. Typically, this routine is called from the security validation service of the **dced** process to breathe life into the default credentials for the host machine process hierarchy.

This routine uses the host name available via the local **dce_config** interface as the principal name for the setup, so it does need a principal name as input.

Return Values

The routine returns a **boolean32** value that is TRUE if the setup was successful, and FALSE otherwise.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_default_use
The default context is already in use and does not need to be set up again.

sec_login_s_no_current_context
The calling process has no context of its own.

sec_login_s_privileged
An unprivileged process was called in.

sec_login_s_config
The DCE configuration (**dce_config**) information is not available.

sec_rgy_object_not_found
The principal does not exist.

sec_rgy_server_unavailable
The network registry is not available.

sec_login_s_no_memory
A memory allocation error occurred.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_init_first(3sec)**, **sec_login_validate_first(3sec)**.

sec_login_setup_identity

Purpose

Sets up the user's network identity

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_setup_identity(
     unsigned_char_p_t principal
     sec_login_flags_t flags
     sec_login_handle_t *login_context
     error_status_t *status);
```

Parameters

Input

principal
A pointer (type **unsigned_char_p_t**) indicating a character string containing the principal name on the registry account corresponding to the calling process.

flags A set of flags of type **sec_login_flags_t**. These contain information about how the new network credentials are to be used.

Output

login_context
A pointer to an opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_setup_identity()** routine creates any local context necessary to perform authenticated network operations. It does not establish any local operating system context; that is the responsibility of the caller. It is the standard network login function. The network identity set up by this operation cannot be used until it is validated via **sec_login_validate_identity()**.

The **sec_login_setup_identity()** operation and the **sec_login_validate_identity()** operation are two halves of a single logical operation. Together they collect the identity data needed to establish an authenticated identity.

Notes

Neither **sec_login_setup_identity()** nor **sec_login_validate_identity()** check for account or identity expiration. The application program using this interface is responsible for such checks.

Return Values

The routine returns TRUE if the identity has been successfully established.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The principal does not exist.

sec_rgy_server_unavailable
The network registry is not available.

sec_login_s_no_memory
Not enough memory is available to complete the operation.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_setup_identity()** routine as part of a straightforward login process:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags,
                          &login_context, &st)) {
   ... get password from user...

   if (sec_login_validate_identity(login_context, password,
                            &reset_passwd, &auth_src, &st)) {

      if (!sec_login_certify_identity(login_context, &st))
          exit(error_weird_auth_svc);

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      if (reset_passwd) {
         ... get new password from user, reset registry record ...
      };

      sec_login_get_pwent(login_context, &pw_entry, &st);

      if (pw_entry.pw_expire < todays_date) {
         sec_login_purge_context(&login_context, &st);
         exit(0)
      }

      ... any other application specific login valid actions ...
      }

   } else {
```

```
        sec_login_purge_context(&login_context, &st);

        ... application specific login failure actions ...
    }
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_set_context(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_valid_and_cert_ident

Purpose

Validates and certifies a login context

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_valid_and_cert_ident(
     sec_login_handle_t login_context
     sec_passwd_rec_t *passwd
     boolean32 *reset_passwd
     sec_login_auth_src_t *auth_src
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

passwd
A password record to be checked against the password in the principal's registry account. The routine returns TRUE if the two match. The contents of the *passwd* parameter are erased after the call has finished processing it.

Output

reset_passwd
A pointer to a 32-bit **boolean32** value. The routine returns TRUE if the account password has expired and must be reset.

auth_src
A 32-bit set of flags identifying the source of the authentication. Upon return after successful authentication, the flags in *auth_src* indicate what authority was used to validate the login context. If the authentication was accomplished with the network authority, the **sec_login_auth_src_network** flag is set, and the process login context has credentials to use the network.

If the authentication was accomplished with local data only (either the principal's account is tailored for the local machine with overrides, or the network authority is unavailable), the **sec_login_auth_src_local** flag is set. Login contexts that are authenticated locally may not be used to establish network credentials because they have none.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_valid_and_cert_ident()** routine validates and certifies a login context established with **sec_login_setup_identity()**. The caller must supply the user's password as input with the *passwd* parameter.

This routine combines the operations of the **sec_login_validate_identity()** and **sec_login_certify_identity()** routines. It is intended for use by system login programs that need to extract trustworthy operating system credentials for use in setting the local identity for a process. This operation destroys the contents of the *passwd* input parameter.

If the network security service is unavailable or if the user's password has been overridden on the host, a locally authenticated context is created, and the *auth_src* parameter is set to **sec_login_auth_src_local**. Data extracted from a locally authenticated context may be used to set the local OS identity, but it cannot be used to establish network credentials.

This routine is a privileged operation.

Return Values

The routine returns TRUE if the login identity has been successfully validated.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_passwd_invalid
The input string does not match the account password.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

sec_login_s_acct_invalid
The account is invalid or has expired.

sec_login_s_privileged
This is a privileged operation and was invoked by an unprivileged process.

sec_login_s_null_password
The input string is NULL.

sec_login_s_default_use
The input context was the default context, which cannot be validated.

sec_login_s_already_valid
The login context has already been validated.

sec_login_s_unsupp_passwd_type
The password type is not supported.

sec_login_s_no_memory
Not enough memory is available to complete the operation.

sec_login_s_preauth_failed
Preauthentication failure.

sec_pk_e_domain_unsupported
The DCE login domain is not supported by the personal security mechanism.

sec_pk_e_device_error
Personal security mechanism device error.

sec_pk_e_usage_unsupported
A private key of the required type was not located in the personal security mechanism.

sec_pk_e_unauthorized
The password is invalid for personal security mechanism access.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_valid_and_cert_ident()** routine as part of a system login process:

```
if (sec_login_setup_identity(<user>,
     sec_login_no_flags, &login_context, &st)) {
         ... get password ...
         if (sec_login_valid_and_cert_ident(login_context,
              password, &st)) {
               if (auth_src == sec_login_auth_src_network) {
                  if (GOOD_STATUS(&st)
                  sec_login_set_context(login_context);
                  }
         }
         if (reset_passwd) {
            ... reset the user's password ...
            if (passwd_reset_fails) {
                sec_login_purge_context(login_context)
                ... application login failure actions ...
         }
         ... application specific login valid actions ...
     }
}
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_certify_identity(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_valid_from_keytable

Purpose

Validates a login context's identity using input from a specified keytable file

Synopsis

```
#include <dce/sec_login.h>

void  sec_login_valid_from_keytable(
     sec_login_handle_t login_context
     unsigned32 authn_service
     void *arg
     unsigned32 try_kvno
     unsigned32 *used_kvno
     boolean32 *reset_passwd
     sec_login_auth_src_t *auth_src
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal's name and UUID, account restrictions, records of the account principal's group memberships, and the account's home directory. (See **sec_intro(3sec)** for more details about the login context.)

authn_service
Identifies the authentication protocol using the key. The possible authentication protocols are as follows:

rpc_c_authn_dce_secret
DCE shared-secret key authentication.

rpc_c_authn_dce_public
DCE public key authentication (reserved for future use).

arg This parameter can specify either the local keytab file or an argument to the *get_key_fn* key acquisition routine of the **rpc_server_register_auth_info** routine.

A value of NULL specifies that the default keytab file should be used. A keytab filename specifies that that file should be used as the keytab file. You must prepend the file's absolute filename with **FILE:** and the file must have been created with the **rgy_edit** command or the **sec_key_mgmt_set_key** routine.

Any other value specifies an argument for the *get_key_fn* key acquisition routine. See the **rpc_server_register_auth_info()** reference page for more information.

try_kvno
The version number of the key in the keytab file to try first. Specify NULL to try the current version of the key.

Output

used_kvno
A pointer to a 32-bit **boolean32** value that specifies the version number of the the key from the keytab file that was used to successfully validate the login context, if any.

reset_passwd
A pointer to a 32-bit **boolean32** value. The routine returns TRUE if the account password has expired and should be reset.

auth_src
How the the login context was authorized. The **sec_login_auth_src_t** data type distinguishes the various ways the login context was authorized. There are three possible values:

sec_login_auth_src_network
Authentication accomplished through the normal network authority. A login context authenticated this way will have all the network credentials it ought to have.

sec_login_auth_src_local
Authentication accomplished via local data. Authentication occurs locally if a principal's account is tailored for the local machine, or if the network authority is unavailable. Since a login contexts authenticated locally has no network credentials, it can not be used for network operations.

sec_login_auth_src_overridden
Authentication accomplished via the override facility.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_valid_from_keytable ()** routine validates the login context established with **sec_login_setup_identity()**. The **sec_login_valid_from_keytable ()** routine obtains the principal's password from the specified keytable.

If *try_kvno* specifies a key version number, that version number key is tried first, otherwise the current key version number is tried first. The function trys all keys in the keytable until it finds one that validates the login context. This operation must be invoked before the network credentials can be used.

Notes

A context is not secure and must not be set or exported until the authentication service is itself authenticated with the **sec_login_certify_identity()** call.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_passwd_invalid
The input string does not match the account password.

sec_rgy_server_unavailable
There is no data with which to compare the input string.

sec_login_s_acct_invalid
The account is invalid or has expired.

sec_login_s_default_use
The input context was the default context, which cannot be validated.

sec_login_s_already_valid
The login context has already been validated.

sec_login_s_unsupp_passwd_type
The password type is not supported.

sec_key_mgmt_e_key_unavailable
The requested key is not present.

sec_key_mgmt_e_authn_invalid
The authentication protocol is not valid.

sec_key_mgmt_e_unauthorized
The caller is not authorized to perform the operation.

sec_s_no_memory
Unable to allocate memory.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_valid_from_keytable()** routine as part of a straightforward login process:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags,
                          &login_context, &st)) {
   ... get password from local keytable...

   if (sec_login_valid_from_keytable(login_context, authn_service,
       arg, try_kvno, &used_kvno, &reset_passwd,
       &auth_src, &st)) {

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      }

      ... any other application specific login valid actions ...
      }

   } else {
      sec_login_purge_context(&login_context, &st);
```

```
            ... application specific login failure actions ...
        }
    }
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_certify_identity(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_valid_and_cert_ident(3sec)**, **sec_login_validate_identity(3sec)**.

sec_login_validate_first

Purpose

Validates the initial login context

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_validate_first(
     sec_login_handle_t init_context
     boolean32 *reset_passwd
     sec_login_auth_src_t *auth_src
     error_status_t *status);
```

Parameters

Input

init_context
: An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. In this call, the context will be that of the host machine initial process. (See **sec_intro(3sec)** for more details about the login context.)

Output

reset_passwd
: A pointer to a 32-bit **boolean32** value. The routine returns TRUE if the account password has expired and must be reset.

auth_src
: A 32-bit set of flags identifying the source of the authentication. Upon return after successful authentication, the flags in *auth_src* indicate what authority was used to validate the login context. If the authentication was accomplished with the network authority, the **sec_login_auth_src_network** flag is set, and the process login context has credentials to use the network. If the authentication was accomplished with local data only (either the principal's account is tailored for the local machine with overrides, or the network authority is unavailable), the **sec_login_auth_src_local** flag is set. Login contexts that are authenticated locally may not be used to establish network credentials because they have none.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_validate_first()** routine validates the default login context established via **sec_login_setup_first()**. Typically, this operation is called from the security validation service of the **dced** process to validate the default credentials for the host machine process hierarchy. This operation uses the password for the local host, and therefore does not require a password parameter.

Return Values

The routine returns a **boolean32** value that is TRUE if the setup was successful, and FALSE otherwise.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_privileged
An unprivileged process was called in.

sec_rgy_server_unavailable
The network authentication service was unavailable.

sec_pk_e_domain_unsupported
The DCE login domain is not supported by the personal security mechanism.

sec_pk_e_device_error
Personal security mechanism device error.

sec_pk_e_usage_unsupported
A private key of the required type was not located in the personal security mechanism.

sec_pk_e_unauthorized
The password is invalid for personal security mechanism access.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_login_init_first(3sec)**, **sec_login_setup_first(3sec)**.

sec_login_validate_identity

Purpose

Validates a login context's identity

Synopsis

```
#include <dce/sec_login.h>

boolean32  sec_login_validate_identity(
     sec_login_handle_t login_context
     sec_passwd_rec_t *passwd
     boolean32 *reset_passwd
     sec_login_auth_src_t *auth_src
     error_status_t *status);
```

Parameters

Input

login_context
An opaque handle to login context data. The login context contains, among other data, the account principal name and UUID, account restrictions, records of group membership, and the process home directory. (See **sec_intro(3sec)** for more details about the login context.)

passwd
A password record to be checked against the password in the principal's registry account. The routine returns TRUE if the two match. The contents of the *passwd* parameter are erased after the call has finished processing it.

Output

reset_passwd
A pointer to a 32-bit **boolean32** value. The routine returns TRUE if the account password has expired and must be reset.

auth_src
How the the login context was authorized. The **sec_login_auth_src_t** data type distinguishes the various ways the login context was authorized. There are three possible values:

- **sec_login_auth_src_network**
- **sec_login_auth_src_local**
- **sec_login_auth_src_overridden**

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_login_validate_identity()** routine validates the login context established with **sec_login_setup_identity()**. This operation must be invoked before the network credentials can be used. The caller must supply the principal's password in a **sec_passwd_rec_t** as input with the *passwd* parameter. The following example sets up a plaintext password for the the *passwd* parameter:

```
sec_passwd_str_t      tmp_passwd;

passwd.version_number = sec_passwd_c_version_none;
passwd.pepper = NULL;
passwd.key.key_type = sec_passwd_plain;

strncpy((char *) tmp_passwd, (char *) my_passwd,
        sec_passwd_str_max_len);
tmp_passwd[sec_passwd_str_max_len] = ' ';
passwd_rec.key.tagged_union.plain = &(tmp_passwd[0]);
```

When a network identity is set, only state information for network operations has been established. The local operating system identity has not been modified. It is the responsibility of the caller to establish any local operating identity state.

The **sec_login_setup_identity()** operation and the **sec_login_validate_identity()** operation are two halves of a single logical operation. Together they collect the identity data needed to establish an authenticated identity. The operations are independent so the principal's password need not be sent across the network. The identity validation performed by **sec_login_validate_identity()** is a local operation.

Notes

A context is not secure and must not be set or exported until the authentication service is itself authenticated with the **sec_login_certify_identity()** call.

System login programs that set local operating system identity using data extracted from a login context should use **sec_login_valid_and_cert_ident()** instead of **sec_login_validate_identity()**.

If the security server and client clocks are not synchronized to within 2 to 3 minutes of each other, this call can return a password validation error.

Return Values

The routine returns TRUE if the login identity has been successfully validated.

Files

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_passwd_invalid
The input string does not match the account password.

sec_rgy_server_unavailable
There is no data with which to compare the input string.

sec_login_s_acct_invalid
The account is invalid or has expired.

sec_login_s_null_password
The input string is NULL.

sec_login_s_default_use
The input context was the default context, which cannot be validated.

sec_login_s_already_valid
The login context has already been validated.

sec_login_s_unsupp_passwd_type
The password type is not supported.

sec_login_s_no_memory
Not enough memory is available to complete the operation.

sec_login_s_preauth_failed
Preauthentication failure.

sec_pk_e_domain_unsupported
The DCE login domain is not supported by the personal security mechanism.

sec_pk_e_device_error
Personal security mechanism device error.

sec_pk_e_usage_unsupported
A private key of the required type was not located in the personal security mechanism.

sec_pk_e_unauthorized
The password is invalid for personal security mechanism access.

error_status_ok
The call was successful.

Examples

The following example illustrates use of the **sec_login_validate_identity()** routine as part of a straightforward login process:

```
if (sec_login_setup_identity(user_name, sec_login_no_flags,
                          &login_context, &st)) {
   ... get password from user...

   if (sec_login_validate_identity(login_context, password,
                            &reset_passwd, &auth_src, &st)) {

      if (!sec_login_certify_identity(login_context, &st))
          exit(error_weird_auth_svc);

      sec_login_set_context(login_context, &st);

      if (auth_src != sec_login_auth_src_network)
         printf("no network credentials");

      if (reset_passwd) {
         ... get new password from user, reset registry record ...
      };

      sec_login_get_pwent(login_context, &pw_entry, &st);

      if (pw_entry.pw_expire < todays_date) {
         sec_login_purge_context(&login_context, &st);
         exit(0)
      }
```

```
            ... any other application specific login valid actions ...
            }

         } else {
            sec_login_purge_context(&login_context, &st);

            ... application specific login failure actions ...
         }
      }
```

Related Information

Functions: **sec_intro(3sec)**, **sec_login_certify_identity(3sec)**, **sec_login_setup_identity(3sec)**, **sec_login_valid_and_cert_ident(3sec)**.

sec_pk_data_free

Purpose

Frees memory allocated to a **sec_pk_data_t** and its aliases. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <sec_pk_base.h>

void  sec_pk_data_free(
     sec_pk_data_t  *data_p);
```

Parameters

Input/Output

data_p
: As input, a pointer to a **sec_pk_data_t** that points to the memory to be reclaimed. As output, a pointer to a **sec_pk_data_t** that is set to NULL.

Description

The **sec_pk_data_free()** routine frees and sets to NULL each nonnull pointer in a **sec_pk_data_t**. Use this function, rather than **sec_pk_data_zero_and_free()**, for **sec_pk_data_t** structures that contain a public key pair and other nonsensitive data.

Files

/usr/include/dce/sec_pk_base.idl
: The **idl** file from which **dce/sec_pk_base.h** was derived.

Related Information

Functions: **sec_pk_data_zero_and_free(3sec)**.

sec_pk_data_zero_and_free

Purpose

Frees and zeros out memory allocated to a **sec_pk_data_t** or its aliases. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <sec_pk_base.h>

void  sec_pk_data_zero_and_free(
     sec_pk_data_t  *data_p);
```

Parameters

Input/Output

data_p
: As input, a pointer to a **sec_pk_data_t** that points to the memory to be reclaimed. As output, a pointer to a **sec_pk_data_t** that is set to NULL.

Description

The **sec_pk_data_zero_and_free()** routine zeros out and frees memory allocated to a **sec_pk_data_t** or its aliases. Use this function, rather than **sec_pk_data_free()**, for structures that contain the private part of a public key pair or secret keys.

Files

/usr/include/dce/sec_pk_base.idl
: The **idl** file from which **dce/sec_pk_base.h** was derived.

Related Information

Functions: **sec_pk_data_free(3sec)**.

sec_psm_close

Purpose

Close a personal security mechanism. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_close(
     sec_psm_handle_t psm_handle
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to the personal security context data. Use the **sec_psm_open()** routine to obtain the handle.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_close()** routine closes the personal security mechanism identified by *psm_handle*. In addition, the routine cleans up the personal security context data and ensures any confidential information (such as passwords or private keys) is zeroed out.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_t

sec_psm_not_init

sec_psm_invalid_handle

sec_psm_internal_error

sec_pvtkey_invalid_handle

sec_pvtkey_mechanism_not_init

Related Information

Functions: **sec_psm_open(3sec)**.

sec_psm_decrypt_data

Purpose

Decrypt data that was encrypted with a public key mechanism. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_decrypt_data(
     sec_psm_handle_t psm_handle
     unsigned32 *kvno
     sec_pk_algorithm_id_t *encryption_alg_id
     sec_pk_usage_flags_t key_usage
     sec_pk_encrypted_t *cipher_data
     sec_pk__gen_data_t *clear_data
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to the personal security mechanism context. Use **sec_psm_open()** to obtain the handle.

kvno The unsigned 32-bit version number of the key in which the data is encrypted.

encryption_alg_id
The ASN.1 DER-encoded object ID of the encryption algorithm, such as RSA, used to encrypt the data.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag for the private key to be used for this operation.

cipher_data
A pointer to a **sec_pk_data_t** that contains the ASN.1 DER-encoded data to be decrypted.

Output

clear_data
A pointer to the decrypted data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_decrypt_data()** routine decrypts data that was encrypted with a public key mechanism. The *encryption_alg_id* parameter specifies the encryption algorithm used. This routine allocates memory for the returned decrypted data. Call the **sec_pk_data_free()** routine to deallocate that memory.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_psm_unsupported_algorithm_id

sec_bsafe_encryption_failure

sec_pvtkey_invalid_handle

sec_pvtkey_mechanism_not_init

sec_pvtkey_internal_error

sec_pvtkey_invalid_password

sec_pvtkey_multiple_key_usage

Related Information

Functions: **sec_pk_data_free(3sec)**, **sec_psm_encrypt_data(3sec)**.

sec_psm_encrypt_data

Purpose

Encrypt data using a public key mechanism. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_encrypt_data(
     sec_psm_handle_t psm_handle
     sec_pk domain_t *encryptee_domain
     void *encryptee_name
     unsigned32 *kvno
     sec_pk_algorithm_id_t *encryption_alg_id
     sec_pk_usage_flags_t key_usage
     sec_pk_gen_data_t *clear_data
     sec_pk_encrypted_t **cipher_data
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to the personal security context data. Use **sec_psm_open()** to obtain the handle.

encryptee_domain
A pointer to the application domain of the principal for which the data is encrypted.

encryptee_name
A pointer to the name of the principal for which the data is encrypted.

encryption_alg_id
The ASN.1 DER-encoded object ID of the encryption algorithm to use, such as RSA.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag for the public key specified by *data*.

clear_data
A pointer to ASN.1 DER-encoded data to be encrypted.

Input/Output

kvno As input, the unsigned 32-bit version number of the key with which to encrypt the data. As output, the unsigned 32-bit version number of the key used to encrypt the data.

Output

cipher_data
A pointer to the encrypted data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_encrypt_data()** routine encrypts data by using a public key encryption mechanism. The *encryption_alg_id* parameter specifies the encryption algorithm. This routine allocates memory for *cipher_data*. Call the **sec_pk_data_free()** routine to deallocate that memory.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_psm_unsupported_algorithm_id

sec_pk_e_domain_unsupported

sec_pk_e_usage_unsupported

sec_rgy_object_not_found

sec_rgy_not_authorized

sec_attr_unsupported

Related Information

Functions: **sec_pk_data_free(3sec)**, **sec_psm_decrypt_data(3sec)**.

sec_psm_gen_pub_key

Purpose

Randomly generate a public key pair. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_gen_pub_key(
     sec_pk_algorithm_id_t *key_type
     unsigned32 modulus_bit_size
     sec_pk_gen_data_t *seed
     sec_pk_data_t *public_key
     sec_pk_data_t *private_key
     error_status_t *status);
```

Parameters

Input

key_type
A pointer to the object ID of the public key encryption algorithm to use. Only the RSA public key algorithm (RSA_PKCS) is currently supported.

modulus_bit_size
The desired length of the key. Interpretation of this parameter is dependent on the algorithm specified by *key_type*. For RSA, the only currently supported key type, *modulus_bit_size* is a number ranging from 256 through 1024 inclusive that specifies the bit length of the key modulus. A value of **0** indicates the default of 1024.

seed A pointer to the string to seed the random key generator.

Output

private_key
A pointer to a **sec_pk_data_t** that contains the private key structure of the newly generated key.

public_key
A pointer to a **sec_pk_data_t** that contains the public key structure of the newly generated key.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_gen_pub_key()** routine generates a public key pair. This routine allocates memory for the returned key. Call the **sec_pk_data_free()** routine to deallocate the public key and **sec_pk_data_zero_and_free()** routine to deallocate the private key.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_wrong_pub_key_type

sec_bsafe_alloc

Related Information

Functions: **sec_psm_update_pub_key(3sec)**, **sec_psm_put_pub_key(3sec)**.

sec_psm_open

Purpose

Open a personal security mechanism. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_open(
     void *name
     char *pwd
     sec_pk_domain_t *domain_id
     sec_psm_handle_t *psm_handle
     error_status_t *status);
```

Parameters

Input

name A pointer to the name of the principal for which to open the personal security mechanism. Supply this name in the form **/.:/principal_name** or **/.../cell_name/principal_name/**.

pwd A pointer to the principal's password.

domain_id
A pointer to the application domain that the principal is operating on. (Currently, the only domain supported is **sec_pk_domain_dce_pk_login**.)

Output

psm_handle
A pointer to an opaque handle to the personal security context data.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_open()** routine obtains a handle to a personal security mechanism for the principal specified by *name* by using the password specified with *pwd*.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_pvtkey_privileged

sec_pvtkey_no_more_memory

sec_psm_no_more_memory

Related Information

Functions: **sec_psm_close(3sec)**.

sec_psm_put_pub_key

Purpose

Store a public key pair. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_put_pub_key(
     sec_psm_handle_t psm_handle
     char *pwd
     sec_pk_usage_flags_t key_usage
     sec_pk_pvtkey_t *pvtkey
     sec_pk_pvtkey_t *pubkey
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to the personal security mechanism in which to store the key. Use **sec_psm_open()** to obtain the handle.

pwd A pointer to the password for the principal associated with the personal security mechanism.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag for the public key for the key pair specified by *pubkey*.

pvtkey A pointer to the ASN.1 DER-encoded private key.

pubkey
A pointer to the ASN.1 DER-encoded public key.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_put_pub_key()** routine stores a public key pair. In the reference implementation, the public key is stored in the registry and the private key in a personal security mechanism. Key versions are not currently supported; only a single version of a key with a given key usage is maintained.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_pk_e_domain_unsupported

sec_pk_e_usage_unsupported

sec_rgy_object_not_found

sec_rgy_not_authorized

sec_attr_unauthorized

sec_pvtkey_invalid_handle

sec_pvtkey_mechanism_not_init

sec_pvtkey_no_more_memory

sec_pvtkey_internal_error

sec_pvtkey_same_domain_and_usage_key_already_exists

Related Information

Functions: **sec_psm_gen_pub_key(3sec)**, **sec_psm_update_pub_key(3sec)**.

sec_psm_sign_data

Purpose

Compute the signature of data using a specified signature algorithm. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_sign_data(
     sec_psm_handle_t psm_handle
     sec_pk_algorithm_id_t *signature_alg_id
     sec_pk_usage_flags_t key_usage
     sec_pk_gen_data_t *data
     unsigned32 *kvno
     sec_pk_signed_t *signature
     error_status_t *status_t);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to the personal security context data. Use **sec_psm_open()** to obtain the handle.

signature_alg_id
The ASN.1 DER-encoded object ID of the signature algorithm. **MD5WithRSAEncryption** is the only algorithm ID currently supported.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag of the private key to be used in this operation.

data A pointer to the ASN.1 DER-encoded data to be signed.

Output

kvno The version of the key being used.

signature
A pointer to the computed signature.

status_t
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_sign_data()** routine computes the signature of input data by using the signature algorithm specified by *signature_alg_id*. This routine allocates memory for the returned signed data. Call the **sec_pk_data_free()** routine to deallocate that memory.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_psm_unsupported_algorithm_id

sec_pvtkey_invalid_handle

sec_pvtkey_mechanism_not_init

sec_pvtkey_internal_error

sec_pvtkey_invalid_password

sec_pvtkey_multiple_key_usages

Related Information

Functions: **sec_pk_data_free(3sec)**, **sec_psm_verify_data(3sec)**.

sec_psm_update_pub_key

Purpose

Update a public key in a personal security mechanism.. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_update_pub_key(
     sec_psm_handle_t psm_handle
     char *oldpwd
     char *newpwd
     sec_pk_usage_flags_t key_usage
     sec_pk_pubkey_t *pubkey
     sec_pk_pvtkey_t *pvtkey
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to a personal security mechanism context data. Use **sec_psm_open()** to obtain the handle.

oldpwd
A pointer to the principal's current password.

newpwd
A pointer to the principal's new password.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag for the public key in the key pair.

pubkey
A pointer to the ASN.1 DER-encoded public key.

pvtkey A pointer to the ASN.1 DER-encoded private key.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_update_pub_key()** routine updates a principal's public key pair or password. The current public key password must be supplied for authentication. Currently, only a single version of a key with a given key usage is maintained. Therefore, any old key versions are overwritten by this routine. Note that there is no routine supplied to delete keys; deletion is assumed to be an internal function initiated by the personal security mechanism.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_pk_e_domain_unsupported

sec_pk_e_usage_unsupported

sec_rgy_object_not_found

sec_rgy_not_authorized

sec_attr_unauthorized

sec_pvtkey_invalid_handle

sec_pvtkey_mechanism_not_init

sec_pvtkey_private_key_is_not_supplied

sec_pvtkey_new_password_required

sec_pvtkey_no_more_memory

Related Information

Functions: **sec_psm_gen_pub_key(3sec)**, **sec_psm_put_pub_key(3sec)**.

sec_psm_verify_data

Purpose

Verify signed data. This routine is not available in the DCE binary code. It is provided in DCE source for use by vendors.

Synopsis

```
#include <dce/sec_pk_base.h>

error_status_t  sec_psm_verify_data(
     sec_psm_handle_t psm_handle
     sec_pk_domain_t *signer_domain_id
     void *signer_name
     unsigned32 *kvno
     sec_pk_algorithm_id_t *signature_alg_id
     sec_pk_usage_flags_t key_usage
     sec_pk_gen_data_t *data
     sec_pk_signed_t *signature
     error_status_t *status);
```

Parameters

Input

psm_handle
A pointer to an opaque handle to personal security context data. Use **sec_psm_open()** to obtain the handle.

signer_domain_id
A pointer to the application domain of the principal that signed the data.

signer_name
A pointer to the name of the principal that signed the data.

kvno The version of the key being used.

signature_alg_id
The ASN.1 DER-encoded object ID of the signature algorithm, such as **MD5WithRSAEncryption**.

key_usage
A **sec_pk_usage_flags_t** that contains the usage flag for the public key.

data A pointer to the data to be verified.

signature
A pointer to the signature to be verified.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_psm_sign_data()** routine verifies input data, usually the data signature of input data.

Files

/usr/include/dce/sec_pk_base.idl
The **idl** file from which **dce/sec_pk_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

sec_psm_not_init

sec_psm_invalid_handle

sec_psm_unsupported_algorithm_id

sec_pk_e_domain_unsupported

sec_rgy_object_not_found

sec_rgy_not_authorized

sec_attr_unauthorized

Related Information

Functions: **sec_psm_sign_data(3sec)**.

sec_pwd_mgmt_free_handle

Purpose

Frees storage allocated for a password management handle

Synopsis

```
#include <dce/sec_pwd_mgmt.h>

void  sec_pwd_mgmt_free_handle(
     sec_pwd_mgmt_handle_t *pwd_mgmt_h
     error_status_t *stp);
```

Parameters

Input/Output

pwd_mgmt_h
A handle to the password management data which is to be freed.

Output

stp A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns an error.

Description

The **sec_pwd_mgmt_free_handle()** routine frees any memory allocated for the contents of a password management handle.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The idl file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful

Related Information

Functions: **sec_intro(3sec)**, **sec_pwd_mgmt_setup(3sec)**.

sec_pwd_mgmt_gen_pwd

Purpose

Generates a set of passwords

Synopsis

```
#include <dce/sec_pwd_mgmt.h>

void  sec_pwd_mgmt_gen_pwd(
     sec_pwd_mgmt_handle_t pwd_mgmt_h
     unsigned32 num_pwds
     unsigned32 *num_returned
     sec_passwd_rec_t gen_pwds[ ]
     error_status_t *stp);
```

Parameters

Input

pwd_mgmt_h
A handle to user's password management data.

num_pwds
Number of generated passwords requested.

Output

num_returned
Number of generated passwords returned in the *gen_pwds*[ ] array.

*gen_pwds***[ ]**
Array of generated passwords. Each generated password is stored in a **sec_passwd_rec_t** structure.

stp A pointer to the completion status. On successful completion, status is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_pwd_mgmt_gen_pwd()** routine retrieves a set of generated passwords from a password management server which is exporting the **rsec_pwd_mgmt_gen_pwd()** routine. It obtains the binding information to this server from the *pwd_mgmt_h* handle.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The idl file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_era_pwd_mgmt_auth_type
The pwd_mgmt_binding ERA must contain authentication information.

sec_pwd_mgmt_svr_unavail
The password management server is unavailable.

sec_pwd_mgmt_svr_error
Generic error returned from password management server. An administrator should check the password management server's log file for more information.

error_status_ok
The call was successful

Various RPC communication errors can be returned if there are failures when binding to the password management server.

Related Information

Functions: **pwd_strengthd(8sec)**, **sec_intro(3sec)**, **sec_pwd_mgmt_setup(3sec)**.

sec_pwd_mgmt_get_val_type

Purpose

Gets users password validation type

Synopsis

```
#include <dce/sec_pwd_mgmt.h>

void  sec_pwd_mgmt_get_val_type(
     sec_pwd_mgmt_handle_t pwd_mgmt_h
     signed32 *pwd_val_type
     error_status_t *stp);
```

Parameters

Input

pwd_mgmt_h
A handle to a user's password management data.

Output

pwd_val_type
The user's password validation type. This is retrieved from the *pwd_val_type* ERA. The possible values and their meaning are as follows:

0 (**none**): the user has no password policy.

1 (**user_select**): the user must choose his/her own password.

2 (**user_can_select**): the user can choose his/her own password or request a generated password.

3 (**generation_required**): the user must use a generated password.

stp A pointer to the completion status. On successful completion, stp is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_pwd_mgmt_get_val_type()** routine returns the value of the user's password validation type, as specified by the *pwd_val_type* ERA. If the ERA does not exist, **0** (**none**) is returned in *pwd_val_type*.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The idl file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Various RPC communication errors can be returned if there are failures when binding to the password management server.

Related Information

Functions: **sec_intro(3sec)**, **sec_pwd_mgmt_setup(3sec)**.

sec_pwd_mgmt_setup

Purpose

Sets up the user's password policy information

Synopsis

```
#include <dce/sec_pwd_mgmt.h>

void  sec_pwd_mgmt_setup(
     sec_pwd_mgmt_handle_t *pwd_mgmt_h
     sec_rgy_handle_t context
     sec_rgy_login_name_t login_name
     sec_login_handle_t your_lc
     rpc_binding_handle_t pwd_mgmt_bind_h
     error_status_t *stp);
```

Parameters

Input

context
A registry server handle indicating the desired registry site.

login_name
The login name of the user.

your_lc
The login context handle of the user currently logged in. If null is specified, the default login context will be used.

pwd_mgmt_bind_h
An RPC binding handle to the password management server. Use of this parameter is currently unsupported. The password management server binding handle will be retrieved from the **pwd_mgmt_binding** ERA. Set this parameter to NULL.

Output

pwd_mgmt_h
A pointer to an opaque handle to password management/policy data. *pwd_mgmt_h* contains, among other data, the account name, values of password management ERAs, and a binding handle to the password management server.

stp A pointer to the completion status. On successful completion, *stp* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_pwd_mgmt_setup()** routine collects the data required to perform remote password management calls to the password management server.

Files

/usr/include/dce/sec_pwd_mgmt.idl
The idl file from which **dce/sec_pwd_mgmt.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_s_no_memory
Not enough memory is available to complete the operation.

sec_rgy_server_unavailable
The network registry is not available.

error_status_ok
The call was successful.

Related Information

Functions: **pwd_strengthd(8sec)**, **sec_intro(3sec)**, **sec_pwd_mgmt_free_handle(3sec)**, **sec_pwd_mgmt_gen_pwd(3sec)**, **sec_pwd_mgmt_get_val_type(3sec)**.

sec_rgy_acct_add

Purpose

Adds an account for a login name

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_add(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_key_t *key_parts
     sec_rgy_acct_user_t *user_part
     sec_rgy_acct_admin_t *admin_part
     sec_passwd_rec_t *caller_key
     sec_passwd_rec_t *new_key
     sec_passwd_type_t new_keytype
     sec_passwd_version_t *new_key_version
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
: A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three names must be completely specified.

key_parts
: A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

user_part
: A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell.

admin_part
: A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information.

caller_key
: The key representing the user's current password, used to encrypt *new_key* for transmission to the registry server.

new_key
: The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode *new_key* if it is sent as plaintext.

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the *version_number* field of the *new_key* input parameter), the server returns the requested version number back to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_add()** routine adds an account with the specified login name. The login name is given in three parts, corresponding to the principal, group, and organization names for the account. All input paramters and all fields in those parameters are required.

The *key_parts* variable specifies the minimum login abbreviation for the account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine supplies the next shortest unique abbreviation and returns this abbreviation in *key_parts*. Abbreviations are not currently implemented.

Permissions Required

The **sec_rgy_acct_add()** routine requires the following permissions on the account (principal) that is to be added:

- The **m** (**mgmt_info**) permission to change management information.
- The **a** (*auth_info*) permission to change authentication information.
- The **u** (**user_info**) permission to change user information.

Notes

The constituent principal, group, and organization (PGO) items for an account must be added before the account can be created. (See the **sec_rgy_pgo_add()** routine). Also, the principal must have been added as a member of the specified group and organization. (See the **sec_rgy_pgo_add_member()** routine).

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to add an account to the registry.

sec_rgy_not_member_group
The indicated principal is not a member of the indicated group.

sec_rgy_not_member_org
The indicated principal is not a member of the indicated organization.

sec_rgy_not_member_group_org
The indicated principal is not a member of the indicated group or organization.

sec_rgy_object exists
The account to be added already exists.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_delete(3sec)**, **sec_rgy_login_get_info(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_add_member(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_acct_admin_replace

Purpose

Replaces administrative account data

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_admin_replace(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_key_t *key_parts
     sec_rgy_acct_admin_t *admin_part
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

admin_part
A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator. The **sec_rgy_acct_admin_t** structure contains the following fields:

creator
The identity of the principal who created this account in **sec_rgy_foreign_id_t** form. This field is set by the registry server.

creation_date
The date (**sec_timeval_sec_t**) the account was created. This field is set by the registry server.

last_changer
The identity of the principal who last modified any of the account information (user or administrative). This field is set by the registry server.

change_date
The date (**sec_timeval_sec_t**) the account was last modified (either user or administrative data). This field is set by the registry server.

expiration_date
The date (**sec_timeval_sec_t**) the account will cease to be valid.

good_since_date
This date (**sec_timeval_sec_t**) is for Kerberos-style, ticket-granting ticket revocation. Ticket-granting tickets issued before this date will not be honored by authenticated network services.

flags Contains administration flags used as part of the administrator's information for any registry account. This field is in **sec_rgy_acct_admin_flags_t** form. (See **sec_intro(3sec)** for a complete description of these flags.)

authentication_flags
Contains flags controlling use of authentication services. This field is in **sec_rgy_acct_auth_flags_t** form. (See **sec_intro(3sec)** for a complete description of these flags.)

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_admin_replace()** routine replaces the administrative information in the account record specified by the input login name. The administrative information contains limitations on the account's use and privileges. It can be modified only by a registry administrator; that is, a user with the **admin_info** (abbreviated as **a**) privilege for an account.

The *key_parts* variable identifies how many of the *login_name* parts to use as the unique abbreviation for the account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine supplies the next shortest unique abbreviation and returns this abbreviation using *key_parts*.

Permissions Required

The **sec_rgy_acct_admin_replace()** routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission, if **flags** or **expiration_date** is to be changed.
- The **a** (*auth_info*) permission, if **authentication_flags** or **good_since_date** is to be changed.

Notes

All users need the **w** (**write**) privilege in the appropriate ACL entry to modify any account information.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to change the administrative information for the specified account.

sec_rgy_object_not_found
The registry server could not find the specified name.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_lookup(3sec)**, **sec_rgy_acct_replace_all(3sec)**, **sec_rgy_acct_user_replace(3sec)**.

sec_rgy_acct_delete

Purpose

Deletes an account

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_delete(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. Only the principal name is required to perform the deletion.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_delete()** routine deletes from the registry the account corresponding to the specified login name.

Permissions Required

The **sec_rgy_acct_delete()** routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission to remove management information.
- The **a** (*auth_info*) permission to remove authentication information.
- The **u** (**user_info**) permission to remove user information.

Notes

Even though the account is deleted, the PGO items corresponding to the account remain. These must be deleted with separate calls to **sec_rgy_pgo_delete()**.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to delete the specified account.

sec_rgy_object_not_found
No PGO item was found with the given name.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_pgo_delete(3sec)**.

sec_rgy_acct_get_projlist

Purpose

Returns the projects in an account's project list

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_get_projlist(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_cursor_t *projlist_cursor
     signed32 max_number
     signed32 *supplied_number
     uuid_t id_projlist[ ]
     signed32 unix_projlist[ ]
     signed32 *num_projects
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
: A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

max_number
: The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist[ ]* arrays.

Input/Output

projlist_cursor
: An opaque pointer indicating a specific project in an account's project list. The **sec_rgy_acct_get_projlist()** routine returns the project indicated by *projlist_cursor*, and advances the cursor to point to the next project in the list. When the end of the list is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

supplied_number
: A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input. If there are more projects in the account list, **sec_rgy_acct_get_projlist()** sets *projlist_cursor* to point to the next entry after the last one in the returned list.

id_projlist[ ]
: An array to receive the UUID of each project returned. The size allocated

for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_get_projlist()** routine returns members of the project list for the specified account. It returns the project information in two arrays. The *id_projlist[ ]* array contains the UUIDs for the returned projects. The *unix_projlist[ ]* array contains the UNIX numbers for the returned projects.

The project list cursor, *projlist_cursor*, provides an automatic place holder in the project list. The **sec_rgy_acct_get_projlist()** routine automatically updates this variable to point to the next project in the project list. To return an entire project list, reset *projlist_cursor* with **sec_rgy_cursor_reset()** on the initial call and then issue successive calls until all the projects are returned.

Permissions Required

The **sec_rgy_acct_get_projlist()** routine requires the **r** (**read**) permission on the account principal for which the project list data is to be returned.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to refresh a cursor for use with another call or for another server.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of projects.

sec_rgy_not_authorized
The client program is not authorized to see a project list for this principal.

sec_rgy_object exists
The account to be added already exists.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_get_next(3sec)**.

sec_rgy_acct_lookup

Purpose

Returns data for a specified account

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_lookup(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *name_key
     sec_rgy_cursor_t *account_cursor
     sec_rgy_login_name_t *name_result
     sec_rgy_sid_t *id_sid
     sec_rgy_unix_sid_t *unix_sid
     sec_rgy_acct_key_t *key_parts
     sec_rgy_acct_user_t *user_part
     sec_rgy_acct_admin_t *admin_part
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_key
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. Blank strings serve as wildcards, matching any entry.

Input/Output

account_cursor
An opaque pointer to a specific account in the registry database. If *name_key* is blank, **sec_rgy_acct_lookup()** returns information about the account to which the cursor is pointing. On return, the cursor points to the next account in the database after the returned account. If *name_key* is blank and the *account_cursor* has been reset with **sec_rgy_cursor_reset()**, **sec_rgy_acct_lookup()** returns information about the first account in the database.

When the end of the list of accounts in the database is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to refresh the cursor.

Output

name_result
A pointer to the full login name of the account (including all three names) for which the information is returned. The remaining parameters contain the information belonging to the returned account.

id_sid A structure containing the three UUIDs of the principal, group, and organization for the account.

unix_sid
A structure containing the three UNIX numbers of the principal, group, and organization for the account.

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

user_part
A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell, all of which are accessible to, and may be modified by, the account owner.

admin_part
A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_lookup()** routine returns all the information about an account in the registry database. The account can be specified either with *name_key* or *account_cursor*. If *name_key* is completely blank, the routine uses the *account_cursor* value instead.

For *name_key*, a zero-length principal, group, or organization key serves as a wildcard. For example, a login name key with the principal and organization fields blank returns the next (possibly first) account whose group matches the input group field. The full login name of the returned account is passed back in *name_result*.

The *account_cursor* provides an automatic place holder in the registry database. The routine automatically updates this variable to point to the next account in the database, after the account for which the information was returned. If *name_key* is blank and the *account_cursor* has been reset with **sec_rgy_cursor_reset()**, **sec_rgy_acct_lookup()** returns information about the first account in the database.

Permissions Required

The **sec_rgy_acct_lookup()** routine requires the **r** (**read**) permission on the account principal to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the accounts in the registry.

sec_rgy_object_not_found
The input account could not be found by the registry server.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_admin_replace(3sec)**, **sec_rgy_acct_replace_all(3sec)**, **sec_rgy_acct_user_replace(3sec)**, **sec_rgy_cursor_reset(3sec)**.

sec_rgy_acct_passwd

Purpose

Changes the password for an account

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_passwd(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_passwd_rec_t *caller_key
     sec_passwd_rec_t *new_key
     sec_passwd_type_t new_keytype
     sec_passwd_version_t *new_key_version
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
: A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three strings must be completely specified.

caller_key
: The key to use to encrypt the key for transmission to the registry server.

new_key
: The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
: The type of the new key. The server uses this parameter to decide how to encode *new_key* if it is sent as plaintext.

Output

new_key_version
: The key version number returned by the server. If the client requests a particular key version number (via the *version_number* field of the *new_key* input parameter), the server returns the requested version number back to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_passwd()** routine changes an account password to the input password character string. Wildcards (blank fields) are not permitted in the specified account name; the principal, group, and organization names of the account must be completely specified.

Permissions Required

The **sec_rgy_acct_passwd()** routine requires the **u** (**user_info**) permission on the account principal whose password is to be changed.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to change the password of this account.

sec_rgy_object_not_found
The account to be modified was not found by the registry server.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_acct_rename

Purpose

Changes an account login name

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_rename(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *old_login_name
     sec_rgy_login_name_t *new_login_name
     sec_rgy_acct_key_t *new_key_parts
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

old_login_name
A pointer to the current account login name. The login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three strings must be completely specified.

new_login_name
A pointer to the new account login name. Again, all three component names must be completely specified.

Input/Output

new_key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_rename()** routine changes an account login name from *old_login_name* to *new_login_name*. Wildcards (empty fields) are not permitted in either input name; both the old and new login names must completely specify their component principal, group, and organization names. Note, though, that the principal component in a login name cannot be changed.

The *new_key_parts* variable identifies how many of the *new_login_name* parts to use as the unique abbreviation for the account. If the requested abbreviation

duplicates an existing abbreviation for another account, the routine identifies the next shortest unique abbreviation and returns this abbreviation using *new_key_parts*.

Permissions Required

The **sec_rgy_acct_rename()** routine requires the **m** (**mgmt_info**) permission on the account principal to be renamed.

Notes

The **sec_rgy_acct_rename()** routine does not affect any of the registry PGO data. The constituent principal, group, and organization items for an account must be added before the account can be created. (See the **sec_rgy_pgo_add()** routine). Also, the principal must have been added as a member of the specified group and organization. (See the **sec_rgy_pgo_add_member()** routine).

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to make the changes.

sec_rgy_object_not_found
The account to be modified was not found by the registry server.

sec_rgy_name_exists
The new account name is already in use by another account.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**.

sec_rgy_acct_replace_all

Purpose

Replaces all account data for an account

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_replace_all(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_key_t *key_parts
     sec_rgy_acct_user_t *user_part
     sec_rgy_acct_admin_t *admin_part
     boolean32 set_password
     sec_passwd_rec_t *caller_key
     sec_passwd_rec_t *new_key
     sec_passwd_type_t new_keytype
     sec_passwd_version_t *new_key_version
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
: A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

user_part
: A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This information can be modified only by the account owner or other authorized user.

admin_part
: A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator.

set_passwd
: The password reset flag. If you set this parameter to TRUE, the account's password will be changed to the value specified in *new_key*.

caller_key
: A key to use to encrypt the key for transmission to the registry server. If communications secure to the **rpc_c_authn_level_pkt_privacy** level are available on a system, then this parameter is not necessary, and the packet encryption is sufficient to ensure security.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode the plaintext key.

Input/Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the *version_number* field of the *new_key* input parameter), the server returns the requested version number back to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_replace_all()** routine replaces both the user and administrative information in the account record specified by the input login name. The administrative information contains limitations on the account's use and privileges. The user information contains information such as the account home directory and default shell. The administrative information can only be modified by a registry administrator or another authorized user (users with **admin_info** (**a**) and **mgmt_info** (**m**) privileges for an account). The user information can be modified by the account owner or another authorized user (users with **user_info** (**u**) privileges for an account).

Use the *set_passwd* parameter to reset the account password. If you set this parameter to TRUE, the account's pasword is changed to the value specified in *new_key*.

The *key_parts* variable identifies how many of the *login_name* parts to use as the unique abbreviation for the replaced account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine identifies the next shortest unique abbreviation and returns this abbreviation using *key_parts*.

Permissions Required

The **sec_rgy_acct_replace_all()** routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission, if **flags** or **expiration_date** is to be changed.
- The **a** (*auth_info*) permission, if **authentication_flags** or **good_since_date** is to be changed.
- The **u** (**user_info**) permission, if user **flags**, **gecos**, **homedir** (home directory), **shell**, or **passwd** (password) are to be changed.

Notes

All users need the **w** (**write**) privilege to modify any account information.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to change account information.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_acct_admin_replace(3sec)**, **sec_rgy_acct_rename(3sec)**, **sec_rgy_acct_user_replace(3sec)**.

sec_rgy_acct_user_replace

Purpose

Replaces user account data

Synopsis

```
#include <dce/acct.h>

void  sec_rgy_acct_user_replace(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_user_t *user_part
     boolean32 set_passwd
     sec_passwd_rec_t *caller_key
     sec_passwd_rec_t *new_key
     sec_passwd_type_t new_keytype
     sec_passwd_version_t *new_key_version
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

user_part
A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This information can be modified only by the account owner or other authorized user. The structure contains the following fields:

gecos A character string containing information about the account owner. This often includes such information as their name and telephone number.

homedir
The default directory upon login for the account.

shell The default shell to use upon login.

passwd_version_number
The password version number, a 32-bit unsigned integer, set by the registry server.

passwd_dtm
The date and time of the last password change (in **sec_timeval_sec_t** form), also set by the registry server.

flags A flag set of type **sec_rgy_acct_user_flags_t**.

passwd
The account's encrypted password.

The only user data fields that can be changed are: **gecos**, **homedir**, **shell**, **flags**, and **passwd**.

set_passwd
The password reset flag. If you set this parameter to TRUE, the user's password will be changed to the value specified in *new_key*.

caller_key
A key to use to encrypt the key for transmission to the registry server. If communications secure to the **rpc_c_authn_level_pkt_privacy** level are available on a system, then this parameter is not necessary, and the packet encryption is sufficient to ensure security.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode the plaintext key.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the *version_number* field of the *new_key* input parameter), the server returns the requested version number back to the client.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_acct_user_replace()** routine replaces the user information in the account record specified by the input login name. The user information consists of information such as the account home directory and default shell. The the user information can be modified only by the account owner or other authorized users (users with **user_info** (**u**) privileges for an account).

Use the *set_passwd* parameter to reset the user's password. If you set this parameter to TRUE, the user's pasword is changed to the value specified in *new_key*.

Permissions Required

The **sec_rgy_acct_user_replace()** routine requires the **u** (**user_info**) permission on the account principal.

Notes

All users need the **w** (**write**) privilege to modify any account information.

Files

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to modify the account data.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_acct_admin_replace(3sec)**, **sec_rgy_acct_rename(3sec)**, **sec_rgy_acct_replace_all(3sec)**.

sec_rgy_attr_cursor_alloc

Purpose

Allocates resources to a cursor used by sec_rgy_attr_lookup_by_id

Synopsis

```
#include <dce/sec_rgy_attr.h>

void  sec_rgy_attr_cursor_alloc(
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Output

cursor A pointer to a **sec_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 **sec_rgy_attr_cursor_alloc()** call allocates resources to a cursor used with the **sec_rgy_attr_lookup_by_id** call. This routine, which is a local operation, does not initialize *cursor*.

The **sec_rgy_attr_cursor_init()** routine, which makes a remote call, allocates and initializes the cursor. In addition, **sec_rgy_attr_cursor_init()** returns the total number of attributes attached to the object as an output parameter; **sec_rgy_attr_cursor_alloc()** does not.

Permissions Required

None.

Files

/usr/include/dce/sec_attr_base.idl
The **idl** file from which **dce/sec_attr_base.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

no such object

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_cursor_init(3sec)**, **sec_rgy_attr_cursor_release(3sec)**, **sec_rgy_attr_cursor_reset(3sec)**,

sec_rgy_attr_lookup_by_id(3sec).

sec_rgy_attr_cursor_init

Purpose

Initializes a cursor used by sec_rgy_attr_lookup_by_id

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_init  (
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     unsigned32 *cur_num_attrs
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute to be scanned is attached.

Output

cur_num_attrs
A pointer to an unsigned 32-bit integer that specifies the number of attributes currently attached to the object.

cursor A pointer to a **sec_rgy_cursor_t** positioned at the first attribute in the list of the object's attributes.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_cursor_init()** routine initializes a cursor of type **sec_attr_cursor_t** (used with the **sec_rgy_attr_lookup_by_id** call) and initializes the cursor to the first attribute in the specified object's list of attributes. This call also supplies the total number of attributes attached to the object as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the registry.

Use the **sec_rgy_attr_cursor_release()** call to release all resources allocated to a **sec_attr_cursor_t** cursor.

Permissions Required

The **sec_rgy_attr_cursor_init()** routine requires at least one permission (of any type) on the person, group, or organization to which the attribute to be scanned is attached.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

no such object

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_cursor_release,** **sec_rgy_attr_lookup_by_id**.

sec_rgy_attr_cursor_release

Purpose

Releases a cursor

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_release  (
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

Input/Output

cursor As an input parameter, a pointer to an uninitialized cursor of type **sec_attr_cursor_t**. As an output parameter, a pointer to an uninitialized cursor of type **sec_attr_cursor_t** 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 **sec_rgy_attr_cursor_release()** routine releases all resources allocated to a **sec_attr_cursor_t** by the **sec_rgy_attr_cursor_init()** or **sec_rgy_attr_cursor_alloc()** call.

This is a local-only operation and makes not remote calls.

Permissions Required

None.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

No such object

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_cursor_alloc(3sec)**, **sec_rgy_attr_cursor_init(3sec)**, **sec_rgy_attr_lookup_by_id**.

sec_rgy_attr_cursor_reset

Purpose

Reinitializes a cursor

Synopsis

```
#include <dce/sec_attr_base.h>

void  sec_attr_cursor_reset(
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor A pointer to a **sec_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 **sec_rgy_attr_cursor_reset()** routine resets a **dce_attr_cursor_t** that has been allocated by either a **sec_rgy_attr_cursor_init()** or **sec_rgy_attr_cursor_alloc()**. The reset cursor can then be used to process a new **sec_rgy_attr_lookup_by_id** 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/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_cursor_alloc(3sec)**, **sec_rgy_attr_cursor_init(3sec)**, **sec_rgy_attr_lookup_by_id(3sec)**.

sec_rgy_attr_delete

Purpose

Deletes specified attributes for a specified object

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_delete  (
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     unsigned32 num_to_delete
     sec_attr_t attrs[ ]
     signed32 *failure_index
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object identified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attributes are attached.

num_to_delete
A 32-bit integer that specifies the number of elements in the *attrs[ ]* array. This integer must be greater than 0.

attrs[ ]
An array of values of type **sec_attr_t** that specifies the attribute instances to be deleted. The size of the array is determined by *num_to_delete*.

Output

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *in_attrs[ ]* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **−1**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_delete()** routine deletes attributes. This is an atomic operation: if the deletion of any attribute in the *attrs[ ]* array fails, all deletions are aborted. The attribute causing the delete to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

The *attrs[ ]* array, which specifies the attributes to be deleted, contains values of type **sec_attr_t** These values consist of

- *attr_id*, a UUID that identifies the attribute type
- *attr_value*, values of **sec_attr_value_t** that specify the attribute's encoding type and values.

To delete attributes that are not multivalued and to delete all instances of a multivalued attribute, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

To delete a specific instance of a multivalued attribute, supply the UUID and value that uniquely identify the multivalued attribute instance in the input array.

Note that if the deletion of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose deletion failed in a failure index by array element number.

Permissions Required

The **sec_rgy_attr_delete()** routine requires the delete permission set for each attribute type identified in the *attrs[ ]* array. These permissions are defined as part of the ACL manager set in the schema entry for the attribute type.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

database read only

invalid/unsupported attribute type

server unavailable

site read only

unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_update(3sec)**.

sec_rgy_attr_get_effective

Purpose

Reads effective attributes by ID

Synopsis

```
#include <dce/sec_rgy_attr.h>

void  sec_rgy_attr_get_effective(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     unsigned32 num_attr_keys
     sec_attr_t attr_keys[ ]
     sec_attr_vec_t *attr_list
     error_status_t status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_principal
The *name* identifies a principal.

sec_rgy_domain_group
The *name* identifies a group.

sec_rgy_domain_org
The *name* identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys
An unsigned 32-bit integer that specifies the number of elements in the *attr_keys*[ ] array. If *num_attr_keys* is set to 0 (zero), all of the effective attributes that the caller is authorized to see are returned.

*attr_keys***[ ]**
An array of values of type **sec_attr_t** that specify the UUIDs of the attributes to be returned if they are effective. If the attribute type is associated with a query attribute trigger, the **sec_attr_t** *attr_value* field can be used to pass in optional information required by the attribute trigger query. If no information is to be passed in the *attr_value* field (whether the type indicates an attribute trigger query or not), set the attribute's encoding type to **sec_rgy_attr_enc_void**. The size of the *attr_keys*[ ] array is determined by the *num_attr_keys* parameter.

Output

attr_list
A pointer an attribute vector allocated by the server containing all of the effective attributes matching the search criteria (defined in *num_attr_keys* or *attr_keys*[ ]). The server allocates a buffer large enough to return all the requested attributes so that subsequent calls are not necessary.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_get_effective()** routine returns the UUIDs of a specified object's effective attributes. Effective attributes are determined by setting of the schema entry **apply_defaults** flag:

- If the flag is set off, only the attributes directly attached to the object are effective.
- If the flag is set on, the effective attributes are obtained by performing the following steps for each attribute identified by UUID in the *attr_keys* array:
  - If the object named by *name* is a principal and if the a requested attribute exists on the principal, that attribute is effective and is returned. If it does not exist, the search continues.
  - The next step in the search depends on the type of object:

    For principals with accounts:
    - The organization named in the principal's account is examined to see if an attribute of the requested type exists. If it does, it is effective and is returned; then the search for that attribute ends. If it does not exist, the search for that attribute continues to the **policy** object as described here.
    - The registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

    For principals without accounts, for groups, and for organizations:

    The registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

For multivalued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set; it does not return the set instance.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information. To do this

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the requested attribute type is associated with a query trigger, the value returned for the attribute will be the binding (as set in the schema entry) of the trigger server. The caller must bind to the trigger server and pass the original input query attribute to the **sec_attr_trig_query** call in order to retrieve the attribute value.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_attr_lookup_by_id

Purpose

Reads a specified object's attributes, expanding attribute sets into individual member attributes

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_lookup_by_id  (
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     sec_attr_cursor_t *cursor
     unsigned32 num_attr_keys
     unsigned32 space_avail
     sec_attr_t attr_keys[ ]
     unsigned32 *num_returned
     sec_attr_t attrs[ ]
     unsigned32 *num_left
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys
An unsigned 32-bit integer that specifies the number of elements in the *attr_keys[ ]* array. Set this parameter to 0 (zero) to return all of the object's attributes that the caller is authorized to see.

space_avail
An unsigned 32-bit integer that specifies the size of the *attr_keys[ ]* array.

attr_keys[ ]
An array of values of type **sec_attr_t** that identify the attribute type ID of the attribute instance(s) to be looked up. If the attribute type is associated

with a query attribute trigger, the **sec_attr_t** *attr_value* field can be used to pass in optional information required by the attribute trigger query. If no information is to be passed in the *attr_value* field (whether the type indicates an attribute trigger query or not), set the attribute's encoding type to **sec_rgy_attr_enc_void**.

The size of the *attr_keys[ ]* array is determined by the *num_attr_keys* parameter.

Input/Output

cursor A pointer to a **sec_attr_cursor_t**. As an input parameter, *cursor* is a pointer to a **sec_attr_cursor_t** initialized by a **sec_rgy_attr_srch_cursor_init** call. As an output parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is positioned past components returned in this call.

Output

num_returned
A pointer to a 32-bit unsigned integer that specifies the number of attribute instances returned in the *attrs[ ] array*.

attrs[ ]
An array of values of type **sec_attr_t** that contains the attributes retrieved by Universal Unique Identifier (UUID). The size of the array is determined by *space_avail* and the length by *num_returned*.

num_left
A pointer to a 32-bit unsigned integer that supplies the number of attributes that were found but could not be returned because of space constraints in the *attrs[ ]* buffer. To ensure that all the attributes will be returned, increase the size of the *attrs[ ]* array by increasing the size of *space_avail* and *num_returned*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**, or, if the requested attributes were not available, it returns the message **not_all_available**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_lookup_by_id()** function reads those attributes specified by UUID for an object specified by name. This routine is similar to the **sec_rgy_attr_lookup_no_expand()** routine with one exception: for attribute sets, the **sec_rgy_attr_lookup_no_expand()** routine returns a **sec_attr_t** for the set instance only; it does not expand the set and return a **sec_attr_t** for each member in the set. This call expands attribute sets and returns a **sec_attr_t** for each member in the set.

If the *num_attr_keys* parameter is set to 0 (zero), all of the object's attributes that the caller is authorized to see are returned. This routine is useful for programmatic access.

After a successful call, free the resources allocated by this routine for each attribute returned in the *attrs[ ]* parameter with the **sec_attr_util_inst_free_ptrs()** routine.

For multivalued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set; it does not return the set instance.

The *attr_keys[ ]* array, which specifies the attributes to be returned, contains values of type **sec_attr_t**. These values consist of the following:

- *attr_id*, a UUID that identifies the attribute type
- *attr_value*, values of **sec_attr_value_t** that specify the attribute's encoding type and values.

Use the *attr_id* field of each *attr_keys[ ]* array element, to specify the UUID that identifies the attribute type to be returned.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information, as follows:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

Note that if you set *num_attr_keys* to zero to return all of the object's attributes and that attribute is associated with a query attribute trigger, the attribute trigger will be called with no input attribute information (that would normally have been passed in via the *attr_value* field).

The *cursor* parameter specifies a cursor of type **sec_attr_cursor_t** initialized to the point in the attribute list at which to start processing the query. Use the **sec_attr_cursor_init** function to initialize *cursor*. If *cursor* is uninitialized, the behavior is undefined.

The *num_left* parameter contains the number of attributes that were found but could not be returned because of space constraints in the *attrs[ ]* array. (Note that this number may be inaccurate if the target server allows updates between successive queries.) To obtain all of the remaining attributes, set the size of the *attrs[ ]* array so that it is large enough to hold the number of attributes listed in *num_left*.

Permissions Required

The **sec_rgy_attr_lookup_by_id()** routine requires the **q** (**query**) permission set for each attribute type identified in the *attr_keys[ ]* array. These permissions are defined as part of the access control list (ACL) manager set in the schema entry of each attribute type.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

unauthorized

registry server unavailable

trigger server unavailable

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_attr_lookup_by_name(3sec)**, **sec_rgy_attr_lookup_no_expand(3sec)**.

sec_rgy_attr_lookup_by_name

Purpose

Reads a single attribute instance for a specific object

Synopsis

```
#include <dce/sec_rgy_attr.h>

void  sec_rgy_attr_lookup_by_name(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     char *attr_name
     sec_attr_t *attr
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_principal
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

attr_name
A pointer to a character string that specifies the name of the attribute to be retrieved.

Output

attr A pointer to a **sec_attr_t** that contains the first instance of the named attribute.

status A pointer to the completion status. The completion status can be one of the following:

error_status_ok
All instances of the value were returned with no errors.

more_available
A multivalued attribute was specified as *name* and the routine

completed successfully. For multivalued attributes, this routine returns the first instance of the attribute.

attribute_set_instance
An attribute set was specified as *name* and the routine completed successfully.

An error message if the routine did not complete successfully.

Description

The **sec_rgy_attr_lookup_by_name()** routine returns the named attribute for a named object. This routine is useful for an interactive editor.

For multivalued attributes, this routine returns the first instance of the attribute. To retrieve every instance of the attribute, use the **sec_rgy_attr_lookup_by_id** call, supplying the attribute Universal Unique Identifier (UUID) returned in the *attr* parameter.

For attribute sets, the routine returns the attribute set instance, not the member instances. To retrieve all members of the set, use the **sec_rgy_attr_lookup_by_id** call, supplying the the attribute set UUID returned in the *attr* parameter.

After a successful call, free the resources allocated by this routine for the **attr** parameter, with the **sec_attr_util_inst_free_ptrs()** routine.

Attention:

This routine does not provide for input data to an attribute trigger query operation. If the named attribute is associated with a query attribute trigger, the attribute trigger will be called with no input attribute value information.

Permissions Required

The **sec_rgy_attr_lookup_by_name()** routine requires the **q** (**query**) permission set for the attribute type of the attribute instance identified by *attr_name*. These permissions are defined as part of the access control list (ACL) manager set in the schema entry of each attribute type.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

registry server unavailable

trigger server unavailable

unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_lookup_by_id(3sec)**, **sec_rgy_attr_lookup_no_expand(3sec)**.

sec_rgy_attr_lookup_no_expand

Purpose

Reads a specified object's attribute(s), without expanding attribute sets into individual member attributes

Synopsis

```
#include <dce/sec_rgy_attr.h>

void  sec_rgy_attr_lookup_no_expand(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     sec_attr_cursor_t *cursor
     unsigned32 num_attr_keys
     unsigned32 space_avail
     uuid_t attr_keys[ ]
     unsigned32 *num_returned
     sec_attr_t attr_sets[ ]
     unsigned32 *num_left
     error_status_t status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_principal
The *name* identifies a principal.

sec_rgy_domain_group
The *name* identifies a group.

sec_rgy_domain_org
The *name* identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys
An unsigned 32-bit integer that specifies the number of elements in the *attr_keys[ ]* array. If *num_attr_keys* is set to 0 (zero), all attribute sets that the caller is authorized to see are returned.

space_avail
An unsigned 32-bit integer that specifies the size of the *attrs_sets*[ ] array.

attr_keys[ ]
An array of values of type **uuid_t** that specify the UUIDs of the attribute sets to be returned. The size of the *attr_keys[ ]* array is determined by the *num_attr_keys* parameter.

Input/Output

cursor A pointer to a **sec_attr_cursor_t**. As an input parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is initialized by the **sec_rgy_attr_cursor_init**. As an output parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is positioned past the attribute sets returned in this call.

Output

num_returned
A pointer to a 32-bit integer that specifies the number of attribute sets returned in the *attrs*[ ] array.

attr_sets[ ]
An array of values of type **sec_attr_t** that contains the attribute sets retrieved by UUID. The size of the array is determined by *space_avail* and the length by *num_returned*.

num_left
A pointer to a 32-bit unsigned integer that supplies the number of attribute sets that were found but could not be returned because of space constraints in the *attr_sets[ ]* buffer. To ensure that all the attributes will be returned, increase the size of the *attr_sets[ ]* array by increasing the size of *space_avail* and *num_returned*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_lookup_no_expand()** routine reads attribute sets. This routine is similar to the **sec_rgy_attr_lookup_by_id()** routine with one exception: for attribute sets, the **sec_rgy_attr_lookup_by_id()** routine expands attribute sets and returns a **sec_attr_t** for each member in the set. This call does not. Instead it returns a **sec_attr_t** for the set instance only. The **sec_rgy_attr_lookup_no_expand()** routine is useful for programmatic access.

cursor is a cursor of type **sec_attr_cursor_t** that establishes the point in the attribute set list from which the server should start processing the query. Use the **sec_rgy_attr_cursor_init** function to initialize *cursor*. If *cursor* is uninitialized, the behavior is undefined.

The *num_left* parameter contains the number of attribute sets that were found but could not be returned because of space constraints of the *attr_sets[ ]* array. (Note that this number may be inaccurate if the target server allows updates between successive queries.) To obtain all of the remaining attribute sets, set the size of the *attr_sets[ ]* array so that it is large enough to hold the number of attributes listed in *num_left*.

Permissions Required

The **sec_rgy_attr_lookup_no_expand()** routine requires the query permission set for each attribute type identified in the *attr_keys[ ]* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

invalid/unsupported attribute type

registry server unavailable

unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_lookup_by_id(3sec)**, **sec_rgy_attr_lookup_by_name(3sec)**.

sec_rgy_attr_sch_aclmgr_strings

Purpose

Returns printable ACL strings associated with an ACL manager protecting a schema object

Synopsis

```
#include  <dce/dce_attr_base.h>

void  sec_rgy_attr_sch_aclmgr_strings(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     uuid_t *acl_mgr_type
     unsigned32 size_avail
     uuid_t *acl_mgr_type_chain
     sec_acl_printstring_t *acl_mgr_info
     boolean32 *tokenize
     unsigned32 *total_num_printstrings
     unsigned32 *size_used
     sec_acl_printstring_t permstrings[ ]
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
: Reserved for future use.

acl_manager_type
: A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the schema object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use **sec_rgy_attr_sch_get_acl_mgrs()** to acquire a list of the manager types protecting a given schema object.

size_avail
: An unsigned 32-bit integer containing the allocated length of the *permstrings[ ]* array.

Output

acl_mgr_type_chain
: If the target object ACL contains more than 32 permission bits, chains of manager types are used: each manager type holds one 32-bit segment of permissions. The UUID returned in *acl_mgr_type_chain* refers to the next ACL manager in the chain. If there are no more ACL managers in the chain, **uuid_nil** is returned.

acl_mgr_info
: A pointer to a printstring that contains the ACL manager type's name, help information, and set of supported of permission bits.

tokenize
A pointer to a variable that specifies whether or not printstrings will be passed separately:

- TRUE indicates that the printstrings must be printed or passed separately.
- FALSE indicates that the printstrings are unambiguous and can be concatenated when printed without confusion.

total_num_printstrings
A pointer to an unsigned 32-bit integer containing the total number of permission entries supported by this ACL manager type.

size_used
A pointer to an unsigned 32-bit integer containing the number of permission entries returned in the *permstrings[ ]* array.

permstrings[ ]
An array of printstrings of type **sec_acl_printstring_t**. Each entry of the array is a structure containing the following three components:

printstring
A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.

helpstring
A character string of maximum length **sec_acl_printstring_help_len** containing some text that can be used to describe the specified permission.

permissions
A **sec_acl_permset_t** permission set describing the permissions that are represented with the companion printstring.

The array consists of one such entry for each permission supported by the ACL manager identified by *acl_mgr_type*.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_sch_aclmgr_strings()** routine returns an array of printable representations (called *printstrings*) for each permission bit or combination of permission bits the specified ACL manager supports. The ACL manager type specified by *acl_mgr_type* must be one of the types protecting the schema object bound to by *h*.

In addition to returning the printstrings, this routine also returns instructions about how to print the strings in the *tokenize* variable. If this variable is set to FALSE, the printstrings can be concatenated. If it is set to TRUE, the printstrings cannot be concatenated. For example a printstrings of **r** or **w** could be concatenated as **rw** without any confusion. However, printstrings in a form of **read** or **write**, should not be concatenated.

ACL managers often define aliases for common permission combinations. By convention, simple entries appear at the beginning of the *printstrings*[ ] array, and combinations appear at the end.

Permissions Required

The **sec_rgy_attr_sch_scl_mgr_strings()** routine requires the **r** permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_no_memory

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_get_acl_mgrs(3sec)**.

sec_rgy_attr_sch_create_entry

Purpose

Creates a schema entry

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_create_entry(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     sec_attr_schema_entry_t *schema_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

schema_entry
A pointer to a **sec_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 **sec_rgy_attr_sch_create_entry()** routine creates schema entries that define attribute types.

Permissions Required

The **sec_rgy_attr_sch_create_entry()** routine requires **i** permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_bad_acl_mgr_set

sec_attr_bad_acl_mgr_type

sec_attr_bad_bind_authn_svc

sec_attr_bad_bind_authz_svc

sec_attr_bad_bind_info

sec_attr_bad_bind_prot_level

sec_attr_bad_bind_svr_name

sec_attr_bad_comment

sec_attr_bad_encoding_type

sec_attr_bad_intercell_action

sec_attr_bad_name

sec_attr_bad_permset

sec_attr_bad_scope

sec_attr_bad_uniq_query_accept

sec_attr_name_exists

sec_attr_no_memory

sec_attr_svr_read_only

sec_attr_svr_unavailable

sec_attr_trig_bind_info_missing

sec_attr_type_id_exists

sec_attr_unauthorized

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_delete_entry(3sec)**, **sec_rgy_attr_sch_update(3sec)**.

sec_rgy_attr_sch_cursor_alloc

Purpose

Allocates resources to a cursor used with sec_rgy_attr_sch_scan

Synopsis

```
void  sec_rgy_att_sch_cursor_alloc(
     dce_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Output

cursor A pointer to a **sec_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 **sec_rgy_attr_sch_cursor_alloc()** call allocates resources to a cursor used with the **sec_rgy_attr_sch_scan()** call. This routine, which is a local operation, does not initialize *cursor*.

The **sec_rgy_attr_sch_cursor_init()** routine, which makes a remote call, allocates and initializes the cursor. In addition, **sec_rgy_attr_sch_cursor_init()** returns the total number of entries found in the schema as an output parameter; **sec_rgy_attr_sch_cursor_alloc()** does not.

Permissions Required

None.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.id** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_no_memory

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_cursor_init(3sec)**, **sec_rgy_attr_sch_cursor_release(3sec)**, **sec_rgy_attr_sch_scan(3sec).**

sec_rgy_attr_sch_cursor_init

Purpose

Initializes and allocates a cursor used with sec_rgy_attr_sch_scan

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_cursor_init(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     unsigned32 *cur_num_entries
     sec_attr_cursor_t *cursor
     error_status_t status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

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 **sec_attr_cursor_t** that is initialized to the first entry in 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 **sec_rgy_attr_sch_cursor_init()** call initializes and allocates resources to a cursor used with the **sec_rgy_attr_sch_scan** call. This call makes remote calls to initialize the cursor.

To limit the number of remote calls, use the **sec_rgy_attr_sch_cursor_alloc()** call to allocate *cursor*, but not initialize it. Be aware, however, that the **sec_rgy_attr_sch_cursor_init()** call supplies the total number of entries found in the schema as an output parameter; the **sec_rgy_attr_sch_cursor_alloc()** call does not.

If the cursor iunput to **sec_rgy_attr_sch_scan** has not been initialized, the **sec_rgy_attr_sch_scan** call will initialize it; if it has been initialized, **sec_rgy_attr_sch_scan** advances it.

Permissions Required

None.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_no_memory

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_cursor_alloc(3sec),sec_rgy_attr_sch_cursor_release(3sec)**, **sec_rgy_attr_sch_scan(3sec).**

sec_rgy_attr_sch_cursor_release

Purpose

Releases states associated with a cursor used by sec_rgy_attr_sch_scan

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_cursor_release(
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor A pointer to a **sec_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 releases.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_sch_cursor_init()** routine releases the resources allocated to the cursor used by the **sec_rgy_attr_sch_scan** routine. This call is a local operation and makes no remote calls.

Permissions Required

None.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_cursor_allocate(3sec)**, **sec_rgy_attr_sch_cursor_init(3sec)**, **sec_rgy_attr_sch_scan(3sec)**.

sec_rgy_attr_sch_cursor_reset

Purpose

Resets a cursor that has been allocated

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  dce_attr_cursor_reset(
     sec_attr_cursor_t *cursor
     error_status_t *status);
```

Parameters

Input/Output

cursor A pointer to a **sec_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 **sec_rgy_attr_sch_cursor_reset()** routine resets a **dce_attr_cursor_t** that has been allocated by either a **sec_rgy_attr_sch_cursor_init()** or **sec_rgy_attr_sch_cursor_alloc()**. The reset cursor can then be used to process a new **sec_rgy_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/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_bad_cursor

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_cursor_alloc(3sec)**, **sec_rgy_attr_sch_cursor_init(3sec)**, **sec_rgy_attr_sch_scan(3sec)**.

sec_rgy_attr_sch_delete_entry

Purpose

Deletes a schema entry

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_delete_entry(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     uuid_t *attr_id
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

attr_id A pointer to a **uuid_t** that identifies the schema entry to be deleted.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_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 **sec_rgy_attr_sch_delete_entry()** routine requires the **d** permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_no_memory

sec_attr_sch_entry_not_found

sec_attr_svr_read_only

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_create_entry(3sec)**, **sec_rgy_attr_sch_update_entry(3sec)**.

sec_rgy_attr_sch_get_acl_mgrs

Purpose

Retrieves the manager types of the ACLs protecting the objects dominated by a named schema

Synopsis

```
#include  <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_get_acl_mgrs(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     unsigned32 size_avail
     unsigned32 *size_used
     unsigned32 *num_acl_mgr_types
     uuid_t acl_mgr_types[ ]
     error_status_t *status);
```

Parameters

Input

context
: An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
: Reserved for future use.

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 **sec_rgy_attr_sch_get_acl_mgrs()** routine returns a list of the manager types protecting the schema object identified by *context*.

ACL editors and browsers can use this operation to determine the ACL manager types protecting a selected schema object. Then, using the **sec_rgy_attr_sch_aclmgr_strings()** routine, they can determine how to format for display the permissions supported by that ACL manager type.

Permissions Required

The **sec_rgy_attr_sch_get_acl_mgrs()** routine requires the **r** permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_no_memory

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_aclmgr_strings(3sec)**.

sec_rgy_attr_sch_lookup_by_id

Purpose

Reads a schema entry identified by UUID

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_lookup_by_id(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     uuid_t *attr_id
     sec_attr_schema_entry_t *schema_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

attr_id A pointer to a **uuid_t** that identifies a schema entry.

Output

schema_entry
A **sec_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 **sec_rgy_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, use the **sec_attr_util_sch_ent_free_ptrs()** routine to free the resources allocated by this routine for the *schema_entry* parameter.

Permissions Required

The **sec_rgy_attr_sch_lookup_by_id()** routine requires the **r** (**read**) permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_sch_entry_not_found

sec_attr_svr_unavailable

sec_attr_unauthorized

sec_attr_no_memory

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_lookup_by_name(3sec)**, **sec_rgy_attr_sch_scan(3sec)**.

sec_rgy_attr_sch_lookup_by_name

Purpose

Reads a schema entry identified by name

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_lookup_by_name(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     char *attr_name
     sec_attr_schema_entry_t *schema_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

attr_name
A pointer to a character string that identifies the schema entry.

Output

schema_entry
A **sec_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 **sec_rgy_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, use the **sec_attr_util_sch_ent_free_ptrs()** routine to free the resources allocated by this routine for the *schema_entry* parameter.

Permissions Required

The **sec_rgy_attr_sch_lookup_by_name()** routine requires the **r** (**read**) permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_bad_name

sec_attr_no_memory

sec_attr_sch_entry_not_found

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_lookup_by_id(3sec)**, **sec_rgy_attr_sch_scan(3sec)**.

sec_rgy_attr_sch_scan

Purpose

Reads a specified number of schema entries

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_scan(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     sec_attr_cursor_t *cursor
     unsigned32 num_to_read
     unsigned32 *num_read
     sec_attr_schema_entry_t schema_entries[ ]
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

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 **sec_attr_cursor_t**. As input *cursor* must be allocated and can be initialized. If *cursor* is not initialized, **sec_rgy_attr_sch_scan** will initialized. As output, *cursor* is positioned at the first schema entry after the returned entries.

Output

num_read
A pointer an unsigned 32-bit integer specifying the number of entries returned in *schema_entries[ ]*.

schema_entries[ ]
A **sec_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 **sec_rgy_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 **sec_rgy_attr_sch_cursor_init()** or the **sec_rgy_attr_sch_cursor_alloc()** call. If the input *cursor* is not initialzed, **sec_rgy_attr_sch_scan()** initializes it; if cursor is initialized, **sec_rgy_attr_sch_scan()** simply advances it.

To read all entries in a schema, make successive **sec_rgy_attr_sch_scan()** calls. When all entries have been read, the call returns the message **no_more_entries**.

This routine is useful as a browser.

Permissions Required

The **sec_rgy_attr_sch_scan()** routine requires **r** permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_bad_cursor

sec_attr_no_memory

sec_attr_svr_unavailable

sec_attr_unauthorized

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_cursor_alloc(3sec)**, **sec_rgy_attr_sch_cursor_init(3sec)**, **sec_rgy_attr_sch_cursor_release(3sec)**.

sec_rgy_attr_sch_update_entry

Purpose

Updates a schema entry

Synopsis

```
#include <dce/sec_rgy_attr_sch.h>

void  sec_rgy_attr_sch_update_entry(
     sec_rgy_handle_t context
     sec_attr_component_name_t schema_name
     sec_attr_schema_entry_parts_t modify_parts
     sec_attr_schema_entry_t *schema_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

schema_name
Reserved for future use.

modify_parts
A value of type **sec_attr_schema_entry_parts_t** that identifies the fields in *schema_entry* that can be modified.

schema_entry
A pointer to a **sec_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 **sec_rgy_attr_sch_update_entry()** routine modifies schema entries. Only those schema entry fields set to be modified in the **sec_attr_schema_entry_parts_t** data type can be modified.

Some schema entry components can never be modified. Instead 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 **sec_rgy_attr_sch_update_entry()** routine requires the **M** (**Member_list**) permission on the **attr_schema** object.

Files

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_attr_field_no_update

sec_attr_bad_name

sec_attr_bad_acl_mgr_set

sec_attr_bad_acl_mgr_type

sec_attr_bad_permset

sec_attr_bad_intercell_action

sec_attr_trig_bind_info_missing

sec_attr_bad_bind_info

sec_attr_bad_bind_svr_name

sec_attr_bad_bind_prot_level

sec_attr_bad_bind_authn_svc

sec_attr_bad_bind_authz_svc

sec_attr_bad_uniq_query_accept

sec_attr_bad_comment

sec_attr_name_exists

sec_attr_sch_entry_not_found

sec_attr_unauthorized

sec_attr_svr_read_only

sec_attr_svr_unavailable

sec_attr_no_memory

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_sch_create_entry(3sec)**, **sec_rgy_attr_sch_delete_entry(3sec)**.

sec_rgy_attr_test_and_update

Purpose

Updates specified attribute instances for a specified object only if a set of control attribute instances match the object's existing attribute instances

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_test_and_update  (
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     unsigned32 num_to_test
     sec_attr_t test_attrs[ ]
     unsigned32 num_to_write
     sec_attr_t update_attrs[ ]
     signed32 *failure_index
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attribute is attached.

num_to_test
An unsigned 32-bit integer that specifies the number of elements in the *test_attrs[ ]* array. This integer must be greater than 0 (zero).

test_attrs[ ]
An array of values of type **sec_attr_t** that specifies the control attributes. The update takes place only if the types and values of the control attributes exactly match those of the attribute instances on the named registry object. The size of the array is determined by *num_to_test*.

num_to_write
A 32-bit integer that specifies the number of attribute instances returned in the *update_attrs[ ]* array.

update_attrs[ ]
An array of values of type **sec_attr_t** that specifies the attribute instances to be updated. The size of the array is determined by *num_to_write*.

Output

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *update_attrs[ ]* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **−1**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_test_and_update()** routine updates an attribute only if the set of control attributes specified in the *test_attrs[ ]* match attributes that already exist for the object.

This update is an atomic operation: if any of the control attributes do not match existing attributes, none of the updates are performed, and if an update should be performed, but the write cannot occur for whatever reason to any member of the *update_attrs*[ ] array, all updates are aborted. The attribute causing the update to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

If an attribute instance already exists which is identical in both *attr_id* and *attr_value* to an attribute specified in *in_attrs[ ]*, the existing attribute information is overwritten by the new information. For multivalued attributes, every instance with the same *attr_id* is overwritten with the supplied values.

If an attribute instance does not exist, it is created.

If you specify an attribute set for updating, the update applies to the set instance, the set itself, not the members of the set. To update a member of an attribute set, supply the UUID of the set member.

If an input attribute is associated with an update attribute trigger server, the attribute trigger server is invoked (by the **sec_attr_trig_update()** function) and the *in_attr*[ ] array is supplied as input. The output attributes from the update attribute trigger server are stored in the registry database and returned in the *out_attrs[ ]* array. Note that the update attribute trigger server may modify the values before they are used to update the registry database. This is the only circumstance under which the values in the *out_attrs[ ]* array differ from the values in the *in_attrs[ ]* array.

Permissions Required

The **sec_rgy_attr_test_and_update()** routine routine requires the test permission and the update permission set set for each attribute type identified in the *test_attrs[ ]* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

control attribute has changed

database read only

invalid encoding type

invalid/unsupported attribute type

server unavailable

site read only

trigger server unavailable

unauthorized

value not unique

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_delete(3sec)**, **sec_rgy_attr_update(3sec)**.

sec_rgy_attr_update

Purpose

Creates and updates attribute instances for a specified object

Synopsis

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_update  (
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     unsigned32 num_to_write
     unsigned32 space_avail
     sec_attr_t in_attrs[ ]
     unsigned32 *num_returned
     sec_attr_t out_attrs[ ]
     unsigned32 *num_left
     signed32 *failure_index
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attribute is attached.

num_to_write
A 32-bit unsigned integer that specifies the number of elements in the *in_attrs[ ]* array. This integer must be greater than 0 (zero).

space_avail
Set this parameter to zero. It is a 32-bit unsigned integer that specifies the size of the *out_attrs[ ]* array. Use of this parameter and its associated *out_attrs[ ]* array is reserved for future use by update trigger servers.

in_attrs[ ]
An array of values of type **sec_attr_t** that specifies the attribute instances to be updated. The size of the array is determined by *num_to_write*.

Output

num_returned
A pointer to an unsigned 32-bit integer that specifies the number of attribute instances returned in the *out_attrs[ ]* array.

out_attrs[ ]
Reserved for future use by update trigger servers.

num_left
A pointer to an unsigned 32-bit integer that supplies the number of attributes that could not be returned because of space constraints in the *out_attrs[ ]* buffer. To ensure that all the attributes will be returned, increase the size of the *out_attrs[ ]* array by increasing the size of *space_avail* and *num_returned*.

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *in_attrs[ ]* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **−1**.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_attr_update()** routine creates new attribute instances and updates existing attribute instances attached to a object specified by name and registry domain. The instances to be created or updated are passed as an array of **sec_attr_t** data types. This is an atomic operation: if the creation of any attribute in the *in_attrs[ ]* array fails, all updates are aborted. The attribute causing the update to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

The *in_attrs[ ]* array, which specifies the attributes to be created, contains values of type **sec_attr_t**. These values are as follows:

attr_id A Universal Unique Identifier (UUID) that identifies the attribute type

attr_value
Values of **sec_attr_value_t** that specify the attribute's encoding type and values.

If an attribute instance already exists which is identical in both *attr_id* and *attr_value* to an attribute specified in *in_attrs[ ]*, the existing attribute information is overwritten by the new information. For multivalued attributes, every instance with the same *attr_id* is overwritten with the supplied values.

If an attribute instance does not exist, it is created.

For multivalued attributes, because every instance of the multivalued attribute is identified by the same UUID, every instance is overwritten with the supplied value. To change only one of the values, you must supply the values that should be unchanged as well as the new value.

To create instances of multivalued attributes, create individual **sec_attr_t** data types to define each multivalued attribute instance and then pass all of them in in the input array.

Permissions Required

The **sec_rgy_attr_update()** routine requires the **U** (**Update**) permission set for each attribute type identified in the *in_attrs*[ ] array. These permissions are defined as part of the access control list (ACL) manager set in the schema entry of each attribute type.

Files

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

unauthorized

database read only

server unavailable

invalid/unsupported attribute type

invalid encoding type

value not unique

attribute instance already exists

trigger server unavailable

site read only

error_status_ok

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_attr_delete(3sec)**, **sec_rgy_attr_test_and_update(3sec)**.

sec_rgy_auth_plcy_get_effective

Purpose

Returns the effective authentication policy for an account

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_auth_plcy_get_effective(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *account
     sec_rgy_plcy_auth_t *auth_policy
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

account
A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. If all three fields contain empty strings, the authentication policy returned is that of the registry.

Output

auth_policy
A pointer to the **sec_rgy_plcy_auth_t** structure to receive the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_auth_plcy_get_effective()** routine returns the effective authentication policy for the specified account. The authentication policy in effect is the more restrictive of the registry and the account policies for each policy category. If no account is specified, the registry's authentication policy is returned.

Permissions Required

The **sec_rgy_auth_plcy_get_effective()** routine requires the **r** (**read**) permission on the policy object from which the data is to be returned. If an account is specified and an account policy exists, the routine also requires the **r** (**read**) permission on the account principal.

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_auth_plcy_get_info(3sec)**, **sec_rgy_auth_plcy_set_info(3sec)**.

sec_rgy_auth_plcy_get_info

Purpose

Returns the authentication policy for an account

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_auth_plcy_get_info(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *account
     sec_rgy_plcy_auth_t *auth_policy
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

account
A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account.

Output

auth_policy
A pointer to the **sec_rgy_plcy_auth_t** structure to receive the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_auth_plcy_get_info()** routine returns the authentication policy for the specified account. If no account is specified, the registry's authentication policy is returned.

Permissions Required

The **sec_rgy_auth_plcy_get_info()** routine requires the **r** (**read**) permission on the policy object or account principal from which the data is to be returned.

Notes

The actual policy in effect will not correspond precisely to what is returned by this call if the overriding registry authentication policy is more restrictive than the policy for the specified account. Use **sec_rgy_auth_plcy_get_effective()** to return the policy currently in effect for the given account.

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No account with the given login name could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_auth_plcy_get_effective(3sec)**, **sec_rgy_auth_plcy_set_info(3sec)**.

sec_rgy_auth_plcy_set_info

Purpose

Sets the authentication policy for an account

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_auth_plcy_set_info(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *account
     sec_rgy_plcy_auth_t *auth_policy
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

account
A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three names must be completely specified.

auth_policy
A pointer to the **sec_rgy_plcy_auth_t** structure containing the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_auth_plcy_set_info()** routine sets the indicated authentication policy for the specified account. If no account is specified, the authentication policy is set for the registry as a whole.

Permissions Required

The **sec_rgy_auth_plcy_set_info()** routine requires the **a** (*auth_info*) permission on the policy object or account principal for which the data is to be set.

Notes

The policy set on an account may be less restrictive than the policy set for the registry as a whole. In this case, the change in policy has no effect, since the

effective policy is the most restrictive combination of the principal and registry authentication policies. (See the **sec_rgy_auth_plcy_get_effective()** routine).

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No account with the given login name could be found.

sec_rgy_not_authorized
The user is not authorized to update the specified record.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_auth_plcy_get_effective(3sec)**, **sec_rgy_auth_plcy_get_info(3sec)**.

sec_rgy_cell_bind

Purpose

Binds to a registry in a cell

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_cell_bind(
     unsigned_char_t *cell_name
     sec_rgy_bind_auth_info_t *auth_info
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

cell_name
A character string (type **unsigned_char_t**) containing the name of the cell in question. Upon return, a security server for that cell is associated with *context*, the registry server handle. The cell must be specified completely and precisely. This routine offers none of the pathname resolving services of **sec_rgy_site_bind()**.

auth_info
A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info()** routine).

Output

context
A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_cell_bind()** routine establishes a relationship with a registry site at an arbitrary level of security. The *cell_name* parameter identifies the target cell.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_bind(3sec)**.

sec_rgy_cursor_reset

Purpose

Resets the registry database cursor

Synopsis

```
#include <dce/misc.h>

void  sec_rgy_cursor_reset(
     sec_rgy_cursor_t *cursor);
```

Parameters

Input/Output

cursor A pointer into the registry database.

Description

The **sec_rgy_cursor_reset()** routine resets the database cursor to return the first suitable entry. A cursor is a pointer into the registry. It serves as a place holder when returning successive items from the registry.

A cursor is bound to a particular server. In other words, a cursor that is in use with one replica of the registry has no meaning for any other replica. If a calling program attempts to use a cursor from one replica with another, the cursor is reset and the routine for which the cursor was specified returns the first item in the database.

A cursor that is in use with one call cannot be used with another. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Files

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

Examples

The following example illustrates use of the cursor within a loop. The initial **sec_rgy_cursor_reset()** call resets the cursor to point to the first item in the registry. Successive calls to **sec_rgy_pgo_get_next()** return the next PGO item and update the cursor to reflect the last item returned. When the end of the list of PGO items is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter.

```
sec_rgy_cursor_reset(&cursor);
do {
    sec_rgy_pgo_get_next(context, domain, scope, &cursor,
                    &item, name &status);
    if (status == error_status_ok) {
         /* Print formatted PGO item info */
    }
}while (status == error_status_ok);
```

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_get_projlist(3sec)**, **sec_rgy_acct_lookup(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_get_members(3sec)**, **sec_rgy_pgo_get_next(3sec)**.

sec_rgy_login_get_effective

Purpose

Returns the effective login data for an account

Synopsis

```
#include <dce/misc.h>

void  sec_rgy_login_get_effective(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_key_t *key_parts
     sec_rgy_sid_t *sid
     sec_rgy_unix_sid_t *unix_sid
     sec_rgy_acct_user_t *user_part
     sec_rgy_acct_admin_t *admin_part
     sec_rgy_plcy_t *policy_data
     signed32 max_number
     signed32 *supplied_number
     uuid_t id_projlist[ ]
     signed32 unix_projlist[ ]
     signed32 *num_projects
     sec_rgy_name_t cell_name
     uuid_t *cell_uuid
     sec_override_fields_t *overridden
     error_status_t *status);
```

Parameters

Input

context
The registry server handle.

max_number
The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist[ ]* arrays.

Input/Output

login_name
A pointer to the account login name. A login name is composed of the names for the account's principal, group, and organization (PGO) items.

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

sid A pointer to a **sec_rgy_sid_t** structure to receive the returned subject identifier (SID) for the account. This structure consists of the UUIDs for the account's PGO items.

unix_sid
A pointer to a **sec_rgy_unix_sid_t** structure to receive the returned UNIX subject identifier (SID) for the account. This structure consists of the UNIX numbers for the account's PGO items.

user_part
A pointer to a **sec_rgy_acct_user_t** structure to receive the returned user data for the account.

admin_part
A pointer to a **sec_rgy_acct_admin_t** structure to receive the returned administrative data for the account.

policy_data
A pointer to a **sec_rgy_policy_t** structure to receive the policy data for the account. The policy data is associated with the account's organization, as identified in the login name.

supplied_number
A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input.

id_projlist[ ]
An array to receive the UUID of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

cell_name
The name of the account's cell.

cell_uuid
The UUID for the account's cell.

overridden
A pointer to a 32-bit set of flags identifying the local overrides, if any, for the account login information.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_login_get_effective()** routine returns effective login information for the specified account. Login information is extracted from the account's entry in the registry database. Effective login information is a combination of the login information from the registry database and any login overrides defined for the account on the local machine.

The *overridden* parameter indicates which, if any, of the following local overrides have been defined for the account:

- The UNIX user ID
- The group ID
- The encrypted password
- The account's miscellaneous information (**gecos**) field

- The account's home directory
- The account's login shell

Local overrides for account login information are defined in the **/etc/passwd_override** file and apply only to the local machine.

Files

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

/etc/passwd_override
The file that defines local overrides for account login information.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy__object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_login_get_info(3sec)**.

Files: **passwd_override(5sec)**.

sec_rgy_login_get_info

Purpose

Returns login information for an account

Synopsis

```
#include <dce/misc.h>

void  sec_rgy_login_get_info(
     sec_rgy_handle_t context
     sec_rgy_login_name_t *login_name
     sec_rgy_acct_key_t *key_parts
     sec_rgy_sid_t *sid
     sec_rgy_unix_sid_t *unix_sid
     sec_rgy_acct_user_t *user_part
     sec_rgy_acct_admin_t *admin_part
     sec_rgy_plcy_t *policy_data
     signed32 max_number
     signed32 *supplied_number
     uuid_t id_projlist[ ]
     signed32 unix_projlist[ ]
     signed32 *num_projects
     sec_rgy_name_t cell_name
     uuid_t *cell_uuid
     error_status_t *status);
```

Parameters

Input

context
The registry server handle.

max_number
The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist[ ]* arrays.

Input/Output

login_name
A pointer to the account login name. A login name is composed of the names for the account's principal, group, and organization (PGO) items.

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

sid A pointer to a **sec_rgy_sid_t** structure to receive the UUID's representing the account's PGO items.

unix_sid
A pointer to a **sec_rgy_unix_sid_t** structure to receive the UNIX numbers for the account's PGO items.

user_part
A pointer to a **sec_rgy_acct_user_t** structure to receive the returned user data for the account.

admin_part
A pointer to a **sec_rgy_acct_admin_t** structure to receive the returned administrative data for the account.

policy_data
A pointer to a **sec_rgy_policy_t** structure to receive the policy data for the account. The policy data is associated with the account's organization, as identified in the login name.

supplied_number
A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input.

id_projlist[ ]
An array to receive the UUID of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

cell_name
The name of the account's cell.

cell_uuid
The UUID for the account's cell.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_login_get_info()** routine returns login information for the specified account. This information is extracted from the account's entry in the registry database. To return any local overrides for the account's login data, use **sec_rgy_login_get_effective()**.

Permissions Required

The **sec_rgy_login_get_info()** routine requires the **r** (**read**) permission on the account principal from which the data is to be returned.

Files

/usr/lib/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_login_get_effective(3sec)**.

sec_rgy_pgo_add

Purpose

Adds a PGO item to the registry database

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_add(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     sec_rgy_pgo_item_t *pgo_item
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

name A pointer to a **sec_rgy_name_t** character string containing the name of the new PGO item.

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure containing the data for the new PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item may have (or belong to) a concurrent group set.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_add()** routine adds a PGO item to the registry database.

The PGO data consists of the following:

- The universal unique identifier (UUID) of the PGO item. Specify NULL to have the registry server create a new UUID for an item.

- The UNIX number for the PGO item. Since the registry uses embedded UNIX IDs (where a subset of the UUID bits represent the UNIX ID), the specified ID must match the UUID, if both are specified.
- The quota for subaccounts allowed for this item entry.
- The full name of the PGO item.
- Flags (in the **sec_rgy_pgo_flags_t** format) indicating whether
  - A principal item is an alias.
  - The PGO item can be deleted from the registry.
  - A principal item can have a concurrent group set.
  - A group item can appear in a concurrent group set.

Permissions Required

The **sec_rgy_pgo_add()** routine requires the **i** (**insert**) permission on the parent directory in which the the PGO item is to be created.

Notes

An account can be added to the registry database only when all its constituent PGO items are already in the database, and the appropriate membership relationships between them are established. For example, to establish an account with principal name **tom**, group name **writers**, and organization name **hp**, all three names must exist as independent PGO items in the database. Furthermore, **tom** must be a member of **writers**, which must be a member of **hp**. (See **sec_rgy_acct_add()** to add an account to the registry.)

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to add the specified PGO item.

sec_rgy_object_exists
A PGO item already exists with the name given in *name*.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_add(3sec)**, **sec_rgy_pgo_delete(3sec)**, **sec_rgy_pgo_rename(3sec)**, **sec_rgy_pgo_replace(3sec)**.

sec_rgy_pgo_add_member

Purpose

Adds a principal to a group or organization

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_add_member(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t go_name
     sec_rgy_name_t principal_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

go_name
A character string (type **sec_rgy_name_t**) containing the name of the group or organization to which the specified principal will be added.

principal_name
A character string (type **sec_rgy_name_t**) containing the name of the principal to be added to the membership list of the group or organization specified by *go_name*. You must use fully qualified names to add foreign principals as members of a group. (Only local principals can be added as members of an organization.)

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_add_member()** routine adds a member to the membership list of a group or organization in the registry database. For this call to succeed when adding a principal from a foreign cell to a group, the Security Server (**secd**) must be running in the foreign cell.

Permissions Required

The **sec_rgy_pgo_add_member()** routine requires the **M** (**Member_list**) permission on the group or organization item specified by *go_name*. If *go_name* specifies a group, the routine also requires the **g** (**groups**) permission on the principal identified by *principal_name*.

Notes

An account can be added to the registry database only when all its constituent PGO items are already in the database, and the appropriate membership relationships between them are established. For example, to establish an account with principal name **tom**, group name **writers**, and organization name **hp**, all three names must exist as independent PGO items in the database. Furthermore, **tom** must be a member of **writers**, which must be a member of **hp**. (See the **sec_rgy_acct_add()** routine to add an account to the registry.)

Files

/usr/include/dce/pgo.idl
: The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
: The client program is not authorized to add members to the specified group or organization.

sec_rgy_bad_domain
: An invalid domain was specified. A member can be added only to a group or organization, not a principal.

sec_rgy_object_not_found
: The registry server could not find the specified name.

sec_rgy_server_unavailable
: The DCE registry server is unavailable.

error_status_ok
: The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_delete_member(3sec)**, **sec_rgy_pgo_get_members(3sec)**, **sec_rgy_pgo_is_member(3sec)**.

sec_rgy_pgo_delete

Purpose

Deletes a PGO item from the registry database

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_delete(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

name A pointer to a **sec_rgy_name_t** character string containing the name of the PGO item to be deleted.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_delete()** routine deletes a PGO item from the registry database. Any account depending on the deleted PGO item is also deleted.

Permissions Required

The **sec_rgy_pgo_delete()** routine requires the following permissions:

- The **d** (**delete**) permission on the parent directory that contains the the PGO item to be deleted.
- The **D** (**Delete_object**) permission on the PGO item itself.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to delete the specified item.

sec_rgy_object_not_found
The registry server could not find the specified item.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**.

sec_rgy_pgo_delete_member

Purpose

Removes a member from a group or organization

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_delete_member(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t go_name
     sec_rgy_name_t principal_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

go_name
A character string (type **sec_rgy_name_t**) containing the name of the group or organization from which the specified principal will be removed.

principal_name
A character string (type **sec_rgy_name_t**) containing the name of the principal to be removed from the membership list of the group or organization specified by *go_name*. You must use fully-qualified names to remove foreign principals from groups. (Only local principals can be members of an organization.)

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_delete_member()** routine removes a member from the membership list of a group or organization. The principal to be removed from a group can be in the local or a foreign cell. (Only local principals can be members of an organization.)

Permissions Required

The **sec_rgy_pgo_delete_member()** routine requires the **M** (**Member_list**) permission on the group or organization item specified by *go_name*.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to remove the specified member.

sec_rgy_bad_domain
An invalid domain was specified. Members can exist only for groups and organizations, not for principals.

sec_rgy_object_not_found
The specified group or organization was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_add_member**.

sec_rgy_pgo_get_by_eff_unix_num

Purpose

Returns the name and data for a PGO item identified by its effective UNIX number

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_by_eff_unix_num(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t scope
     signed32 unix_id
     boolean32 allow_aliases
     sec_rgy_cursor_t *item_cursor
     sec_rgy_pgo_item_t *pgo_item
     sec_rgy_name_t name
     boolean32 *overridden
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UNIX number identifies a principal.

sec_rgy_domain_group
The UNIX number identifies a group.

Note that this function does *not* support the value **sec_rgy_domain_org**.

scope A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

unix_id
The UNIX number of the desired registry PGO item.

allow_aliases
A **boolean32** value indicating whether to search for a primary PGO item, or

whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_pgo_get_next()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set. The data is as it appears in the registry, for that UNIX number, even though some of the fields may have been overridden locally.

name A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item. This string might contain a local override value if the supplied UNIX number is found in the **passwd_override** or **group_override** file.

overridden
A pointer to a **boolean32** value indicating whether or not the supplied UNIX number has an entry in the local override file (**passwd_override** or **group_override**).

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_by_eff_unix_num()** routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UNIX number.

This routine is similar to the **sec_rgy_pgo_get_by_unix_num()** routine. The difference between the routines is that **sec_rgy_pgo_get_by_eff_unix_num()** first searches the local override files for the respective *name_domain* for a match with the supplied UNIX number. If an override match is found, and an account or group name is found in that entry, then that name is used to obtain PGO data from the registry and the value of the *overridden* parameter is set to TRUE.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The **sec_rgy_pgo_get_by_eff_unix_num()** routine requires the **r** (**read**) permission on the PGO item to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

group_override
The local group override file.

passwd_override
The local password override file.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_get_next(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_get_by_id

Purpose

Returns the name and data for a PGO item identified by its UUID

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_by_id(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t scope
     uuid_t *item_id
     boolean32 allow_aliases
     sec_rgy_cursor_t *item_cursor
     sec_rgy_pgo_item_t *pgo_item
     sec_rgy_name_t name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UUID identifies a principal.

sec_rgy_domain_group
The UUID identifies a group.

sec_rgy_domain_org
The UUID identifies an organization.

scope A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

item_id
A pointer to the **uuid_t** variable containing the UUID (Unique Universal Identifier) of the desired PGO item.

allow_aliases
A **boolean32** value indicating whether to search for a primary PGO item, or

whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_pgo_get_by_id()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_by_id()** routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UUID.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The **sec_rgy_pgo_get_by_id()** routine requires the **r** (**read**) permission on the PGO item to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_get_next(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_get_by_name

Purpose

Returns the data for a named PGO item

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_by_name(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t pgo_name
     sec_rgy_cursor_t *item_cursor
     sec_rgy_pgo_item_t *pgo_item
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization to search for.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_pgo_get_by_name()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_by_name()** routine returns the data for a named PGO item from the registry database. The desired item is identified by its type (*name_domain*) and name.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The **sec_rgy_pgo_get_by_name()** routine requires the **r** (**read**) permission on the PGO item to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_get_next(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_get_by_unix_num

Purpose

Returns the name and data for a PGO item identified by its UNIX ID

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_by_unix_num(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t scope
     signed32 unix_id
     boolean32 allow_aliases
     sec_rgy_cursor_t *item_cursor
     sec_rgy_pgo_item_t *pgo_item
     sec_rgy_name_t name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UNIX number identifies a principal.

sec_rgy_domain_group
The UNIX number identifies a group.

sec_rgy_domain_org
The UNIX number identifies an organization.

scope A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

unix_id
The UNIX number of the desired registry PGO item.

allow_aliases
A **boolean32** value indicating whether to search for a primary PGO item, or

whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_pgo_get_by_unix_num()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_by_unix_num()** routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UNIX number.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The **sec_rgy_pgo_get_by_unix_num()** routine requires the **r** (**read**) permission on the PGO item to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_next(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_get_members

Purpose

Returns the membership list for a specified group or organization or returns the set of groups in which the specified principal is a member

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_members(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t go_name
     sec_rgy_cursor_t *member_cursor
     signed32 max_members
     sec_rgy_member_t member_list[ ]
     signed32 *number_supplied
     signed32 *number_members
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a **secd** server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable specifies whether *go_name* identifies a principal, group, or organization. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

sec_rgy_domain_person
The *go_name* parameter identifies an principal.

go_name
A character string (type **sec_rgy_name_t**) that contains the name of a group, organization, or principal. If *go_name* is the name of a group or organization, the call returns the group's or organization's member list. If *go_name* is the name of a principal, the call returns a list of all groups in which the principal is a member. (Contrast this with the **sec_rgy_acct_get_proj** call, which returns only those groups in which the principal is a member and that have been marked to be included in the principal's project list.)

max_members
A **signed32** variable containing the allocated dimension of the *member_list[ ]* array. This is the maximum number of members or groups that can be returned by a single call.

Input/Output

member_cursor
An opaque pointer to a specific entry in the membership list or list of groups. The returned list begins with the entry specified by *member_cursor*. Upon return, the cursor points to the next entry after the last one returned. If there are no more entries, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor to the beginning of the list.

Output

member_list[ ]
An array of character strings to receive the returned member or group names. The size allocated for the array is given by *max_number*. If this value is less than the total number of members or group names, multiple calls must be made to return all of the members or groups. For groups, fully qualified names are returned for foreign principals and local names for local principals. (Only local principals can be members of an organization.)

number_supplied
A pointer to a **signed32** variable to receive the number of members or groups actually returned in *member_list[ ]*.

number_members
A pointer to a **signed32** variable to receive the total number of members or groups. If this number is greater than *number_supplied*, multiple calls to **sec_rgy_pgo_get_members()** are necessary. Use the *member_cursor* parameter to coordinate successive calls.

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_members()** routine returns a list of the members in the specified group or organization, or a list of groups in which a specified principal is a member.

The *member_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates *member_cursor* to point to the next member or group (if any) after the returned list. If not all of the members or groups are returned, the updated cursor can be supplied on successive calls to return the remainder of the list.

Permissions Required

The **sec_rgy_pgo_get_members()** routine requires the **r** (**read**) permission on the group, organization, or principal object specified by *go_name*.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example,

you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Return Values

The routine returns

- The names of the groups or members in *member_list[ ]*
- The number of members or groups returned by the call in *number_supplied*
- The total number of members in the group or organization, or the total number of groups in which the principal is a member in *number_members*

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor points to the end of the membership list for a group or organization or to the end of the list of groups for a principal.

sec_rgy_object_not_found
The specified group, organization, or principal could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_acct_get_proj(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add_member(3sec)**, **sec_rgy_pgo_is_member(3sec)**.

sec_rgy_pgo_get_next

Purpose

Returns the next PGO item in the registry database

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_get_next(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t scope
     sec_rgy_cursor_t *item_cursor
     sec_rgy_pgo_item_t *pgo_item
     sec_rgy_name_t name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
Returns the next principal item.

sec_rgy_domain_group
Returns the next group item.

sec_rgy_domain_org
Returns the next organization item.

scope A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_pgo_get_next()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use **sec_rgy_cursor_reset()** to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name A pointer to a **sec_rgy_name_t** character string containing the name of the returned PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_get_next()** routine returns the data and name for the PGO in the registry database indicated by *item_cursor*. It also advances the cursor to point to the next PGO item in the database. Successive calls to this routine return all the PGO items in the database of the specified type (given by *name_domain*), in storage order.

The PGO data consists of the following:

- The universal unique identifier (UUID) of the PGO item.
- The UNIX number for the PGO item.
- The quota for subaccounts.
- The full name of the PGO item.
- Flags indicating whether
  - A principal item is an alias.
  - The PGO item can be deleted.
  - A principal item can have a concurrent group set.
  - A group item can appear on a concurrent group set.

Permissions Required

The **sec_rgy_pgo_get_next()** routine requires the **r** (**read**) permission on the PGO item to be viewed.

Cautions

There are several different types of cursors used in the registry application programmer interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to **sec_rgy_acct_get_projlist()** and **sec_rgy_pgo_get_next()**. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use **sec_rgy_cursor_reset()** to renew a cursor for use with another call or for another server.

Return Values

The routine returns the data for the returned PGO item in *pgo_item* and the name in *name*.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cursor_reset(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_id_to_name

Purpose

Returns the name for a PGO item identified by its UUID

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_id_to_name(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     uuid_t *item_id
     sec_rgy_name_t pgo_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_id* parameter identifies a group.

sec_rgy_domain_org
The *item_id* parameter identifies an organization.

item_id
A pointer to the **uuid_t** variable containing the input UUID (unique universal identifier).

Output

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization with the input UUID.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_id_to_name()** routine returns the name of the PGO item having the specified UUID.

Permissions Required

The **sec_rgy_pgo_id_to_name()** routine requires at least one permission of any kind on the PGO item to be viewed.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_id_to_unix_num

Purpose

Returns the UNIX number for a PGO item identified by its UUID

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_id_to_unix_num(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     uuid_t *item_id
     signed32 *item_unix_id
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_id* parameter identifies a group.

sec_rgy_domain_org
The *item_id* parameter identifies an organization.

item_id
A pointer to the **uuid_t** variable containing the input UUID (unique universal identifier).

Output

item_unix_id
A pointer to the **signed32** variable to receive the returned UNIX number for the PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_id_to_unix_num()** routine returns the UNIX number for the PGO item having the specified UUID.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_is_member

Purpose

Checks group or organization membership

Synopsis

```
#include <dce/pgo.h>

boolean32  sec_rgy_pgo_is_member(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t go_name
     sec_rgy_name_t principal_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

go_name
A character string (type **sec_rgy_name_t**) containing the name of the group or organization whose membership list is in question.

principal_name
A character string (type **sec_rgy_name_t**) containing the name of the principal whose membership in the group or organization specified by *go_name* is in question. For groups, use fully-qualified names for foreign principals. (Only local principals can be members of an organization.)

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_is_member()** routine tests whether the specified principal is a member of the named group or organization.

Permissions Required

The **sec_rgy_pgo_is_member()** routine requires the **t** (**test**) permission on the group or organization item specified by *go_name*.

Return Values

The routine returns TRUE if the principal is a member of the named group or organization. If the principal is not a member, the routine returns FALSE.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The named group or organization was not found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add_member(3sec)**, **sec_rgy_pgo_get_members(3sec)**.

sec_rgy_pgo_name_to_id

Purpose

Returns the UUID for a named PGO item

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_name_to_id(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t pgo_name
     uuid_t *item_id
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization whose UUID is desired.

Output

item_id
A pointer to the **uuid_t** variable containing the UUID (unique universal identifier) of the resulting PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_name_to_id()** routine returns the UUID associated with the named PGO item.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The specified PGO item could not be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_name_to_unix_num

Purpose

Returns the UNIX number for a PGO item identified by its name

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_name_to_unix_num(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t pgo_name
     signed32 *item_unix_id
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string containing the name of the PGO item in question.

Output

item_unix_id
A pointer to the **signed32** variable to receive the returned UNIX number for the PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_name_to_unix_num()** routine returns the UNIX number for the PGO item having the specified name.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**, **sec_rgy_pgo_unix_num_to_id(3sec)**.

sec_rgy_pgo_rename

Purpose

Changes the name of a PGO item in the registry database

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_rename(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t old_name
     sec_rgy_name_t new_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

old_name
A pointer to a **sec_rgy_name_t** character string containing the existing name of the PGO item.

new_name
A pointer to a **sec_rgy_name_t** character string containing the new name for the PGO item.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_rename()** routine renames a PGO item in the registry database.

Permissions Required

If the **sec_rgy_pgo_rename()** routine is performing a rename within a directory, it requires the **n** (**name**) permission on the old name of the PGO item. If the routine is performing a move between directories, it requires the following permissions:

- The **d** (**delete**) permission on the parent directory that contains the PGO item.
- The **n** (**name**) permission on the old name of the PGO item.
- The **i** (**insert**) permission on the parent directory in which the PGO item is to be added under the new name.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to change the name of the specified PGO item.

sec_rgy_object_not_found
The registry server could not find the specified PGO item.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_replace(3sec)**.

sec_rgy_pgo_replace

Purpose

Replaces the data in an existing PGO item

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_replace(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     sec_rgy_name_t pgo_name
     sec_rgy_pgo_item_t *pgo_item
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization whose data is to be replaced.

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure containing the new data for the PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_replace()** routine replaces the data associated with a PGO item in the registry database.

The UNIX ID and UUID of a PGO item cannot be replaced. To change the UNIX ID or UUID, the existing PGO item must be deleted and a new PGO item added in its

place. The one exception to this rule is that the UNIX ID can be replaced in the PGO item for a cell principal. The reason for this exception is that the UUID for a cell principal does not contain an embedded UNIX ID.

Permissions Required

The **sec_rgy_pgo_replace()** routine requires at least one of the following permissions:

- The **m** (**mgmt_info**) permission on the PGO item, if **quota** or **flags** is being set.
- The **f** (**fullname**) permission on the PGO item, if **fullname** is being set.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The client program is not authorized to replace the specified PGO item.

sec_rgy_object_not_found
No PGO item was found with the given name.

sec_rgy_unix_id_changed
The UNIX number of the PGO item was changed.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_delete(3sec)**, **sec_rgy_pgo_rename(3sec)**.

sec_rgy_pgo_unix_num_to_id

Purpose

Returns the UUID for a PGO item identified by its UNIX number

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_unix_num_to_id(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     signed32 item_unix_id
     uuid_t *item_id
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_unix_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_unix_id* parameter identifies a group.

sec_rgy_domain_org
The *item_unix_id* parameter identifies an organization.

item_unix_id
The **signed32** variable containing the UNIX number for the PGO item.

Output

item_id
A pointer to the **uuid_t** variable containing the UUID (unique universal identifier) of the resulting PGO item.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_unix_num_to_id()** routine returns the universal unique identifier (UUID) for a PGO item that has the specified UNIX number.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No item with the specified UNIX number could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**.

sec_rgy_pgo_unix_num_to_name

Purpose

Returns the name for a PGO item identified by its UNIX number

Synopsis

```
#include <dce/pgo.h>

void  sec_rgy_pgo_unix_num_to_name(
     sec_rgy_handle_t context
     sec_rgy_domain_t name_domain
     signed32 item_unix_id
     sec_rgy_name_t pgo_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name_domain
The type of the principal, group, or organization (PGO) item identified by *item_unix_id*. Valid values are as follows:

sec_rgy_domain_person
The *item_unix_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_unix_id* parameter identifies a group.

sec_rgy_domain_org
The *item_unix_id* parameter identifies an organization.

item_unix_id
The **signed32** variable containing the UNIX number for the PGO item.

Output

pgo_name
A character string containing the name of the PGO item in question.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_pgo_unix_num_to_name()** routine returns the name for a PGO item that has the specified UNIX number.

Permissions Required

The **sec_rgy_pgo_unix_num_to_name()** routine requires at least one permission of any kind on the PGO item identified by *item_unix_id*.

Files

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
No item with the specified UNIX number could be found.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_pgo_add(3sec)**, **sec_rgy_pgo_get_by_id(3sec)**, **sec_rgy_pgo_get_by_name(3sec)**, **sec_rgy_pgo_get_by_unix_num(3sec)**, **sec_rgy_pgo_id_to_name(3sec)**, **sec_rgy_pgo_id_to_unix_num(3sec)**, **sec_rgy_pgo_name_to_id(3sec)**.

sec_rgy_plcy_get_effective

Purpose

Returns the effective policy for an organization

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_plcy_get_effective(
     sec_rgy_handle_t context
     sec_rgy_name_t organization
     sec_rgy_plcy_t *policy_data
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

organization
A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine returns the registry's policy data.

Output

policy_data
A pointer to the **sec_rgy_plcy_t** structure to receive the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_plcy_get_effective()** routine returns the effective policy for the specified organization.

The effective policy data is the most restrictive combination of the registry and the organization policies.

The policy data consists of the following:

- The password expiration date. This is the date on which account passwords will expire.
- The minimum length allowed for account passwords.
- The period of time (life span) for which account passwords will be valid.
- The period of time (life span) for which accounts will be valid.

- Flags indicating whether account passwords can consist entirely of spaces or entirely of alphanumeric characters.

Permissions Required

The **sec_rgy_plcy_get_effective()** routine requires the **r** (**read**) permission on the policy object from which the data is to be returned. If an organization is specified, the routine also requires the **r** (**read**) permission on the organization.

Notes

If no organization is specified, the routine returns the registry's policy data. To return the effective policy, an organization must be specified. This is because the routine compares the registry's policy data with that of the organization to determine which is more restrictive.

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The registry server could not find the specified organization.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_plcy_get_info(3sec)**, **sec_rgy_plcy_set_info(3sec)**.

sec_rgy_plcy_get_info

Purpose

Returns the policy for an organization

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_plcy_get_info(
     sec_rgy_handle_t context
     sec_rgy_name_t organization
     sec_rgy_plcy_t *policy_data
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

organization
A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine returns the registry's policy data.

Output

policy_data
A pointer to the **sec_rgy_plcy_t** structure to receive the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_plcy_get_info()** routine returns the policy data for the specified organization. If no organization is specified, the registry's policy data is returned.

The policy data consists of the following:

- The password expiration date. This is the date on which account passwords will expire.
- The minimum length allowed for account passwords.
- The period of time (life span) for which account passwords will be valid.
- The period of time (life span) for which accounts will be valid.
- Flags indicating whether account passwords can consist entirely of spaces or entirely of alphanumeric characters.

Permissions Required

The **sec_rgy_plcy_get_info()** routine requires the **r** (**read**) permission on the policy object or organization from which the data is to be returned.

Notes

The returned policy may not be in effect if the overriding registry authorization policy is more restrictive. (See the **sec_rgy_auth_plcy_get_effective()** routine.)

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_object_not_found
The registry server could not find the specified organization or the organization exists, but policy has not been set for the the specified organizaton.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_plcy_get_effective_info(3sec)**, **sec_rgy_plcy_set_info(3sec)**.

sec_rgy_plcy_set_info

Purpose

Sets the policy for an organization

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_plcy_set_info(
     sec_rgy_handle_t context
     sec_rgy_name_t organization
     sec_rgy_plcy_t *policy_data
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

organization
A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine sets the registry's policy data.

policy_data
A pointer to the **sec_rgy_plcy_t** structure containing the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_plcy_set_info()** routine sets the authentication policy for a specified organization. If no organization is specified, the registry's policy data is set.

Policy data can be returned or set for individual organizations and for the registry as a whole.

Permissions Required

The **sec_rgy_plcy_set_info()** routine requires the **m** (**mgmt_info**) permission on the policy object or organization for which the data is to be set.

Notes

The policy set on an account may be less restrictive than the policy set for the registry as a whole. In this case, the changes in policy have no effect, since the effective policy is the most restrictive combination of the organization and registry authentication policies. (See the **sec_rgy_auth_plcy_get_effective()** routine.)

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The user is not authorized to perform this operation.

sec_rgy_object_not_found
The registry server could not find the specified organization.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_plcy_get_effective(3sec)**, **sec_rgy_plcy_get_info(3sec)**.

sec_rgy_properties_get_info

Purpose

Returns registry properties

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_properties_get_info(
     sec_rgy_handle_t context
     sec_rgy_properties_t *properties
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

Output

properties
A pointer to a **sec_rgy_properties_t** structure to receive the returned property information. A registry's property information contains information such as the default and minimum lifetime and other restrictions on privilege attribute certificates, the realm authentication name, and whether or not this replica of the registry supports updates.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_properties_get_info()** routine returns a list of the registry properties.

The property information consists of the following:

read_version
A stamp specifying the earliest version of the registry server software that can read from this registry.

write_version
A stamp specifying the earliest version of the registry server software that can write to this registry.

minimum_ticket_lifetime
The minimum period of time for which an authentication ticket remains valid.

default_certificate_lifetime
The default period of time for which an authentication certificate (ticket-granting ticket) remains valid. A process can request an authentication certificate with a longer lifetime. Note that the maximum lifetime for an authentication certificate cannot exceed the lifetime established by the effective policy for the requesting account.

low_unix_id_person
The lowest UNIX ID that can be assigned to a principal in the registry.

low_unix_id_group
The lowest UNIX ID that can be assigned to a group in the registry.

low_unix_id_org
The lowest UNIX ID that can be assigned to an organization in the registry.

max_unix_id
The maximum UNIX ID that can be used for any item in the registry.

realm A character string naming the cell controlled by this registry.

realm_uuid
The UUID of the cell controlled by this registry.

flags Flags include the following:

sec_rgy_prop_readonly
If TRUE, the registry database is read-only.

sec_rgy_prop_auth_cert_unbound
If TRUE, privilege attribute certificates can be generated for use at any site.

sec_rgy_prop_shadow_password
If FALSE, passwords can be distributed over the network. If this flag is TRUE, passwords will be stripped from the returned data to the **sec_rgy_acct_lookup()**, and other calls that return an account's encoded password.

sec_rgy_prop_embedded_unix_id
All registry UUIDs contain embedded UNIX IDs. This implies that the UNIX ID of any registry object cannot be changed, since UUIDs are immutable.

Permissions Required

The **sec_rgy_properties_get_info()** routine requires the **r** (**read**) permission on the policy object from which the property information is to be returned.

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_properties_set_info(3sec)**.

sec_rgy_properties_set_info

Purpose

Sets registry properties

Synopsis

```
#include <dce/policy.h>

void  sec_rgy_properties_set_info(
     sec_rgy_handle_t context
     sec_rgy_properties_t *properties
     error_status_t *status);
```

Parameters

Input

context
The registry server handle. An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

properties
A pointer to a **sec_rgy_properties_t** structure containing the registry property information to be set. A registry's property information contains information such as the default and minimum lifetime and other restrictions on privilege attribute certificates, the realm authentication name, and whether or not this replica of the registry supports updates.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_properties_set_info()** routine sets the registry properties.

The property information consists of the following:

read_version
A stamp specifying the earliest version of the registry server software that can read from this registry.

write_version
A stamp specifying the earliest version of the registry server software that can write to this registry.

minimum_ticket_lifetime
The minimum period of time for which an authentication ticket remains valid.

default_certificate_lifetime
The default period of time for which an authentication certificate (ticket-granting ticket) remains valid. A process can request an authentication certificate with a longer lifetime. Note that the maximum lifetime for an authentication certificate cannot exceed the lifetime established by the effective policy for the requesting account.

low_unix_id_person
The lowest UNIX ID that can be assigned to a principal in the registry.

low_unix_id_group
The lowest UNIX ID that can be assigned to a group in the registry.

low_unix_id_org
The lowest UNIX ID that can be assigned to an organization in the registry.

max_unix_id
The maximum UNIX ID that can be used for any item in the registry.

realm A character string naming the cell controlled by this registry.

realm_uuid
The UUID of the cell controlled by this registry.

flags Flags include the following:

sec_rgy_prop_readonly
If TRUE, the registry database is read-only.

sec_rgy_prop_auth_cert_unbound
If TRUE, privilege attribute certificates can be generated for use at any site.

sec_rgy_prop_shadow_password
If FALSE, passwords can be distributed over the network. If this flag is TRUE, passwords will be stripped from the returned data to the **sec_rgy_acct_lookup()**, and other calls that return an account's encoded password.

sec_rgy_prop_embedded_unix_id
All registry UUIDs contain embedded UNIX IDs. This implies that the UNIX ID of any registry object cannot be changed, since UUIDs are immutable.

Permissions Required

The **sec_rgy_properties_set_info()** routine requires the **m** (**mgmt_info**) permission on the policy object for which the property information is to be set.

Files

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_not_authorized
The user is not authorized to change the registry properties.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_properties_get_info(3sec)**.

sec_rgy_site_bind

Purpose

Binds to a registry site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_bind(
     unsigned_char_t *site_name
     sec_rgy_bind_auth_info_t *auth_info
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) specifying the security server to bind to in one of the following forms:

- To bind to an arbitrary security server site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To bind to a specific security server site in a specific cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2)** or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine binds to an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info(3rpc)** reference page.) If the **sec_rgy_bind_auth_info_t** structure specifies authenticated RPC, the caller must have established a valid network identity for this call to succeed.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (''bound to'') the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_bind()** call binds to a registry site at the security level specified by the *auth_info* parameter. The *site_name* parameter identifies the registry to use. If *site_name* is NULL, or a zero-length string, a registry site in the local cell is selected by the client agent.

Note:

Like the **sec_rgy_site_bind_query()** routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use **sec_rgy_site_bind_update()**.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_cell_bind(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_site_bind_query

Purpose

Binds to a registry query site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_bind_query(
     unsigned_char_t *site_name
     sec_rgy_bind_auth_info_t *auth_info
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) specifying the security server to bind to in one of the following forms:

- To bind to an arbitrary security server site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To bind to a specific security server site in a specific cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2)** or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine binds to an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info(3rpc)** reference page.) If the **sec_rgy_bind_auth_info_t** structure specifies authenticated RPC, the caller must have established a valid network identity for this call to succeed.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (''bound to'') the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_bind_query()** routine binds to a registry query site at an arbitrary level of security. A registry query site is a satellite server that operates on a periodically updated copy of the main registry database. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies the query site to use. If *site_name* is NULL, or a zero-length string, a query site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*.

Note:

Like **sec_rgy_bind_open()** routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use **sec_rgy_site_bind_update()**.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_bind(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_site_bind_update

Purpose

Binds to a registry update site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_bind_update(
     unsigned_char_t *site_name
     sec_rgy_bind_auth_info_t *auth_info
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) containing the name of the security server to bind to. Supply this name in any of the following forms:

- To bind to the update site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To start the search for the update site at a specific replica in the replica's cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine starts the search at an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info(3rpc)** reference page.) If the **sec_rgy_bind_auth_info_t** structure specifies authenticated RPC, the caller must have established a valid network identity for this call to succeed.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_bind_update()** routine binds to a registry update site. A registry update site is a master server that may control several satellite (query) servers. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies either the cell in which to find the update site or the replica at which to start the search for the update site. If *site_name* is NULL, or a zero-length string, an update site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*. The handle is to an update site. Use this registry context handle in subsequent calls that update or query the the registry database (for example, the **sec_rgy_pgo_add()** or **sec_rgy_acct_lookup()** calls).

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_bind(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_site_binding_get_info

Purpose

Returns information from the registry binding handle

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_binding_get_info(
     sec_rgy_handle_t context
     unsigned_char_t **cell_name
     unsigned_char_t **server_name
     unsigned_char_t **string_binding
     sec_rgy_bind_auth_info_t *auth_info
     error_status_t *status);
```

Parameters

Input

context
A **sec_rgy_handle_t** variable that contains a registry server handle indicating (bound to) the desired registry site. To obtain information on the default binding handle, initialize *context* to **sec_rgy_default_handle**. A valid login context must be set for the process if *context* is set to **sec_rgy_default_handle**; otherwise the error **sec_under_login_s_no_current_context** is returned.

Output

cell_name
The name of the home cell for this registry.

server_name
The name of the node on which the server is resident. This name is either a global name or a network address, depending on the form in which the name was input to the call that bound to the site.

string_binding
A string containing binding information from **sec_rgy_handle_t**.

auth_info
A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the **rpc_binding_set_auth_info()** routine).

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_binding_get_info()** routine returns the site name and authentication information associated with the *context* parameter. If the context is the default context, the information for the default binding is returned. Passing in a NULL value for any of the output values (except for *status*) will prevent that value from being returned. Memory is allocated for the string returned in the *cell_name*,

server_name, and *string_binding* parameters. The application calls the **rpc_string_free()** routine to deallocate that memory.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_under_login_s_no_current_context

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_bind(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_site_close

Purpose

Frees the binding handle for a registry server

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_close(
     sec_rgy_handle_t context
     error_status_t *status);
```

Parameters

Input

context
An opaque handle indicating (bound to) a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

Output

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_close()** routine frees the memory occupied by the specified handle and destroys its binding with the registry server.

Notes

A handle cannot be used after it is freed.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_get(3sec)**, **sec_rgy_site_is_readonly(3sec)**, **sec_rgy_site_open(3sec)**, **sec_rgy_site_open_query(3sec)**, **sec_rgy_site_open_update(3sec)**.

sec_rgy_site_get

Purpose

Returns the string representation for a bound registry site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_get(
     sec_rgy_handle_t context
     unsigned_char_t **site_name
     error_status_t *status);
```

Parameters

Input

context
An opaque handle indicating (bound to) a registry server. Use **sec_rgy_site_open()** to acquire a bound handle. To obtain information on the default binding handle, initialize *context* to **sec_rgy_default_handle**. A valid login context must be set for the process if *context* is set to **sec_rgy_default_handle** ; otherwise the error **sec_under_login_s_no_current_context** is returned.

Output

site_name
A pointer to a character string (type **unsigned_char_t**) containing the returned name of the registry site associated with *context*, the given registry server handle.

The name is either a global name or a network address, depending on the form in which the name was input to the call that bound to the site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_get()** routine returns the name of the registry site associated with the specified handle. If the handle is the default context, the routine returns the name of the default context's site. Memory is allocated for the string returned in the *site_name* parameter. The application calls the **rpc_string_free()** routine to deallocate that memory.

Notes

To obtain binding information, the use of the **sec_rgy_site_binding_get_info()** call is recommended in place of this call.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_under_login_s_no_current_context

sec_rgy_server_unavailable
The requested registry server is not available.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_open(3sec)**.

sec_rgy_site_is_readonly

Purpose

Checks whether a registry site is read-only

Synopsis

```
#include <dce/binding.h>

boolean32  sec_rgy_site_is_readonly(
     sec_rgy_handle_t context);
```

Parameters

Input

context
An opaque handle indicating (bound to) a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

Description

The **sec_rgy_site_is_readonly()** routine checks whether the registry site associated with the specified handle is a query site or an update site. A query site is a read-only replica of a master registry database. The update site accepts changes to the registry database, and duplicates the changes in its associated query sites.

Return Values

The routine returns

- TRUE, if the registry site is read-only or if there was an error using the specified handle
- FALSE, if the registry site is an update site

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_open(3sec)**, **sec_rgy_site_open_query(3sec)**.

sec_rgy_site_open

Purpose

Binds to a registry site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_open(
     unsigned_char_t *site_name
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) specifying the security server to bind to in one of the following forms:

- To bind to an arbitrary security server site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To bind to a specific security server site in a specific cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine binds to an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_open()** routine binds to a registry site at the level of security specified in the **rpc_binding_set_auth_info()** call. The *site_name* parameter identifies the registry to use. If *site_name* is NULL, or a zero-length string, a registry site in the local cell is selected by the client agent. The caller must have established a valid network identity for this call to succeed.

Note:

To bind to a registry site, the use of the **sec_rgy_site_bind()** call is recommended in place of this call.

Like **sec_rgy_site_open_query()** routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use **sec_rgy_site_open_update()**.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The requested registry server is not available.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_close(3sec)**, **sec_rgy_site_is_readonly(3sec)**, **sec_rgy_site_open_query(3sec)**, **sec_rgy_site_open_update(3sec)**.

sec_rgy_site_open_query

Purpose

Binds to a registry query site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_open_query(
     unsigned_char_t *site_name
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) specifying the security server to bind to in one of the following forms:

- To bind to an arbitrary security server site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To bind to a specific security server site in a specific cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine binds to an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_open_query()** routine binds to a registry query site. A registry query site is a satellite server that operates on a periodically updated copy of the main registry database. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies the query site to use. If *site_name* is NULL, or a zero-length string, a query site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*.

The caller must have established a valid network identity for this call to succeed.

Note:

To bind to a registry query site, the use of the **sec_rgy_site_bind_query()** call is recommended in place of this call.

Like **sec_rgy_site_open()** routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use **sec_rgy_site_open_update()**.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_close(3sec)**, **sec_rgy_site_get(3sec)**, **sec_rgy_site_is_readonly(3sec)**, **sec_rgy_site_open(3sec)**, **sec_rgy_site_open_update(3sec)**.

sec_rgy_site_open_update

Purpose

Binds to a registry update site

Synopsis

```
#include <dce/binding.h>

void  sec_rgy_site_open_update(
     unsigned_char_t *site_name
     sec_rgy_handle_t *context
     error_status_t *status);
```

Parameters

Input

site_name

A character string (type **unsigned_char_t**) specifying the security server to bind to in one of the following forms:

- To bind to the update site in a named cell, specify a cell name (for example, **/.../r_d.com**) or **/.:** for the local cell.
- To start the search for the update site at a specific replica in the replica's cell, specify either the cell name and the server name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the server's network address (for example, **ncadg_ip_udp:15.22.144.248**). If the server name is not valid, the routine binds to an arbitrary security site in the named cell.

Note that the routine ignores anything after the cell name that does not refer to an item in the Cell Directory Service (CDS) namespace. If the specified CDS namespace item does not resolve to a security server, the call fails.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_site_open_update()** routine binds to a registry update site. A registry update site is a master server that may control several satellite (query) servers. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies either the cell in which to find the update site or the replica at which to start the search for the update site. If *site_name* is NULL, or a zero-length string, an update site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*. The handle is to an update site. Use this registry context handle in subsequent calls that update or query the the registry database (for example, the **sec_rgy_pgo_add()** or **sec_rgy_acct_lookup()** calls). The caller must have established a valid network identity for this call to succeed.

Note:

To bind to a registry update site, the use of the **sec_rgy_site_bind_update()** call is recommended in place of this call.

Files

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**, **sec_rgy_site_close(3sec)**, **sec_rgy_site_get(3sec)**, **sec_rgy_site_is_readonly(3sec)**, **sec_rgy_site_open(3sec)**, **sec_rgy_site_open_query(3sec)**.

sec_rgy_unix_getgrgid

Purpose

Returns a UNIX style group entry for the account matching the specified group ID

Synopsis

```
#include <dce/rgynbase.h>

void  sec_rgy_unix_getgrgid(
     sec_rgy_handle_t context
     signed32 gid
     signed32 max_number
     sec_rgy_cursor_t *item_cursor
     sec_rgy_unix_group_t *group_entry
     signed32 *number_members
     sec_rgy_member_t member_list[ ]
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

gid A 32-bit integer specifying the group ID to match.

max_number
The maximum number of members to be returned by the call. This must be no larger than the allocated size of the *member_list[ ]* array.

Input/Output

item_cursor
An opaque pointer indicating a specific principal, group, and organization (PGO) item entry in the registry database. The **sec_rgy_unix_getgrgid()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use **sec_rgy_cursor_reset()** to refresh the cursor.

Output

group_entry
A UNIX style group entry structure returned with information about the account matching *gid*.

number_members
A signed 32-bit integer containing the total number of member names returned in the *member_list[ ]* array.

member_list[ ]
An array of character strings to receive the returned member names. The size allocated for the array is given by *max_number*. If this value is less than the total number of members in the membership list, multiple calls must be made to return all of the members.

status On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_unix_getgrgid()** routine returns the next UNIX group structure that matches the input UNIX group ID. The structure is in the following form:

```
typedef struct {
   sec_rgy_name_t name;
   signed32 gid;
   sec_rgy_member_buf_t members;
}         sec_rgy_unix_group_t;
```

The structure includes the following:

- The name of the group
- The group's UNIX ID
- A string containing the names of the group members. This string is limited in size by the size of the **sec_rgy_member_buf_t** type defined in **rgynbase.idl**.

The routine also returns an array of member names, limited in size by the *number_members* parameter.

This call is supplied in source code form.

Files

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgybase.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_nomore_entries
The end of the list of entries has been reached.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_unix_getgrnam

Purpose

Returns a UNIX style group entry for the account matching the specified group name

Synopsis

```
#include <dce/rgynbase.h>

void  sec_rgy_unix_getgrnam(
     sec_rgy_handle_t context
     sec_rgy_name_t name
     signed32 name_length
     signed32 max_num_members
     sec_rgy_cursor_t *item_cursor
     sec_rgy_unix_group_t *group_entry
     signed32 *number_members
     sec_rgy_member_t member_list[ ]
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

name A character string (of type **sec_rgy_name_t**) specifying the group name to be matched.

name_length
An signed 32-bit integer specifying the length of *name* in characters.

max_num_members
The maximum number of members to be returned by the call. This must be no larger than the allocated size of the *member_list[ ]* array.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_unix_getgrnam()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries** . Use **sec_rgy_cursor_reset()** to refresh the cursor.

Output

group_entry
A UNIX style group entry structure returned with information about the account matching *name*.

number_members
An signed 32-bit integer containing the total number of member names returned in the *member_list[ ]* array.

member_list[ ]
An array of character strings to receive the returned member names. The size allocated for the array is given by *max_number*. If this value is less than the total number of members in the membership list, multiple calls must be made to return all of the members.

status On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_unix_getgrnam()** routine looks up the next group entry in the registry that matches the input group name and returns the corresponding UNIX style group structure. The structure is in the following form:

```
typedef struct {
    sec_rgy_name_t name;
    signed32 gid;
    sec_rgy_member_buf_t members;
}               sec_rgy_unix_group_t;
```

The structure includes the following:

- The name of the group.
- The group's UNIX ID.
- A string containing the names of the group members. This string is limited in size by the size of the **sec_rgy_member_buf_t** type defined in **rgynbase.idl**.

The routine also returns an array of member names, limited in size by the *number_members* parameter. Note that the array contains only the names explicitly specified as members of the group. A principal that was made a member of the group because that group was assigned as the principal's primary group will not appear in the array.

This call is provided in source code form.

Files

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgybase.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

sec_rgy bad_data
The name supplied as input was too long.

error_status_ok
The call was successful.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_unix_getpwnam

Purpose

Returns a UNIX style passwd entry for account matching the specified name

Synopsis

```
#include <dce/rgynbase.h>

void sec_rgy_unix_getpwnam  (
     sec_rgy_handle_t context
     sec_rgy_name_t name
     unsigned32 name_len
     sec_rgy_cursor_t *item_cursor
     sec_rgy_unix_passwd_t *passwd_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open** to acquire a bound handle.

name A character string (of type **sec_rgy_name_t**) containing the name of the person, group, or organization whose name entry is desired.

name_len
A 32-bit integer representing the length of the *name* in characters.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_unix_getpwnam** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use **sec_rgy_cursor_reset** to refresh the cursor.

Output

passwd_entry
A UNIX style passwd structure returned with information about the account matching *name*.

status On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_unix_getpwnam** routine returns the next UNIX passwd structure that matches the input name. The structure is in the following form:

```
typedef struct {
       sec_rgy_unix_login_name_t name;
       sec_rgy_unix_passwd_buf_t passwd;
       signed32 uid;
```

```
        signed32 gid;
        signed32 oid;
        sec_rgy_unix_gecos_t gecos;
        sec_rgy_pname_t homedir;
        sec_rgy_pname_t shell;
    }               sec_rgy_unix_passwd_t;
```

The structure includes the following:

- The account's login name.
- The account's password.
- The account's UNIX ID.
- The UNIX ID of group and organization associated with the account.
- The account's GECOS information.
- The account's home directory.
- The account's login shell

This call is provided in source code form.

Files

/usr/include/dce/rgynbase.idl
The **idl** file from which **rgynbase.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy bad_data
The name supplied as input was too long.

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_unix_getpwuid

Purpose

Returns a UNIX style passwd entry for the account matching the specified UID

Synopsis

```
#include <dce/rgynbase.h>

void  sec_rgy_unix_getpwuid(
     sec_rgy_handle_t context
     signed32 uid
     sec_rgy_cursor_t *item_cursor
     sec_rgy_unix_passwd_t *passwd_entry
     error_status_t *status);
```

Parameters

Input

context
An opaque handle bound to a registry server. Use **sec_rgy_site_open()** to acquire a bound handle.

uid A 32-bit integer UNIX ID.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The **sec_rgy_unix_getpwuid()** routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries** . Use **sec_rgy_cursor_reset()** to refresh the cursor.

Output

passwd_entry
A UNIX style password structure returned with information about the account matching *uid*.

status On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_unix_getpwuid()** routine looks up the next **passwd** entry in the registry that matches the input UNIX ID and returns the corresponding **sec_rgy_passwd** structure. The structure is in the following form:

```
typedef struct {
      sec_rgy_unix_login_name_t name;
      sec_rgy_unix_passwd_buf_t passwd;
      signed32 Vuid;
      signed32 Vgid;
      signed32 oid;
```

```
        sec_rgy_unix_gecos_t gecos;
        sec_rgy_pname_t homedir;
        sec_rgy_pname_t shell;
}               sec_rgy_unix_passwd_t;
```

The structure includes the following:

- The account's login name.
- The account's password.
- The account's UNIX ID.
- The UNIX ID of group and organization associated with the account.
- The account's GECOS information.
- The account's home directory.
- The account's login shell

Files

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgynbase.h** was derived.

This call is provided in source code form.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

sec_rgy_server_unavailable
The DCE registry server is unavailable.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

sec_rgy_wait_until_consistent

Purpose

Blocks the caller while prior updates are propagated to the registry replicas

Synopsis

```
#include <dce/misc.h>

boolean32  sec_rgy_wait_until_consistent(
     sec_rgy_handle_t context
     error_status_t *status);
```

Parameters

Input

context
The registry server handle associated with the master registry.

Output

status A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

Description

The **sec_rgy_wait_until_consistent()** routine blocks callers until all prior updates to the master registry have been propagated to all active registry replicas.

Return Values

The routine returns TRUE when all active replicas have received the prior updates. It returns FALSE if at least one replica did not receive the updates.

Files

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

Errors

The following describes a partial list of errors that might be returned. Refer to the *OSF DCE Problem Determination Guide* for complete descriptions of all error messages.

sec_rgy_server_unavailable
The server for the master registry is not available.

sec_rgy_read_only
Either the master site is in maintenance mode or the site associated with the handle is a read-only (query) site.

error_status_ok
The call was successful.

Related Information

Functions: **sec_intro(3sec)**.

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y