First Edition (March 2001)

This edition, SC24-5908-00, applies to Version 1 Release 1 of z/OS DCE Base Services, z/OS DCE user Data Privacy (DES and CDMF), z/OS DCE User Data Privacy (CDMF) (program number 5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for reader's comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Information Development, Dept. G60
1701 North Street
Endicott, NY 13760-5553
United States of America

FAX (United States & Canada): 1+607+752-2327
FAX (Other Countries):
   Your International Access Code +1+607+752-2327

IBMLink™ (United States customers only): GDLVME(PUBRCF)
Internet e-mail: pubrcf@vnet.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994, 2001. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
The following statements are provided by the Open Software Foundation.

The information contained within this document is subject to change without notice.
OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.
OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing,
performance, or use of this material.
Copyright © 1993, 1994 Open Software Foundation, Inc.
This documentation and the software to which it relates are derived in part from materials supplied by the following:
• © Copyright 1990, 1991 Digital Equipment Corporation
• © Copyright 1990, 1991 Hewlett-Packard Company
• © Copyright 1989, 1990, 1991 Transarc Corporation
• © Copyright 1990, 1991 Siemens Nixdorf Informationssysteme AG
• © Copyright 1990, 1991 International Business Machines Corporation
• © Copyright 1988, 1989 Massachusetts Institute of Technology
• © Copyright 1988, 1989 The Regents of the University of California
All Rights Reserved.
Printed in the U.S.A.
 THIS DOCUMENT AND THE SOFTWARE DESCRIBED HEREIN ARE FURNISHED UNDER A LICENSE, AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE
TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. TITLE TO AND OWNERSHIP OF THE DOCUMENT AND SOFTWARE REMAIN
WITH OSF OR ITS LICENSORS.
Open Software Foundation, OSF, the OSF logo, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.
UNIX is a registered trademark of The Open Group in the United States and other countries.
DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation.
DECNet 3100 and DECnet are trademarks of Digital Equipment Corporation.
HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.
Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.
AFS and Transarc are registered trademarks of the Transarc Corporation.
Episode is a trademark of the Transarc Corporation.
Ethernet is a registered trademark of Xerox Corporation.
DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG.
MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.
NFS, Network File System, SunOS and Sun Microystems are trademarks of Sun Microsystems, Inc.
X/Open is a trademark of The Open Group in the U.K. and other countries.
PostScript is a trademark of Adobe Systems Incorporated.
FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE
These notices shall be marked on any reproduction of this data, in whole or in part.
NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government
regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.
RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the
Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.
RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights
in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication or
disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial Computer Software-Restricted Rights (April
1985)." If the contract contains the Clause at 18-52.227-74 "Rights in Data General" then the "Alternate III" clause applies.
US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract.
Unpublished—All rights reserved under the Copyright Laws of the United States.
This notice shall be marked on any reproduction of this data, in whole or in part.
## Contents

**About This Book** ................................................................. xxv
Who Should Use This Book ................................................... xxv
How This Book Is Organized .................................................. xxv
Terminology Used in This Book ........................................... xxvi
Conventions Used in This Book ........................................... xxvi
Where to Find More Information ........................................ xxvii
  - Softcopy Publications .................................................. xxvii
  - Internet Sources ........................................................ xxvii
  - Using LookAt to Look up Message Explanations ................ xxvii
  - Accessing Licensed Books on the Web ......................... xxvii

### Volume 1.

**Chapter 1. OSF Function Not Supported by z/OS DCE** ........................................ 1-1
  - Routines .............................................................................. 1-1
  - Macros ............................................................................... 1-1
  - Files .................................................................................. 1-1

**Chapter 2. Remote Procedure Call** ........................................ 2-1
  - IDL Base Data Types and IDL-to-C Type Mappings ................ 2-1
  - Remote Procedure Call Routines ......................................... 2-2
    - DCE RPC Runtime Services ............................................. 2-2
    - Environment Variables ............................................... 2-8
    - Data Types and Structures .......................................... 2-9
    - Permissions Required ................................................ 2-19
    - Frequently Used Routine Arguments ............................. 2-20
      - Related Information .................................................. 2-22
    - cs_byte_from_netcs ...................................................... 2-23
    - cs_byte_local_size ........................................................ 2-26
    - cs_byte_net_size ............................................................ 2-29
    - cs_byte_to_netcs ............................................................ 2-32
    - dce_cs_loc_to_rgy .......................................................... 2-35
    - dce_cs_rgy_to_loc .......................................................... 2-37
    - idl_es_decode_buffer ...................................................... 2-39
    - idl_es_decode_incremental ............................................. 2-41
    - idl_es_encode_dyn_buffer ............................................. 2-43
    - idl_es_encode_fixed_buffer ......................................... 2-45
    - idl_es_encode_incremental ............................................ 2-47
    - idl_es_handle_free ........................................................ 2-49
    - idl_es_inq_attrs ............................................................ 2-50
    - idl_es_inq_encoding_id ............................................... 2-52
    - idl_es_set_attrs ............................................................ 2-54
    - idl_es_set_transfer_syntax ........................................... 2-56
    - rpc_binding_copy .......................................................... 2-58
    - rpc_binding_free ........................................................... 2-60
    - rpc_binding_from_string_binding ................................... 2-62
    - rpc_binding_inq_auth_caller ......................................... 2-64
    - rpc_binding_inq_auth_client ......................................... 2-67
    - rpc_binding_inq_auth_info ........................................... 2-70
    - rpc_binding_inq_object ................................................ 2-73
    - rpc_binding_reset ........................................................ 2-74
    - rpc_binding_server_from_client ..................................... 2-76
    - rpc_binding_set_auth_info ........................................... 2-79
    - rpc_binding_set_object ............................................... 2-83
    - rpc_binding_to_string_binding ...................................... 2-84
    - rpc_binding_vector_free ............................................... 2-86

© Copyright IBM Corp. 1994, 2001
uuid_equal .............................................................. 2-319
uuid_from_string ................................................. 2-320
uuid_hash ............................................................ 2-321
uuid_is_nil ......................................................... 2-322
uuid_to_string .................................................... 2-323

Chapter 3. Threads .................................................. 3-1
  Threads Management ............................................. 3-1
  Attributes Object Routines .................................... 3-2
  Mutex Routines .................................................. 3-2
  Condition Variable Routines .................................. 3-3
  Thread-Specific Data Routines ................................. 3-3
  Cancelation Routines .......................................... 3-3
  Priority and Scheduling Routines ............................. 3-3
  Cleanup Routines ................................................ 3-3
  Exceptions in DCE Threads .................................... 3-4
  The sigwait Routine ............................................ 3-4
Exceptions in DCE Threads ........................................ 3-5
EXCEPTION_INIT .................................................. 3-6
RAISE ............................................................... 3-7
TRY-ENDTRY ........................................................ 3-8
CATCH .............................................................. 3-9
CATCH_ALL .......................................................... 3-10
RERAISE ............................................................ 3-11
FINALLY ............................................................. 3-12
exc_first_catch ................................................... 3-13
exc_get_reason_code ............................................. 3-14
exc_get_status .................................................... 3-15
exc_matches ........................................................ 3-16
exc_raise_status ................................................. 3-17
exc_set_status .................................................... 3-18
Threads Routines .................................................. 3-19
pthread_attr_create ............................................. 3-20
pthread_attr_delete ............................................. 3-21
pthread_attr_getstacksize ..................................... 3-22
pthread_attr_setstacksize ..................................... 3-23
pthread_cancel ................................................... 3-24
pthread_cleanup_pop ............................................. 3-25
pthread_cleanup_push .......................................... 3-26
pthread_cond_broadcast ........................................ 3-27
pthread_cond_destroy .......................................... 3-28
pthread_cond_init ............................................... 3-29
pthread_cond_signal ............................................. 3-31
pthread_cond_timedwait ........................................ 3-32
pthread_cond_wait ............................................... 3-34
pthread_condattr_create ....................................... 3-36
pthread_condattr_delete ....................................... 3-37
pthread_create .................................................. 3-38
pthread_delay_np ............................................... 3-40
pthread_detach .................................................. 3-41
pthread_equal .................................................... 3-42
pthread_exit ...................................................... 3-43
pthread_get_expiration_np ..................................... 3-44
pthread_getspecific ............................................. 3-45
pthread_join ...................................................... 3-46
pthread_keycreate .............................................. 3-47
pthread_lock_global_np ........................................ 3-49
pthread_mutex_destroy ......................................... 3-50
pthread_mutex_init .............................................. 3-51
pthread_mutex_lock .............................................. 3-52
Chapter 6. Security and Related Services ........................................................................  6-1
General ACL Manager Interface APIs ........................................................................  6-2
dce_acl_copy_acl ...........................................................................................................  6-3
dce_acl_inq_acl_from_header .......................................................................................  6-4
dce_acl_inq_client_creds ............................................................................................  6-5
dce_acl_inq_client_permset .........................................................................................  6-6
dce_acl_inq_permset_for_creds ...................................................................................  6-8
dce_acl_inq_prin_and_group .......................................................................................  6-10
dce_acl_is_client_authorized .....................................................................................  6-12
dce_acl_obj_add_any_other_entry ................................................................................  6-13
dce_acl_obj_add_foreign_entry ...................................................................................  6-14
dce_acl_obj_add_group_entry ......................................................................................  6-15
dce_acl_obj_add_id_entry ...........................................................................................  6-16
dce_acl_obj_add_obj_entry .........................................................................................  6-17
dce_acl_obj_add_unauth_entry ...................................................................................  6-18
dce_acl_obj_add_user_entry .......................................................................................  6-19
dce_acl_obj_free_entries ............................................................................................  6-20
dce_acl_obj_init ..........................................................................................................  6-21
dce_acl_register_object_type ......................................................................................  6-22
dce_acl_resolve_by_name ............................................................................................  6-25
dce_acl_resolve_by_uuid ............................................................................................  6-27
Introduction to the DCE Audit API Runtime ................................................................  6-29
Audit API Routines .......................... 6-29
Audit Data Types .................................. 6-30
Environment Variables .......................... 6-32
Permissions Required .......................... 6-33
dce_aud_close .................................. 6-34
dce_aud_commit .................................. 6-35
dce_aud_discard .................................. 6-38
dce_aud_free_ev_info .......................... 6-39
dce_aud_free_header ......................... 6-40
dce_aud_get_ev_info .......................... 6-41
dce_aud_get_header .......................... 6-42
dce_aud_length .................................. 6-43
dce_aud_next .................................. 6-45
dce_aud_open .................................. 6-48
dce_aud_print .................................. 6-51
dce_aud_put_ev_info .......................... 6-53
dce_aud_set_trail_size_limit .............. 6-54
dce_aud_start .................................. 6-56
dce_aud_start_with_name ...................... 6-59
dce_aud_start_with_pac ...................... 6-62
dce_aud_start_with_server_binding ........ 6-65
dce_aud_start_with_uuid ...................... 6-68
DCE Generic Security Service API ......... 6-71
  General Information .......................... 6-71
  GSSAPI Services .......................... 6-72
  Error Handling .......................... 6-73
  Data Types .................................. 6-75
  Using GSSAPI Services ................. 6-77
  gss_accept_sec_context ................. 6-80
  gss_acquire_cred .......................... 6-84
  gss_add_cred .................................. 6-86
  gss_add_oid_set_member ................. 6-88
  gss_canonicalize_name ................. 6-89
  gss_compare_name .......................... 6-90
  gss_context_time .......................... 6-91
  gss_create_empty_oid_set .............. 6-92
  gss_delete_sec_context ................. 6-93
  gss_display_name .......................... 6-94
  gss_display_status ...................... 6-95
  gss_duplicate_name ...................... 6-97
  gss_get_mic .................................. 6-98
  gss_import_name .......................... 6-100
  gss_indicate_mechs ...................... 6-102
  gss_init_sec_context ...................... 6-103
  gss_inquire_context ...................... 6-107
  gss_inquire_cred .......................... 6-109
  gss_inquire_cred_by_mech .............. 6-111
  gss_inquire_mechs_for_name ............. 6-113
  gss_inquire_names_for_mech ............. 6-114
  gss_oid_to_str .................................. 6-115
  gss_process_context_token .............. 6-116
  gss_release_buffer ...................... 6-117
  gss_release_cred .................................. 6-118
  gss_release_name .................................. 6-119
  gss_release_oid .................................. 6-120
  gss_release_oid_set ...................... 6-121
  gss_str_to_oid .................................. 6-122
  gss_test_oid_set_member .................. 6-123
  gss_unwrap .................................. 6-124
  gss_verify_mic .................................. 6-126

Contents xi
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gss_wrap</td>
<td>6-128</td>
</tr>
<tr>
<td>gss_wrap_size_limit</td>
<td>6-130</td>
</tr>
<tr>
<td>gssdce_cred_to_login_context</td>
<td>6-132</td>
</tr>
<tr>
<td>gssdce_export_cred</td>
<td>6-133</td>
</tr>
<tr>
<td>gssdce_extract_creds_from_sec_context</td>
<td>6-134</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_cred</td>
<td>6-135</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_sec_context</td>
<td>6-136</td>
</tr>
<tr>
<td>gssdce_import_cred</td>
<td>6-137</td>
</tr>
<tr>
<td>gssdce_login_context_to_cred</td>
<td>6-138</td>
</tr>
<tr>
<td>gssdce_register_acceptor_identity</td>
<td>6-139</td>
</tr>
<tr>
<td>gssdce_set_cred_context_ownership</td>
<td>6-140</td>
</tr>
<tr>
<td>gss_krb5_get_ccache</td>
<td>6-142</td>
</tr>
<tr>
<td>gss_krb5_get_tkt_flags</td>
<td>6-143</td>
</tr>
<tr>
<td>rdacl Routines</td>
<td>6-144</td>
</tr>
<tr>
<td>rdacl_get_access</td>
<td>6-145</td>
</tr>
<tr>
<td>rdacl_get_manager_types</td>
<td>6-147</td>
</tr>
<tr>
<td>rdacl_get_mngr_types_semantics</td>
<td>6-149</td>
</tr>
<tr>
<td>rdacl_get_printstring</td>
<td>6-151</td>
</tr>
<tr>
<td>rdacl_get_referral</td>
<td>6-153</td>
</tr>
<tr>
<td>rdacl_lookup</td>
<td>6-155</td>
</tr>
<tr>
<td>rdacl_replace</td>
<td>6-157</td>
</tr>
<tr>
<td>rdacl_release</td>
<td>6-159</td>
</tr>
<tr>
<td>rdacl_test_access</td>
<td>6-161</td>
</tr>
<tr>
<td>rdacl_test_access_on_behalf</td>
<td>6-161</td>
</tr>
<tr>
<td>SEC_Routines</td>
<td>6-163</td>
</tr>
<tr>
<td>Data Types</td>
<td>6-163</td>
</tr>
<tr>
<td>sec_acl</td>
<td>6-164</td>
</tr>
<tr>
<td>sec_attr</td>
<td>6-167</td>
</tr>
<tr>
<td>sec_id</td>
<td>6-170</td>
</tr>
<tr>
<td>sec_key</td>
<td>6-170</td>
</tr>
<tr>
<td>sec_login</td>
<td>6-171</td>
</tr>
<tr>
<td>sec_rgy</td>
<td>6-172</td>
</tr>
<tr>
<td>sec_rgy_attr</td>
<td>6-177</td>
</tr>
<tr>
<td>Constants</td>
<td>6-185</td>
</tr>
<tr>
<td>sec_acl</td>
<td>6-185</td>
</tr>
<tr>
<td>sec_attr</td>
<td>6-186</td>
</tr>
<tr>
<td>sec_id</td>
<td>6-187</td>
</tr>
<tr>
<td>sec_key</td>
<td>6-187</td>
</tr>
<tr>
<td>sec_login</td>
<td>6-187</td>
</tr>
<tr>
<td>sec_rgy</td>
<td>6-187</td>
</tr>
<tr>
<td>sec_acl_bind</td>
<td>6-189</td>
</tr>
<tr>
<td>sec_acl_bind_to_addr</td>
<td>6-190</td>
</tr>
<tr>
<td>sec_acl_calc_mask</td>
<td>6-191</td>
</tr>
<tr>
<td>sec_acl_get_access</td>
<td>6-192</td>
</tr>
<tr>
<td>sec_acl_get_error_info</td>
<td>6-194</td>
</tr>
<tr>
<td>sec_acl_get_manager_types</td>
<td>6-195</td>
</tr>
<tr>
<td>sec_acl_get_mngr_types_semantics</td>
<td>6-197</td>
</tr>
<tr>
<td>sec_acl_get_printstring</td>
<td>6-199</td>
</tr>
<tr>
<td>sec_acl_lookup</td>
<td>6-201</td>
</tr>
<tr>
<td>sec_acl_release</td>
<td>6-203</td>
</tr>
<tr>
<td>sec_acl_release_handle</td>
<td>6-204</td>
</tr>
<tr>
<td>sec_acl_replace</td>
<td>6-205</td>
</tr>
<tr>
<td>sec_acl_test_access</td>
<td>6-207</td>
</tr>
<tr>
<td>sec_acl_test_access_on_behalf</td>
<td>6-209</td>
</tr>
<tr>
<td>sec_attr_trig_query</td>
<td>6-211</td>
</tr>
<tr>
<td>sec_attr_trig_update</td>
<td>6-214</td>
</tr>
<tr>
<td>sec_cred_free_attr_cursor</td>
<td>6-216</td>
</tr>
<tr>
<td>sec_cred_free_cursor</td>
<td>6-217</td>
</tr>
<tr>
<td>sec_cred_free_pa_handle</td>
<td>6-218</td>
</tr>
<tr>
<td>sec_cred_get_authz_session_info</td>
<td>6-219</td>
</tr>
<tr>
<td>sec_cred_get_client_princ_name</td>
<td>6-220</td>
</tr>
</tbody>
</table>
sec_rgy_pgo_delete_member .................................................. 6-434
sec_rgy_pgo_get_by_eff_unix_num ........................................... 6-436
sec_rgy_pgo_get_by_id ......................................................... 6-439
sec_rgy_pgo_get_by_name ....................................................... 6-441
sec_rgy_pgo_get_by_unix_num .................................................. 6-443
sec_rgy_pgo_get_members ...................................................... 6-445
sec_rgy_pgo_get_next ............................................................ 6-448
sec_rgy_pgo_id_to_name ........................................................ 6-450
sec_rgy_pgo_id_to_unix_num .................................................... 6-452
sec_rgy_pgo_is_member .......................................................... 6-453
sec_rgy_pgo_name_to_id ........................................................ 6-455
sec_rgy_pgo_name_to_unix_num ............................................... 6-456
sec_rgy_pgo_rename .............................................................. 6-457
sec_rgy_pgo_replace ............................................................ 6-459
sec_rgy_pgo_unix_num_to_id ................................................... 6-461
sec_rgy_pgo_unix_num_to_name ............................................... 6-462
sec_rgy_plcy_get_effective .................................................... 6-464
sec_rgy_plcy_get_info .......................................................... 6-466
sec_rgy_plcy_set_info .......................................................... 6-468
sec_rgy_properties_get_info .................................................. 6-470
sec_rgy_properties_set_info .................................................. 6-472
sec_rgy_site_bind ............................................................... 6-474
sec_rgy_site_bind_query ........................................................ 6-476
sec_rgy_site_bind_update ...................................................... 6-478
sec_rgy_site_binding_get_info ............................................... 6-480
sec_rgy_site_close ............................................................... 6-482
sec_rgy_site_get ................................................................. 6-483
sec_rgy_site_is_readonly ...................................................... 6-485
sec_rgy_site_open ............................................................... 6-486
sec_rgy_site_open_query ....................................................... 6-487
sec_rgy_site_open_update ...................................................... 6-488
sec_rgy_unix_getgrgid .......................................................... 6-491
sec_rgy_unix_getgrnam .......................................................... 6-493
sec_rgy_unix_getpwnam .......................................................... 6-495
sec_rgy_unix_getpwuid .......................................................... 6-497
sec_rgy_wait_until_consistent ............................................... 6-499

Chapter 7. Kerberos .............................................................. 7-1
Kerberos Basics ...................................................................... 7-1
Purpose of Realms .................................................................. 7-1
Fundamental Assumptions About the Environment ......................... 7-2
Kerberos Terminology ........................................................... 7-2
Using Kerberos Files ............................................................. 7-3
Credentials Cache ................................................................... 7-3
Replay Cache ......................................................................... 7-3
Key Table .............................................................................. 7-4
Using Kerberos Services .......................................................... 7-4
Configuration Files ............................................................... 7-4
<krb5/krb.conf> ..................................................................... 7-4
<krb5/krb5.conf> ................................................................... 7-5
Sample <krb5/krb5.conf> Configuration File ................................. 7-7
Environment Variables ........................................................... 7-9
krb5_address_compare ............................................................. 7-10
krb5_address_search ............................................................... 7-11
krb5_auth_con_free ............................................................... 7-12
krb5_auth_con_genaddr ............................................................ 7-13
krb5_auth_con_getaddr ............................................................ 7-14
krb5_auth_con_getauthenticator ............................................... 7-15
krb5_auth_con_getflags ........................................................... 7-16
krb5_auth_con_getivector ......................................................... 7-17
krb5_free_creds ............................................................. 7-82
krb5_free_data ............................................................ 7-83
krb5_free_enc_tkt_part .................................................... 7-84
krb5_free_error ............................................................ 7-85
krb5_free_host_realm ...................................................... 7-86
krb5_free_kdc_rep .......................................................... 7-87
krb5_free_keyblock ......................................................... 7-88
krb5_free_keyblock_contents ............................................. 7-89
krb5_free_krbhst ............................................................ 7-90
krb5_free_principal ........................................................ 7-91
krb5_free_tgtcreds ........................................................ 7-92
krb5_free_ticket ............................................................ 7-93
krb5_free_tickets ............................................................ 7-94
krb5_gen_replay_name ...................................................... 7-95
krb5_generate_seq_number ................................................. 7-96
krb5_generate_subkey ....................................................... 7-97
krb5_get_cred_from_kdc ..................................................... 7-98
krb5_get_cred_from_kdc_renew ............................................. 7-99
krb5_get_cred_from_kdc_validate ........................................ 7-100
krb5_get_cred_via_tkt ....................................................... 7-101
krb5_get_credentials ......................................................... 7-102
krb5_get_credentials_renew ................................................. 7-103
krb5_get_credentials_validate ............................................ 7-104
krb5_get_default_in_tkt_ktypes .......................................... 7-105
krb5_get_default Realm ...................................................... 7-106
krb5_get_default_tgs_ktypes ................................................. 7-107
krb5_get_host_realm ........................................................ 7-108
krb5_get_in_tkt_with_keytab ............................................. 7-109
krb5_get_in_tkt_with_password .......................................... 7-111
krb5_get_in_tkt_with_skey ............................................... 7-113
krb5_get krbhst .............................................................. 7-115
krb5_get_server_rcache ..................................................... 7-116
krb5_init_context ............................................................ 7-117
krb5_kt_add_entry .......................................................... 7-118
krb5_kt_close ................................................................. 7-119
krb5_kt_default .............................................................. 7-120
krb5_kt_default_name ......................................................... 7-121
krb5_kt_end_seq_get ........................................................ 7-122
krb5_kt_free_entry .......................................................... 7-123
krb5_kt_get_entry ............................................................. 7-124
krb5_kt_get_name ............................................................. 7-125
krb5_kt_get_type .............................................................. 7-126
krb5_kt_next_entry ............................................................ 7-127
krb5_kt_read_service_key ................................................. 7-128
krb5_kt_register ............................................................. 7-129
krb5_kt_remove_entry ......................................................... 7-130
krb5_kt_resolve ............................................................... 7-131
krb5_kt_start_seq_get ....................................................... 7-132
krb5_md4_crypto_compat_ctl ............................................... 7-133
krb5_md5_crypto_compat_ctl ............................................... 7-134
krb5_mk_error ................................................................. 7-135
krb5_mk_priv ................................................................. 7-136
krb5_mk_rep ................................................................. 7-137
krb5_mk_req ................................................................. 7-138
krb5_mk_req_extended ....................................................... 7-140
krb5_mk_safe ................................................................. 7-141
krb5_os_hostaddr ............................................................ 7-142
krb5_os_localaddr ........................................................... 7-143
krb5_parse_name ............................................................. 7-144
krb5_principal_compare ..................................................... 7-145

Contents xvii
The DCE Configuration Routines ...............................................  8-21
 DCE Configuration File ...................................................  8-21
 DCE Configuration API Output ...............................................  8-22

dce_cf_binding_entry_from_host ........................................... 8-23

dce_cf_dced_entry_from_host .............................................. 8-25

dce_cf_find_name_by_key .................................................. 8-27

dce_cf_free_cell_aliases .................................................. 8-29

dce_cf_get_cell_aliases .................................................. 8-30

dce_cf_get_cell_name .................................................... 8-31

dce_cf_get_csgy_filename ................................................ 8-33

dce_cf_get_host_name .................................................... 8-35

dce_cf_prin_name_from_host ............................................. 8-37

dce_cf_get_host_name .................................................... 8-35

dce_cf_get_cell_aliases .................................................. 8-30

dce_cf_get_cell_name .................................................... 8-31

dce_cf_get_csgy_filename ................................................ 8-33

dce_cf_get_host_name .................................................... 8-35

dce_cf_prin_name_from_host ............................................. 8-37

Chapter 8. DCE Utility Routines ........................................ 8-1
The DCE Attribute Interface Routines .................................... 8-2
dce_attr_sch_bind ......................................................... 8-3

dce_attr_sch_bind_free .................................................. 8-5

dce_attr_sch_create_entry ............................................... 8-6

dce_attr_sch_cursor_alloc .............................................. 8-7

dce_attr_sch_cursor_init ................................................ 8-8

dce_attr_sch_cursor_release ............................................ 8-10

dce_attr_sch_cursor_reset .............................................. 8-11

dce_attr_sch_delete_entry .............................................. 8-12

dce_attr_sch_get_acl_mgrs ............................................. 8-13

dce_attr_sch_lookup_by_id .............................................. 8-15

dce_attr_sch_lookup_by_name .......................................... 8-16

dce_attr_sch_scan ........................................................ 8-17

dce_attr_sch_update_entry .............................................. 8-19

The DCE Configuration Routines ........................................ 8-21
DCE Configuration File ................................................... 8-21
DCE Configuration API Output ............................................... 8-22

dce_cf_binding_entry_from_host ........................................... 8-23

dce_cf_dced_entry_from_host .............................................. 8-25

dce_cf_find_name_by_key .................................................. 8-27

dce_cf_free_cell_aliases .................................................. 8-29

dce_cf_get_cell_aliases .................................................. 8-30

dce_cf_get_cell_name .................................................... 8-31

dce_cf_get_csgy_filename ................................................ 8-33

dce_cf_get_host_name .................................................... 8-35

dce_cf_prin_name_from_host ............................................. 8-37
Appendix A. z/OS DCE Supplied Header, IDL and ACF Files
Header Files in /usr/lpp/dce/share/include ...........................................  A-1
Header Files in /usr/lpp/dce/share/include/dce .........................................  A-3
z/OS DCE IDL and ACF Files ...........................................................................  A-8

Appendix B. Notices
Trademarks .................................................................................................  B-2
Programming Interface Information .........................................................  B-2

Glossary ........................................................................................................  X-1

Bibliography .................................................................................................  X-19
z/OS DCE Publications ................................................................................  X-19
z/OS SecureWay® Security Server Publications ........................................  X-19
Tool Control Language Publication .............................................................  X-20
IBM C/C++ Language Publication ...............................................................  X-20
z/OS DCE Application Support Publications ..............................................  X-20
Encina Publications ...................................................................................  X-21

Index ............................................................................................................  X-23
Figures

5-1. DTS Application Program Interface Routines by Function ........................................ 5-2
### Tables

2-1. Base Data Type Specifiers ............................................. 2-1
2-2. RPC Stub Support Routines ........................................... 2-2
2-3. RPC Nonstub Support Routines ........................................ 2-3
2-4. Client and Server Binding Handles .................................... 2-9
2-5. DCE Valid Protocol Sequences ....................................... 2-16
2-6. Valid Name Syntaxes ............................................. 2-21
2-7. Casts for Authorization Information .................................. 2-67
2-8. How rpc_ep_register affects the EPM ................................ 2-99
2-9. How rpc_ep_register_wlb affects the EPM .......................... 2-103
2-10. Values Returned by Routine rpc_mgmt_is_server_listening ........... 2-141
2-11. Operation Values Passed to Authorization Function .................. 2-142
2-12. Default Controls for Remote Management Routines .................. 2-143
2-13. Predefined Timeout Values ........................................ 2-147
2-14. Rules for Returning an Object’s Type ................................ 2-240
2-15. RPC Key Acquisition for Authentication Services ................. 2-259
2-16. Rules for Calling Manager Routines ................................... 2-263
2-17. Rules for Removing an Interface ...................................... 2-264
3-1. Threads Management ..................................................... 3-1
3-2. Attributes Object Routines ........................................... 3-2
3-3. Mutex Routines .................................................... 3-2
3-4. Condition Variable Routines .......................................... 3-3
3-5. Thread-Specific Data Routines ....................................... 3-3
3-6. Cancelation Routines .............................................. 3-3
3-7. Priority and Scheduling Routines ..................................... 3-3
3-8. Cleanup Routines .................................................. 3-3
3-9. Exceptions in DCE Threads ........................................... 3-4
3-10. The sigwait Routine ............................................... 3-4
4-1. XDS Routines ..................................................... 4-2
4-2. DS_C_ERROR Object Error Messages .................................. 4-91
5-1. DTS Application Program Interface Routines ........................ 5-2
6-1. Supported ACL Entry Types ......................................... 6-9
6-2. Event Data Format Specifiers ....................................... 6-31
About This Book

This book explains in detail the Distributed Computing Environment (DCE) Application Program Interfaces (APIs) that you will need to write distributed applications on the IBM z/OS operating system using z/OS DCE.

Who Should Use This Book

As an application developer or programmer creating distributed applications, you will use the z/OS DCE APIs listed in this book. You should already be familiar with C programming concepts, and be able to:

- Allocate z/OS data sets
- Edit, browse, and copy z/OS data sets and associated members
- Print data sets
- Write and submit batch jobs on z/OS
- Write, compile, link, and run C/C++ programs on z/OS
- Write and understand JCL to run on z/OS
- Understand Shell and TSO/E commands.

You should have a good knowledge and understanding of the following:

- Interactive System Productivity Facility/Program Development Facility (ISPF/PDF)
- Structured programming techniques
- Concepts behind a distributed application
- Using the Spool Display and Search Facility (SDSF) to check on the status of your application

Some exposure to the UNIX or AIX® operating systems is helpful.

How This Book Is Organized

This book is divided into two volumes: Volume 1 and Volume 2. Volume 1 contains the following information:

- Chapter 1, “OSF Function Not Supported by z/OS DCE” on page 1-1 lists the unsupported OSF function.
- Chapter 2, “Remote Procedure Call” on page 2-1 describes the DCE Remote Procedure Call (RPC) Application Programming Interfaces (APIs).
- Chapter 3, “Threads” on page 3-1 describes the DCE Threads, threads exception handling macros, and APIs.
- Chapter 4, “Directory Services” on page 4-1 describes the DCE Directory Services and the APIs and header files for X/Open Data Services (XDS) and X/Open OSI-Abstract-Data Manipulation (XOM).
- Chapter 5, “Distributed Time Services” on page 5-1 describes the RPC Distributed Time Services (DTS), and APIs.

Volume 2 contains the following information:

- Chapter 6, “Security and Related Services” on page 6-1 describes the RPC Security Services, and the rdacl, acl, ID Mapping, Key Management, Login, Audit, Delegation, Password Management and Registry APIs. It also describes the Generic Security Service APIs and the General ACL APIs.
- Chapter 7, “Kerberos” on page 7-1 describes the routines that make up the Kerberos Version 5 application programming interface.
- Chapter 8, “DCE Utility Routines” on page 8-1 describes the DCE utility APIs.

Each chapter is organized alphabetically by API and shows the format, parameters, and usage for each API.
Terminology Used in This Book

Although every attempt has been made to conform to z/OS terminology, the DCE technology has been developed from the UNIX environment, and many DCE concepts and terms in this book relate to that environment. z/OS terms and concepts are used throughout this book wherever possible.

The term file can refer to a sequential data set, a member of a partitioned data set, or a hierarchical file system (HFS) data set. For more information on hierarchical file systems in z/OS, see z/OS UNIX System Services User's Guide, SA22-7801.

The term user prefix refers to the names of sequential or partitioned data sets. In a z/OS environment, the user prefix is usually a user's login identification. If desired, you can set the user prefix to a value other than the your login identification by using the TSO/E profile command. In z/OS batch mode, your user prefix depends on whether Resource Access Control Facility (RACF®), a component of the SecureWay® Security Server for z/OS, is installed on your system. If RACF is installed, and you are processing in batch mode, your user prefix can be the same as your login user identification. If RACF is not installed and you are processing in batch mode under z/OS, you may not have to use a prefix. See your systems programmer to determine the RACF settings for your site.

Unless otherwise specified, when the full name of a sequential or partitioned data set is referred to, the high-level qualifier for that data set will be represented by USERPRFX. The USERPRFX is determined by the application developer, and depends on the library where the application is installed. For example, USERPRFX.EXAMPLE.C(MEMBER) represents a partitioned data set whose first level qualifier is represented by USERPRFX, whose second level qualifier is EXAMPLE, and whose third level qualifier is C. Its member is MEMBER.

The terms API, call, and routine all refer to the z/OS DCE application program interface. For example, rpc_binding_free API, rpc_binding_free call, and rpc_binding_free routine all refer to the same rpc_binding_free API.

All references to individual DCE components (such as RPC) refer to that component within z/OS DCE. For example, RPC, DCE RPC, and z/OS DCE RPC all refer to the same z/OS DCE component.

In this book the term “DCE Security Server” (or simply “Security Server”) refers to the z/OS SecureWay Security Server DCE or to a DCE Security Server provided on another host in the DCE cell. The z/OS SecureWay Security Server DCE is a component of the SecureWay Security Server for z/OS.

Conventions Used in This Book

This book uses the following typographic conventions:

**Bold**  
**Bold** words or characters represent API names, attributes, status codes, environment variables, parameter values, and system elements that you must enter into the system literally, such as commands, options, or path names.

*Italic*  
*Italic* words or characters represent values for variables that you must supply.

Example font  
Examples and information displayed by the system appear in constant width type style.

[]  
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.

\  
A backslash is used as a continuation character when entering commands from the shell that exceed one line (255 characters). If the command exceeds one line, use the backslash character \ as the last nonblank character on the line to be continued, and continue the command on the next line.

This book uses the following keying conventions:

<Alt-c>  
The notation <Alt-c> followed by the name of a key indicates a control character sequence.

<Return>  
The notation <Return> refers to the key on your keyboard that is labeled with the word Return or Enter, or with a left arrow.
Entering commands  When instructed to enter a command, type the command name and then press <Return>.

Where to Find More Information

Where necessary, this book references information in other books using shortened versions of the book title. For complete titles and order numbers of the books for all products that are part of z/OS, see the z/OS Information Roadmap, SA22-7500. For complete titles and order numbers of the books for z/OS DCE, refer to the publications listed in the “Bibliography” on page X-19.

For information about installing z/OS DCE components, see the z/OS Program Directory.

Softcopy Publications

The z/OS DCE library is available on a CD-ROM, z/OS Collection, SK3T-4269. The CD-ROM online library collection is a set of unlicensed books for z/OS and related products that includes the IBM Library Reader. This is a program that enables you to view the BookManager files. This CD-ROM also contains the Portable Document Format (PDF) files. You can view or print these files with the Adobe Acrobat reader.

Internet Sources

The Softcopy z/OS publications are also available for web-browsing and for viewing or printing PDFs using the following URL:

You can also provide comments about this book and any other z/OS documentation by visiting that URL. Your feedback is important in helping to provide the most accurate and high-quality information.

Using LookAt to Look up Message Explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages. You can also use LookAt to look up explanations of system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on the News and Help link or from the z/OS Collection, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: lookat message-id as in the following:
lookat iec192i

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID and select the release with which you are working.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in z/OS MVS Routing and Descriptor Codes, SA22-7624. For such messages, LookAt prompts you to choose which book to open.

Accessing Licensed Books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link site:
http://www.ibm.com/servers/resourcelink
Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link user ID, password, and z/OS licensed book key code. The z/OS order that you received provides a memo that includes your key code.

To obtain your IBM Resource Link user ID and password, logon to:
http://www.ibm.com/servers/resourcelink

To register for access to the z/OS licensed books:
1. Logon to Resource Link using your Resource Link user ID and password.
2. Select User Profiles located on the left-hand navigation bar.
5. Supply your key code where requested and select the Submit button.

If you supplied the correct key code you will receive confirmation that your request is being processed.

After your request is processed you will receive an e-mail confirmation.

Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:
1. Logon to Resource Link using your Resource Link user ID and password.
2. Select Library.
3. Select zSeries.
4. Select Software.
5. Select z/OS.
6. Access the licensed book by selecting the appropriate element.
Chapter 1. OSF Function Not Supported by z/OS DCE

The following OSF routines and macros are not supported in z/OS DCE and are not discussed in this book:

### Routines

- `dce_assert`
- `dce_fprintf`
- `dce_msg_cat_close`
- `dce_msg_cat_open`
- `dce_msg_define_msg_table`
- `dce_err_get`
- `dce_msg_get_cat_msg`
- `dce_msg_get_msg`
- `dce_msg_get_default_msg`
- `dce_msg_translate_table`
- `dce_pgm_fprintf`
- `dce_pgm_printf`
- `dce_pgm_sprintf`
- `dce_printf`
- `dce_sprintf`
- `dce_svc_components`
- `dce_svc_debug_routing`
- `dce_svc_debug_set_levels`
- `dce_svc_define_filter`
- `dce_svc_filter`
- `dce_svc_log_close`
- `dce_svc_log_get`
- `dce_svc_log_rewind`
- `dce_svc_log_set`
- `dce_svc_register`
- `dce_svc_routing`
- `dce_svc_set_prograde`
- `dce_svc_set progname`
- `dce_svc_table`
- `dce_svc_unregister`
- `dced_server_disable_if`
- `dced_server_enable_if`
- `ds_modify_rdn`
- `ds_search`
- `ds_receive_result`
- `ds_modify_rdn`
- `ds_search`
- `ds_receive_result`

### Macros

- `DCE_SVC_DEBUG`
- `DCE_SVC_DEBUG_IS`
- `DCE_SVC_LOG`
- `DCE_SVC_DEBUG_ATLEAST`
- `DCE_SVC_DEFINE_HANDLE`

### Files

- `svcroute`
Chapter 2. Remote Procedure Call

This chapter describes Distributed Computing Environment (DCE) application program interfaces (APIs) in detail.

IDL Base Data Types and IDL-to-C Type Mappings

The **IDL compiler** converts an interface definition into output files. A complete summary of the **idl** command, which calls the IDL compiler, can be found in the [z/OS DCE Command Reference](http://www.ibm.com/support/docview.wss?rs=160&uid=swg27009408).

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>
<thead>
<tr>
<th>Specifier</th>
<th>Size</th>
<th>Type Macro Emitted by idl Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>small int</td>
<td>8 bits</td>
<td>idl_small_int</td>
</tr>
<tr>
<td>short int</td>
<td>16 bits</td>
<td>idl_short_int</td>
</tr>
<tr>
<td>long int</td>
<td>32 bits</td>
<td>idl_long_int</td>
</tr>
<tr>
<td>hyper int</td>
<td>64 bits</td>
<td>idl_hyper_int</td>
</tr>
<tr>
<td>unsigned small int</td>
<td>8 bits</td>
<td>idl_usmall_int</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>16 bits</td>
<td>idl_ushort_int</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>32 bits</td>
<td>idl_ulong_int</td>
</tr>
<tr>
<td>unsigned hyper int</td>
<td>64 bits</td>
<td>idl_uhyper_int</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>idl_short_float</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>idl_long_float</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>idl_char</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>idl_norm_char</td>
</tr>
<tr>
<td>boolean</td>
<td>8 bits</td>
<td>idl_boolean</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>idl_byte</td>
</tr>
<tr>
<td>void</td>
<td>--</td>
<td>idl_void_p_t</td>
</tr>
</tbody>
</table>

To ensure that your type declarations are consistent with those in the **stubs**, you can write the **idl** macros in the code you create for your application, even when your application will be run on another **platform**. The **idl** macros are especially useful when you want to pass constant values to RPC calls that are declared in your **network** interfaces. For maximum portability, all constants passed to RPC calls should be cast to the appropriate type, because the size of an integer constant (like the size of the **int** data type) is unspecified in the C language.

The **idl** macros are defined in `dce/idlbase.h`. This file is included by header files that the IDL compiler generates.

For related information, see [z/OS DCE Messages and Codes](http://www.ibm.com/support/docview.wss?rs=160&uid=swg27009408) and [z/OS DCE Application Development Guide: Core Components](http://www.ibm.com/support/docview.wss?rs=160&uid=swg27009408).
Remote Procedure Call Routines

This section provides an overview of DCE RPC API runtime and then discusses each API in detail.

This introduction gives general information about the DCE RPC Application Program Interface (API) and an overview of the different parts of the DCE RPC API runtime. The following topics are discussed in this introduction:

- Runtime services
- Environment variables
- Data types and structures
- Permissions required
- Frequently used routine arguments

DCE RPC Runtime Services

The RPC runtime services consist of RPC routines that perform a variety of operations. They can be divided up into either stub support routines or nonstub support routines.

Note: 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 section about the internals of remote procedure calls in the RPC module of z/OS DCE Application Development Guide: Core Components.

RPC Stub Support Routines: An alphabetical list of the RPC stub memory management and support APIs follows. With each API name is its description and the type of application program that most likely calls the routine. Detailed descriptions of each RPC API start on page 2-58.

<table>
<thead>
<tr>
<th>Routines</th>
<th>Description</th>
<th>Application Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs_byte_from_netcset</td>
<td>Converts international character data from a network code set to a local code set.</td>
<td>Client or server</td>
</tr>
<tr>
<td>cs_byte_local_size</td>
<td>Calculates the necessary buffer size for a code set conversion from a network code set to a local code set.</td>
<td>Client or server</td>
</tr>
<tr>
<td>cs_byte_net_size</td>
<td>Calculates the necessary buffer size for a code set conversion from a local code set to a network code set.</td>
<td>Client or server</td>
</tr>
<tr>
<td>cs_byte_to_netcset</td>
<td>Converts international character data from a local code set to a network code set.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_sm_allocate</td>
<td>Allocates storage within the RPC stub memory management scheme.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>rpc_sm_client_free</td>
<td>Frees storage allocated by the current storage allocation and freeing mechanism used by the client stubs.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_sm_destroy_client_context</td>
<td>Reclaims the client storage resources for a context handle, and sets the context handle to NULL</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_sm_disable_allocate</td>
<td>Releases resources and allocated storage within the RPC stub memory management.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_sm_enable_allocate</td>
<td>Enables the stub memory management environment.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_sm_free</td>
<td>Frees storage allocated by the rpc_sm_allocate routine.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>rpc_sm_get_thread_handle</td>
<td>Gets a thread handle for the stub memory management environment.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>rpc_sm_set_client_alloc_free</td>
<td>Sets the storage allocation and freeing mechanism used by the client stubs.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_sm_set_thread_handle</td>
<td>Sets a thread handle for the stub memory management environment.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>rpc_sm_swap_client_alloc_free</td>
<td>Exchanges the current storage allocation and freeing mechanism used by the client stubs with one specified by the client.</td>
<td>Client</td>
</tr>
</tbody>
</table>
### Table 2-2 (Page 2 of 2). RPC Stub Support Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Application Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>.rpc_ss_allocate</td>
<td>Allocates storage within the RPC stub memory management scheme.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>.rpc_ss_bind_authn_client</td>
<td>Authenticates a client's identity to a server from a client stub.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_client_free</td>
<td>Frees storage returned from a client stub.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_destroy_client_context</td>
<td>Reclaims the client storage resources for the context handle, and sets the context handle to null.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_disable_allocate</td>
<td>Releases resources and allocated storage.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_enable_allocate</td>
<td>Enables the allocation of storage by the rpc_ss_allocate API when not in manager code.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_free</td>
<td>Frees storage allocated by the rpc_ss_allocate routine.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>.rpc_ss_get_thread_handle</td>
<td>Gets a thread handle for the manager code before it spawns additional threads, or for the client code when it becomes a server.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>.rpc_ss_set_client_alloc_free</td>
<td>Sets the storage allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage storage for pointed-to nodes.</td>
<td>Client.</td>
</tr>
<tr>
<td>.rpc_ss_set_thread_handle</td>
<td>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.</td>
<td>Usually server, possibly client</td>
</tr>
<tr>
<td>.rpc_ss_swap_client_alloc_free</td>
<td>Exchanges the current storage allocation and freeing mechanism used by the client stubs with one specified by the client.</td>
<td>Client.</td>
</tr>
</tbody>
</table>

### RPC Nonstub Support Routines:
The following is a list of the abbreviations used in naming the nonstub support APIs:

- auth: authentication, authorization
- com: communication
- 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

The following table lists the RPC nonstub support routines. Along with the routines is a description and the type of application program that most likely calls the routines.

### Table 2-3 (Page 1 of 6). RPC Nonstub Support Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Application Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_cs_loc_to_rgy</td>
<td>Maps a local name for a code set to a code set value in the code set registry.</td>
<td>Client or server</td>
</tr>
<tr>
<td>Routines</td>
<td>Description</td>
<td>Application Program</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>dce_cs_rgy_to_loc</td>
<td>Maps a code set value in the code set registry to the local name for a code set.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_decode_buffer</td>
<td>Returns a buffer decoding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_decode_incremental</td>
<td>Returns an incremental decoding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_encode_dyn_buffer</td>
<td>Returns a dynamic buffer encoding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_encode_fixed_buffer</td>
<td>Returns a fixed buffer encoding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_encode_incremental</td>
<td>Returns an incremental encoding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_handle_free</td>
<td>Frees an IDL encoding services handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_inq_attrs</td>
<td>Returns the attribute flags from an IDL encoding services handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_inq_encoding_id</td>
<td>Identifies an application encoding operation.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_set_attribs</td>
<td>Sets attribute flags from an IDL encoding services handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>idl_es_set_transfer_syntax</td>
<td>Sets a transfer syntax.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_binding_copy</td>
<td>Returns a copy of a binding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_binding_free</td>
<td>Releases server binding handle resources.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_binding_from_string_binding</td>
<td>Returns a server binding handle from a string representation of a binding handle.</td>
<td>Client or server, management</td>
</tr>
<tr>
<td>rpc_binding_inq_auth_caller</td>
<td>Returns authentication and authorization information from the binding handle for an authenticated client.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_binding_inq_auth_client</td>
<td>Returns authentication and authorization information from the client binding handle for an authenticated client.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_binding_inq_auth_info</td>
<td>Returns authentication and authorization information from a server binding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_binding_inq_object</td>
<td>Returns the object UUID from a binding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_binding_reset</td>
<td>Resets a server binding handle so the host remains specified, but the server instance on that host is unspecified.</td>
<td>Client or server, management</td>
</tr>
<tr>
<td>rpc_binding_server_from_client</td>
<td>Converts a client binding handle to a server binding handle.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_binding_set_auth_info</td>
<td>Sets authentication and authorization information into a server binding handle.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_binding_set_object</td>
<td>Sets the object UUID value into a server binding handle.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_binding_to_string_binding</td>
<td>Returns a string representation of a binding handle.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_binding_vector_free</td>
<td>Frees the storage used to store a vector of binding handles.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_cs_binding_set_tags</td>
<td>Places code set tags into a server binding handle.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_cs_char_set_compat_check</td>
<td>Evaluates character set compatibility between a client and a server.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_cs_eval_with_universal</td>
<td>Evaluates a server's supported character sets and code sets during the server binding selection process.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_cs_eval_without_universal</td>
<td>Evaluates a server's supported character sets and code sets during the server binding selection process.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_cs_get_tags</td>
<td>Retrieves code set tags from a binding handle.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_ep_register</td>
<td>Adds to, or replaces, server address information in the local endpoint map.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ep_register_wlb</td>
<td>Adds to, or replaces, server address information in the local endpoint map.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace</td>
<td>Adds to server address information in the local endpoint map.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace_wlb</td>
<td>Adds to server address information in the local endpoint map.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ep_resolve_binding</td>
<td>Resolves a partially bound server binding handle into a fully bound server binding handle.</td>
<td>Client or management</td>
</tr>
<tr>
<td>Routines</td>
<td>Description</td>
<td>Application Program</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>rpc_ep_unregister</td>
<td>Removes server address information from the local endpoint map.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_get_ae_name</td>
<td>Gets the AENAME the server will use to identify equivalent instances to workload balancing.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_if_id_vector_free</td>
<td>Frees a vector and the interface identifier structures it contains.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_if_inq_id</td>
<td>Returns the interface identifier for an interface specification.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_mgmt_ep_elt_inq_begin</td>
<td>Creates an inquiry context for viewing the elements in a local or remote endpoint map.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_mgmt_ep_elt_inq_done</td>
<td>Deletes the inquiry context for viewing the elements in a local or remote endpoint map.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_mgmt_ep_elt_inq_next</td>
<td>Returns one element at a time from a local or remote endpoint map.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_mgmt_ep_setActivated_wlb</td>
<td>Sets or resets the value of the activated bit for the designated endpoint.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_mgmt_ep_unregister</td>
<td>Removes server address information from a local or remote endpoint map.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_mgmt_inq_com_timeout</td>
<td>Returns the communication timeout value in a binding handle.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_mgmt_inq_dfllt_protect_level</td>
<td>Returns the default protection level for an authentication service.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_mgmt_inq_if_ids</td>
<td>Returns a vector of interface identifiers of interfaces a server offers.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_mgmt_inq_server_princ_name</td>
<td>Returns a server's principal name.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_mgmt_inq_stats</td>
<td>Returns RPC runtime statistics.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_mgmt_is_server_listening</td>
<td>Tells whether a server is listening for remote procedure calls.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_mgmt_set_authorization_fn</td>
<td>Establishes an authorization function for processing remote calls to a server's management routines.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_mgmt_set_cancel_timeout</td>
<td>Sets the lower bound on the time to wait before timing out after forwarding a cancel.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_mgmt_set_com_timeout</td>
<td>Sets the communication timeout value in a binding handle.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_mgmt_stats_vector_free</td>
<td>Frees a statistics vector.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_mgmt_stop_server_listening</td>
<td>Tells a server to stop listening for remote procedure calls.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_network_inq_protseqs</td>
<td>Returns all protocol sequences supported by both the RPC runtime and the operating system.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_network_is_protseq_valid</td>
<td>Tells whether the specified protocol sequence is supported by both the RPC runtime and the operating system.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_ns_binding_export</td>
<td>Establishes a name service database entry with binding handles or object UUIDs for a server.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ns_binding_import_begin</td>
<td>Creates an import context for an interface and an object in the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_import_done</td>
<td>Deletes the import context for searching the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_import_next</td>
<td>Returns a binding handle of a compatible server (if found) from the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_inq_entry_name</td>
<td>Returns the name of an entry in the name service database from which the server binding handle came.</td>
<td>Client</td>
</tr>
<tr>
<td>Routines</td>
<td>Description</td>
<td>Application Program</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>rpc_ns_binding_lookup_begin</td>
<td>Creates a lookup context for an interface and an object in the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_lookup_done</td>
<td>Deletes the lookup context for searching the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_lookup_next</td>
<td>Returns a list of binding handles of one or more compatible servers (if found) from the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_select</td>
<td>Returns a binding handle from a list of compatible server binding handles.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_binding_unexport</td>
<td>Removes the binding handles for an interface, or object UUIDs, from an entry in the name service database.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_ns_entry_expand_name</td>
<td>Expands the name of a name service entry.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_entry_inq_resolution</td>
<td>Resolves the cell namespace components of a name and returns partial results.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_entry_object_inq_begin</td>
<td>Creates an inquiry context for viewing the objects of an entry in the name service database.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_entry_object_inq_done</td>
<td>Deletes the inquiry context for viewing the objects of an entry in the name service database.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_entry_object_inq_next</td>
<td>Returns one object at a time from an entry in the name service database.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_delete</td>
<td>Deletes a group attribute.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_mbr_add</td>
<td>Adds an entry name to a group; if necessary, creates the entry.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_mbr_inq_begin</td>
<td>Creates an inquiry context for viewing group members.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_mbr_inq_done</td>
<td>Deletes the inquiry context for a group.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_mbr_inq_next</td>
<td>Returns one member name at a time from a group.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_group_mbr_remove</td>
<td>Removes an entry name from a group.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_import_ctx_add_eval</td>
<td>Adds an evaluation routine to an import context</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_mgmt_binding_unexport</td>
<td>Removes multiple binding handles or object UUIDs from an entry in the name service database.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_entry_create</td>
<td>Creates an entry in the name service database.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_entry_delete</td>
<td>Deletes an entry from the name service database.</td>
<td>Management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_entry_inq_if_ids</td>
<td>Returns the list of interfaces exported to an entry in the name service database.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_free_codesets</td>
<td>Frees a code sets array that has been allocated in memory.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_ns_mgmt_handle_set_exp_age</td>
<td>Sets a handle’s expiration age for local copies of name service data.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_inq_exp_age</td>
<td>Returns the application’s global expiration age for local copies of name service data.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_read_codesets</td>
<td>Reads the code sets attribute associated with an RPC server entry in the name service database.</td>
<td>Client</td>
</tr>
<tr>
<td>rpc_ns_mgmt_remove_attribute</td>
<td>Removes an attribute from an RPC server entry in the name service database.</td>
<td>Server or management</td>
</tr>
<tr>
<td>rpc_ns_mgmt_set_attribute</td>
<td>Adds an attribute to an RPC server entry in the name service database.</td>
<td>Server or management</td>
</tr>
<tr>
<td>Routines</td>
<td>Description</td>
<td>Application Program</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>rpc_ns_mgmt_set_exp_age</td>
<td>Changes the application’s global expiration age for local copies of name service data.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_delete</td>
<td>Deletes a profile attribute.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_elt_add</td>
<td>Adds an element to a profile. If necessary, creates the entry.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_elt_inq_begin</td>
<td>Creates an inquiry context for viewing the elements in a profile.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_elt_inq_done</td>
<td>Deletes the inquiry context for a profile.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_elt_next</td>
<td>Returns one element at a time from a profile.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_ns_profile_elt_remove</td>
<td>Removes an element from a profile.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_object_inq_type</td>
<td>Returns the type of an object.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_object_set_inq_fn</td>
<td>Registers an object inquiry function.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_object_set_type</td>
<td>Assigns the type of an object.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_protseq_vector_free</td>
<td>Frees the storage used by a vector and its protocol sequences.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_rgy_get_codesets</td>
<td>Gets supported code sets information from the local host.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_rgy_get_max_bytes</td>
<td>Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_server_inq_bindings</td>
<td>Returns binding handles for communication with a server.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_inq_if</td>
<td>Returns the manager entry point vector registered for an interface.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_listen</td>
<td>Tells the RPC runtime to listen for remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_register_auth_info</td>
<td>Registers authentication information with the RPC runtime.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_register_if</td>
<td>Registers an interface with the RPC runtime.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_unregister_if</td>
<td>Unregisters an interface from the RPC runtime.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_use_all_protseqs</td>
<td>Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_use_all_protseqs_if</td>
<td>Tells the RPC runtime to use all the protocol sequences and endpoints specified in the interface specification for receiving remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_use_protseq</td>
<td>Tells the RPC runtime to use the specified protocol sequence for receiving remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_use_protseq_ep</td>
<td>Tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_server_use_protseq_if</td>
<td>Tells the RPC runtime to use the specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_set_ae_name</td>
<td>Sets the AENAME the server will use to identify equivalent instances to workload balancing.</td>
<td>Server</td>
</tr>
<tr>
<td>rpc_string_binding_compose</td>
<td>Combines the components of a string binding into a string binding.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_string_binding_parse</td>
<td>Returns, as separate strings, the components of a string binding.</td>
<td>Client or server</td>
</tr>
<tr>
<td>rpc_string_free</td>
<td>Frees a character string allocated by the runtime.</td>
<td>Client, server, or management</td>
</tr>
<tr>
<td>rpc_tower_to_binding</td>
<td>Returns a binding handle from a tower representation.</td>
<td>Client</td>
</tr>
</tbody>
</table>
### RPC Nonstub Support Routines

<table>
<thead>
<tr>
<th>Routines</th>
<th>Description</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_tower_vector_free</code></td>
<td>Releases memory associated with a tower vector.</td>
<td>Server</td>
</tr>
<tr>
<td><code>rpc_tower_vector_from_binding</code></td>
<td>Creates a tower vector from a binding handle.</td>
<td>Server</td>
</tr>
<tr>
<td><code>uuid_compare</code></td>
<td>Compares two UUIDs and determines their order.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_create</code></td>
<td>Creates a new UUID.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_create_nil</code></td>
<td>Creates a nil UUID.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_equal</code></td>
<td>Determines if two UUIDs are equal.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_from_string</code></td>
<td>Converts a string UUID to its binary representation.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_hash</code></td>
<td>Creates a hash value for a UUID.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_is_nil</code></td>
<td>Determines if a UUID is nil.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
<tr>
<td><code>uuid_to_string</code></td>
<td>Converts a UUID from a binary representation to a string representation.</td>
<td>Client, server, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>management</td>
</tr>
</tbody>
</table>

### Environment Variables

For detailed information on environment variables and how to set them, refer to [z/OS DCE Administration Guide](#). The RPC Name Service Interface (NSI) routines use the following environment variables:

- **RPC_DEFAULT_ENTRY**

  This environmental variable 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` argument, 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 argument `entry_name` in routine `rpc_ns_binding_import_begin`.

- **RPC_DEFAULT_ENTRY_SYNTAX**

  This environmental variable specifies the syntax for the name provided in the `RPC_DEFAULT_ENTRY` environment variable. In addition, it provides syntax for 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 a list of the valid name syntaxes, see [Table 2-6 on page 2-21](#).

Each of your applications will optionally define either or both of the first two environment variables. They can then change the value of either one, or both, during run time.
Data Types and Structures

This section describes the data types and structures used by client, server, and management application programs.

Much of the information in this section is derived from the RPC section of z/OS DCE Application Development Guide: Core Components. You may want to refer to it as you read this section. For example, this section contains a brief description of a binding handle. It also explains concepts related to binding handles, such as binding information and string bindings. The RPC section of z/OS DCE Application Development Guide: Core Components explains binding handles in detail.

**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 running 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 refers 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 fully bound. If the information lacks an endpoint, the binding handle is partially bound.

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 API 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 RPC section of z/OS 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 z/OS DCE Messages and Codes.

Binding information can contain an object UUID. The default object UUID associated with a binding handle is a nil UUID. Clients can obtain a non-nil UUID in various ways, such as from a string representation of binding information (a string binding), or by importing it.

The following table lists the RPC runtime routines that operate on binding handles. It also specifies the type of binding handle, client or server, allowed.

<table>
<thead>
<tr>
<th>Table 2-4 (Page 1 of 2), Client and Server Binding Handles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Routine</strong></td>
</tr>
<tr>
<td>rpc_binding_copy</td>
</tr>
<tr>
<td>rpc_binding_free</td>
</tr>
<tr>
<td>rpc_binding_from_string_binding</td>
</tr>
<tr>
<td>rpc_binding_inq_auth_client</td>
</tr>
<tr>
<td>rpc_binding_inq_auth_info</td>
</tr>
<tr>
<td>rpc_binding_inq_object</td>
</tr>
<tr>
<td>rpc_binding_reset</td>
</tr>
<tr>
<td>rpc_binding_server_from_client</td>
</tr>
<tr>
<td>rpc_binding_set_auth_info</td>
</tr>
<tr>
<td>rpc_binding_set_object</td>
</tr>
<tr>
<td>rpc_binding_to_string_binding</td>
</tr>
<tr>
<td>rpc_binding_vector_free</td>
</tr>
<tr>
<td>rpc_ns_binding_export</td>
</tr>
</tbody>
</table>
Table 2-4 (Page 2 of 2). Client and Server Binding Handles

<table>
<thead>
<tr>
<th>Routine</th>
<th>Input Argument</th>
<th>Output Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_ns_binding_import_next</td>
<td>none</td>
<td>server</td>
</tr>
<tr>
<td>rpc_ns_binding_inq_entry_name</td>
<td>server</td>
<td>none</td>
</tr>
<tr>
<td>rpc_ns_binding_lookup_next</td>
<td>none</td>
<td>server</td>
</tr>
<tr>
<td>rpc_ns_binding_select</td>
<td>server</td>
<td>server</td>
</tr>
<tr>
<td>rpc_server_inq_bindings</td>
<td>none</td>
<td>server</td>
</tr>
</tbody>
</table>

If an application provides the incorrect binding handle type, when the input argument type is either only client or only a server, the routine will return the `rpc_s_wrong_kind_of_binding` status code.

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 change a binding handle. The related routines are:

- `rpc_binding_free`
- `rpc_binding_reset`
- `rpc_binding_set_auth_info`
- `rpc_binding_set_object`
- `rpc_ep_resolve_binding`
- `rpc_mgmt_set_com_timeout`

Take, for example, an application that shares a binding handle across two threads of execution. It resets the binding handle endpoint in one of the threads (by calling the `rpc_binding_reset` API). The binding handle in the other thread will also be reset. Similarly, freeing the binding handle in one thread (by calling the `rpc_binding_free` routine) also frees the binding handle in the other thread.

If you do not want this behavior, your application can create a copy of a binding handle by calling routine `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 using the `rpc_binding_inq_object` and `rpc_binding_set_object` APIs.

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:

```c
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` API.

A client application obtains a binding vector of compatible servers from the name service database by calling the `rpc_ns_binding_lookup_next` API.

In both routines, the RPC runtime allocates storage for the binding vector. An application calls the `rpc_binding_vector_free` API 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 routine `rpc_binding_free` with the address of the binding handle in the binding vector. This routine also sets the corresponding pointer in the binding vector to `NULL`. 

2-10 Application Development Reference Volumes 1 and 2
Note: You should not decrement the field count in a binding vector structure when you call the routine rpc_binding_free to free an individual binding handle.

The following routines require a binding vector, and they show an argument data type of rpc_binding_vector_t:

- rpc_binding_vector_free
- rpc_ep_register
- rpc_ep_register_wlb
- rpc_ep_register_no_replace
- rpc_ep_register_no_replace_wlb
- 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 with 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 EBCDIC code sets. The RPC runtime automatically converts DCE PCS characters encoded in ASCII or 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 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, known as the code set registry. The code set registry assigns a unique identifier to each character set and code set. These identifiers are consistent across a network of heterogeneous operating systems, providing a way for clients and servers in such an environment to identify code sets without having to rely on operating system-specific string names. For more information on the code set registry, see refer to z/OS DCE Administration Guide.

The code set data structure contains a 32-bit hexadecimal value (c_set) that uniquely identifies the code set and 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. Following this information is a count of the number of character sets supported by this code set (ch_sets_num) and a pointer to a conformant array containing the 16-bit hexadecimal values that uniquely identify each of the character sets supported by 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()
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:

```c
typedef struct {
    unsigned32 c_set;
    unsigned16 c_max_bytes;
    unsigned16 ch_sets_num;
    [ptr,size_is(ch_sets_num)]unsigned16 *ch_sets;
} 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 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).

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.

The C language representation of a code set structure is as follows:

```c
typedef struct rpc_codeset_mgmt_t {
    unsigned32 version;
    unsigned32 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 the routine 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:

```c
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;

Note: z/OS DCE RPC-supplied buffer sizing routines do not support the idl_cs_in_place_convert conversion type. The reason is that the actual conversion method used (RMIR, SMIR, CMIR, or Universal) is determined at runtime. There is no guarantee that the conversion can be performed in a single storage area, especially in the case where an intermediate code set, such as UCS, is used as the network code page.

Endpoint Map Inquiry Handle: An endpoint map inquiry handle is a pointer-size opaque variable. It contains information the RPC runtime uses to access the elements in a local or remote endpoint map. The description of the rpc_ep_register API lists the contents of an element.

The following routines require an endpoint map inquiry handle, and they 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
- rpc_mgmt_ep_elt_inq_next_wlb

Global Name: The Name Service Interface (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 path name and a leaf name. (For a description of global names, see the NSI usage chapter of the RPC section in z/OS DCE Application Development Guide: Core Components). 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 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 will return global names.

For example, when you specify a cell-relative name as the member_name argument to the rpc_ns_group_mbr_add API, you can receive the corresponding global name by calling the rpc_ns_group_mbr_inq_next API.

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: In order to use the IDL encoding service, 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. It contains information the RPC runtime uses to access the interface specification data structure.

The DCE IDL compiler creates an interface specification data structure from each IDL file. It creates a global variable of type rpc_if_handle_t for the interface specification.

The compiler also places an interface handle declaration in the generated interface-name.h file. It generates this include 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:

- For the client:
  
  if-name_vmajor-version_minor-version_c_ifspec

- For the server:
if-name_vmajor-version_minor-version_s_ifspec

where:
- *if-name* is the interface identifier specified in the IDL file.
- *major-version* is the interface’s major-version number specified in the IDL file.
- *minor-version* is the interface’s minor-version number specified in the IDL file.

An example is notes_v1_2_c_ifspec.

The interface handle name can be a maximum of 31 characters. However, the interface name can be no more than 17 characters because the major-version and minor-version numbers must each be at least one character each and twelve characters are reserved for the format.

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_wlb`
- `rpc_ep_register_no_replace`
- `rpc_ep_register_no_replace_wlb`
- `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- 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:

```c
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 these routines, the application is responsible for providing storage for the structure.

The `rpc_if_inq_id` API returns the interface identifier from an interface specification. Other routines requiring an interface identifier include the following:

- `rpc_mgmt_ep_elt_inq_begin`
- `rpc_mgmt_ep_elt_inq_next`
- `rpc_mgmt_ep_elt_inq_next_wlb`
- `rpc_mgmt_ep_set_activated_wlb`
- `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 id vector contains a count member `count`, followed by an array of pointers to interface ids `rpc_if_id_t`.

The C language representation of an interface identifier vector is:

```c
typedef struct {
    unsigned32 count;
    rpc_if_id_t *if_id[1];
} rpc_if_id_vector_t;
```

The interface id vector is a read-only vector. To obtain a vector of the interface IDs registered by a server with the RPC runtime, an application calls the `rpc_mgmt_inq_if_ids` API. To obtain a vector of the interface IDs exported by a server to a name service database, an application calls the `rpc_ns_mgmt_entry_inq_if_ids` API.

The RPC runtime allocates storage for the interface id vector. The application calls the `rpc_if_id_vector_free` routine to free the interface id vector.

**Manager Entry Point Vector:** The manager Entry Point Vector (EPV) is an array of pointers to remote procedures.

The DCE IDL compiler 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:

```c
if-name_vmajor-version_minor-version_epv_t
```

where:

- `if-name` is the interface identifier specified in the IDL file.
- `major-version` is the interface’s major version number specified in the IDL file.
- `minor-version` is the interface’s minor version number specified in the IDL file.

By default, the DCE IDL compiler 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 EPV with addresses of local routines. Client applications can call these routines.

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_vmajor-versionMinor-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. It contains 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`
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, `ncadg` or `ncacn`.
- The format used in the network address specified in the binding, `ip`.
- The transport protocol used for communications, `udp` or `tcp`.

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 lists predefined strings representing valid protocol sequences. NCA is an abbreviation of **Network Computing Architecture**.

<table>
<thead>
<tr>
<th>Protocol Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ncadg_ip_udp</code></td>
<td>NCA Datagram over <strong>Internet Protocol</strong> (IP): <strong>User Datagram Protocol</strong> (UDP)</td>
</tr>
<tr>
<td><code>ncacn_ip_tcp</code></td>
<td>NCA Connection over Internet Protocol: <strong>Transmission Control Protocol</strong> (TCP)</td>
</tr>
</tbody>
</table>

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:

```c
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 \texttt{rpc_network_inq_protseqs} API. The RPC runtime allocates storage for the protocol sequence vector. To free the protocol sequence vector, the server application calls the \texttt{rpc_protseq_vector_free} API.

**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 (\texttt{count}), followed by an array of statistics. Each array element contains an \texttt{unsigned32} value. The following list describes the statistics indexed by the specified constant:

- \texttt{rpc\_c\_stats\_calls\_in} The number of remote procedure calls received by the runtime.
- \texttt{rpc\_c\_stats\_calls\_out} The number of remote procedure calls initiated by the runtime.
- \texttt{rpc\_c\_stats\_pkts\_in} The number of network packets received by the runtime.
- \texttt{rpc\_c\_stats\_pkts\_out} The number of network packets sent by the runtime.

The C language representation of a statistics vector is:

```c
typedef struct {
    unsigned32 count;
    unsigned32 stats[1];
} rpc_stats_vector_t;
```

To obtain runtime statistics, an application calls the \texttt{rpc_mgmt_inq_stats} API. The RPC runtime allocates storage for the statistics vector. To free the statistics vector, the application calls the \texttt{rpc_mgmt_stats_vector_free} API 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 \texttt{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 the \texttt{rpc\_binding\_to\_string\_binding} API 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 \texttt{rpc\_ep\_register}, \texttt{rpc\_ep\_register\_no\_replace}, and \texttt{rpc\_ns\_binding\_export} APIs. Instead, the server calls only the \texttt{rpc\_server\_inq\_bindings} API to obtain a vector of binding handles. The server obtains binding handles one at a time from the vector and calls the \texttt{rpc\_binding\_to\_string\_binding} API to convert each binding handle into a string binding. The resulting string binding is always fully bound and may contain a non-nil object UUID. The server then makes some or all of its string bindings available to clients. The server can place the string bindings in a file to be read by clients or users or both, or can deliver the string bindings to clients or users by means of a file, mail, or paper.

You can continue to develop the distributed application 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 the \texttt{rpc\_binding\_from\_string\_binding} API to convert the string binding into a server binding handle.

Other useful routines for working with string bindings are:

- \texttt{rpc\_string\_binding\_compose} Creates a string binding from its component parts.
- \texttt{rpc\_string\_binding\_parse} Separates a string binding into its component parts.

The two formats of a string binding are:

- \texttt{object-uuid@rpc-protocol-sequence: nw-addr[endpoint, option...]}

or
object-uuid@ rpc-protocol-sequence: nw-addr[endpoint =endpoint, option...]

where:

- **object-uuid** 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 call the correct manager routine. See ["rpc_server_register_if" on page 2-262] for an explanation on mapping object UUIDs to manager EPVs. This field is optional. If you do not provide it, the RPC runtime assumes a nil UUID.

- @ is the delimiter character for the object UUID field.

- **rpc-protocol-sequence** is a required sequence that specifies the protocol sequence used for making remote procedure calls. The valid protocol sequences are:
  - ncacn_ip_tcp and ncadg_ip_udp

  For more information about the valid protocol sequences see ["Protocol Sequence" on page 2-16]

  This field is required.

- : is the delimiter character for the **RPC protocol sequence** field.

- **nw-addr** 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**:
  - ncacn_ip_tcp and ncadg_ip_udp

    Specify an Internet address using the common Internet address notation or host name.

    An example with common Internet address notation is 128.10.2.30.

    An example with a host name is ko. If the specified host name is multihomed, the binding handle returned from the [rpc_binding_from_string_binding](#) routine 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 specify this field, the string binding refers to your local host.

- [ 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** 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:
  - ncacn_ip_tcp and ncadg_ip_udp

    Specify an Internet port number.

    An example of an Internet port number is 1025.

    The endpoint field is optional. For more information about endpoints, see ["Binding Handle" on page 2-9]

- , is the delimiter character specifying that option data follows. If an option follows, this delimiter is required.

- **option** specifies any options. Each option is specified as `option name=option value`.

  The format and content of the option depend on the specified protocol sequence:
  - ncacn_ip_tcp and ncadg_ip_udp

    There are no Internet options.

    The option field is optional.

    - ] 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 filtered out for all string binding fields. A string binding does not contain any white space.

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('@ncadg_ip_udp:16.20.16.27)[1551]
- obj-uuid('@ncacn_ip_tcp:16.20.16.27)[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 hyphens separate the fields. An example of a string UUID is:

989c6e5c-2cc1-11ca-a044-08002b1bb4f5

When you specify a string UUID as an input argument to an RPC runtime routine, you can enter the alphabetic hexadecimal characters in either uppercase or lowercase. 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. 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:

```c
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

Normally with z/OS DCE, to register endpoints to or unregister endpoints from the endpoint map, you require sufficient Access Control List (ACL) permissions. You also require ACL permissions to access entries in a Cell Directory Service (CDS) database using the Name Service Interface (NSI) routines. If your DCE administrator configures your z/OS machine to allow any DCE principal (authenticated or unauthenticated) to register or unregister endpoints to the endpoint map, the principals running your applications do not require any special permissions to access the endpoint map. Note that the ACLs that protect the endpoint map or the CDS namespace can be modified at any time, usually by your DCE administrator. Consult your DCE administrator on the permissions required for your environment.
**DCED Endpoint Map Permissions:** z/OS DCE has implemented access control to the DCE Host Daemon daemon that runs on z/OS. For server applications that access the endpoint map, which is controlled by the DCE Host Daemon, the principal name (or the group to which it belongs) running the application requires the **insert** permission to register the server endpoint and the **delete** permission to unregister the server endpoint. **Server principals** should use a special **server** permission that combines the insert and delete permission. There is also a Security group (**rpc_server_group**) that is defined during the configuration of your machine which contains the necessary permissions to the DCE Host Daemon endpoint map. Your server application inherits this group’s permissions to the DCE Host Daemon endpoint map if it becomes a member of this group. See [z/OS DCE Administration Guide](#) for information on granting these permissions to DCE server principals.

The name service entry that points to the DCE Host Daemon endpoint map maintained on a z/OS machine is `./hosts/<hostname>/config/epmap`, where `<hostname>` is the DCE host name of interest. For example, to list the ACL entries associated with the DCE Host daemon that runs on a machine with host name **bass**, use the following command:

```
  acl_edit ./hosts/bass/config/epmap -l
```

**Name Service Interface Permissions:** To use the Name Service Interface (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:

- 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:** **Write** permission does not imply **read** permission.

To find the ACL permissions for the NSI routines (whose names begin with **rpc_ns**), see [Permissions Required](#) on page 2-19.

Because the non-NSI routines (whose names do not begin with **rpc_ns**) do not need ACL permissions, their reference information does not include a Permissions Required section.

**Frequently Used Routine Arguments**

A few arguments are common to many of the DCE RPC routines. These arguments are described fully here and again briefly when the routine is described.

- **binding**
  - Used as an input or output argument.
  - 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 communication between the client and server. The communication occurs when a client makes a remote procedure call.
  - As an input argument 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 or output argument.

When used as an input argument, the value of this argument depends on the syntax selected in the name_syntax argument. 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 the argument name.

As an output argument, it returns an entry in the name service database in the form of a character string that includes an ending null character. The value of this argument 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 storage for the returned string. The application is responsible for calling the rpc_string_free API to deallocate the string.

If an application does not want a returned name string, the application usually specifies NULL for this argument. The one exception is the rpc_ns_entry_expand_name API. It always returns a name string.

name_syntax

Used only as an input argument.

It is 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>
<thead>
<tr>
<th>Constant Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_ns_syntax_default</td>
<td>0</td>
</tr>
<tr>
<td>rpc_c_ns_syntax_dce</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2-6. Valid Name Syntaxes

The name_syntax argument tells routines how to parse the entry name specified in an input name argument, or specifies the syntax to use when returning an entry name as an output name argument.

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 argument.

It returns a character string, which always includes the ending null character \0. The DCE RPC runtime allocates storage for the returned string. The application calls the rpc_string_free routine to deallocate the storage 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 storage occupied by this string.

If an application does not require a returned output string, the application specifies NULL for this argument.

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: 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_.

Chapter 2. Remote Procedure Call  2-21
For example, the exception `rpc_x_already_listening` is equivalent to the status code `rpc_s_already_listening`.

The documentation for each status code includes the message text, the explanation, and the suggested user action. These can be found in [z/OS DCE Messages and Codes](#).

**uuid**

Used as an input or output argument.

When you need to specify a nil UUID to a `uuid` input argument in any of the DCE RPC routines, you can specify the value `NULL`.

### Related Information

**Books**

- “RPC Section,” [z/OS DCE Application Development Guide: Core Components](#)
- [z/OS DCE Command Reference](#)
cs_byte_from_netcs

Purpose
Converts international character data from a network code set to a local code set.

Used by client or server applications.

Format
#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. When called from the server stub, this value is a pointer to binding information that the client stub passed in the RPC call.

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.

Output
local_data
A pointer to the converted data, in idl_byte format.

local_data_length
The length of the converted data, in units of idl_byte. Specify NULL if a fixed 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.

The possible status codes and their meanings are as follows:

rpc_s_ok
Success.
cs_byte_from_netcs

**rpc_s_ss_incompatible_codesets**  
The specified code set does not match the code set specified in the sending tag in the binding handle. 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**  
Invalid character input for conversion (for example, multibyte input or lock-shift sequence that spans the input buffer boundary, or an input character not in the set of characters being converted from.)

**rpc_s_ss_short_conv_buffer**  
No room left in the output buffer to place the converted characters.

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.

**Usage**

Applications should not need to call this API directly.

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 **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 **idl_byte** format. If the data is a varying, 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** (the local data type 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.

If you are using PDS (as opposed to HFS), modify the **include** statement in the format to be `#include <dce/codestub.h>`.

**Permissions Required**

No permissions are required.

**Return Values**

None.

**Related Information**
Routines

cs_byte_to_netcs
Purpose
Calculates the necessary buffer size for code set conversion from a network code set to a local code set.

Used by client or server applications.

Format
#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. When called from the server stub, this value is a pointer to binding information that the client stub passed in the RPC call.

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_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 idl_byte. This value is to be used as the local value of the size_is variable for the array, and is non-NULL 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.

The possible status codes and their meanings are as follows:

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The specified code set does not match the code set specified in the sending tag in the binding handle. 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.

**Usage**

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 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 new one is required); 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 `cs_byte_*` 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.

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 `c_max_bytes` value for the code set specified in `network_code_set_value` and by an expansion factor of 2.

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 `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_no_convert`, the routine assumes that `network_buffer_size` can store the data (because no conversion is necessary) and returns `idl_cs_no_convert` in the `conversion_type` parameter. The routine also returns the value of `network_buffer_size` in `local_buffer_size` if a conformant or conformant varying array is to be marshalled.

In cases where the binding handle does not contain the results of character and code sets evaluation, or where it is being called from the server stub, the `cs_byte_local_size()` routine 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, the routine calculates the size of this new buffer (by multiplying the value of `network_buffer_size` by the network code set `c_max_bytes` value and an expansion factor of 2) and returns the results, in units of `cs_byte`, in `local_buffer_size`.

If you are using PDS (as opposed to HFS), modify the `include` statement in the format to be `#include <dce/codestub.h>`.

**Permissions Required**

No permissions are required.

**Return Values**

None.

**Related Information**

Chapter 2. Remote Procedure Call 2-27
cs_byte_local_size

Routines

cs_byte_net_size
cs_byte_net_size

Purpose
Calculates the necessary buffer size for code set conversion from a local code set to a network code set.
Used by client or server applications.

Format
```
#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. When called from the server stub, this value is a pointer to binding information that the client stub passed in the RPC call.

* 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 *idl_byte*, of the buffer that is allocated for the international character data. 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_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.

The possible status codes and their meanings are as follows:

  * rpc_s_ok Success.
  * rpc_s_ss_incompatible_codesets The specified code set does not match the code set specified in the sending tag in the binding handle. 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_net_size() routine raises an exception to the client application.

Usage

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 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 new one is required). 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 cs_byte_*_size routines when the cs_byte type (which is equivalent to byte) has been specified as the local data type using the cs_char attribute in the attribute configuration file for the application.

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 idl_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 c_max_bytes value for the code set specified in network_code_set_value (the sending tag parameter) and by an expansion factor of 2.

The routine returns the new buffer size in the network_buffer_size parameter. The size is specified in units of cs_byte, which is the network representation used for international character data (and is equivalent to the byte type). For fixed and varying arrays, the routine assumes that local_buffer_size is sufficient to store the converted data.

If the handle information specifies idl_cs_no_convert, the routine assumes that local_buffer_size can store the converted data (because no conversion is necessary) and returns idl_cs_no_convert in the conversion_type parameter. The routine also returns the value of local_buffer_size in network_buffer_size if a conformant or conformant varying array is to be marshalled.

In cases where the binding handle does not contain the results of character and code sets evaluation, or where it is being called from the server stub, the cs_byte_net_size() routine 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, the routine calculates the size of this new buffer (by multiplying the value of local_buffer_size by the network code set c_max_bytes value and an expansion factor of 2) and returns the results, in units of cs_byte, in network_buffer_size.

If you are using PDS (as opposed to HFS), modify the include statement in the format to be #include <dce/codestub.h>.

Permissions Required

No permissions are required.

Return Values

None.

Related Information
Routines

 cs_byte_local_size
cs_byte_to_netcs

**Purpose**
Converts international character data from a local code set to a network code set.

Used by client or server applications.

**Format**
```c
#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. When called from the server stub, this value is a pointer to binding information that the client stub passed in the RPC call.

- **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 `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.

**Output**

- **network_data**
  A pointer to the converted data, in units of `idl_byte`.

- **network_data_length**
  The length of the converted data, in units of `idl_byte`. Specify NULL if a fixed 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.

The possible status codes and their meanings are as follows:

- **rpc_s_ok**
  Success.

- **rpc_s_ss_incompatible_codesets**
  The specified code set does not match the code set specified in the sending tag in the binding handle. 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:
cs_byte_to_netcs

rpc_s_ss_invalid_char_input
Invalid character input for conversion (for example, multibyte input or lock-shift sequence that spans the input buffer boundary, or an input character not in the set of characters being converted from.)

rpc_s_ss_short_conv_buffer
No room left in the output buffer to place the converted characters.

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.

Usage

Applications should not need to call this API directly.

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 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 varying, 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 (the local data type 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.

If you are using PDS (as opposed to HFS), modify the include statement in the format to be #include <dce/codestub.h>.

Permissions Required
No permissions are required.

Return Values
None.

Related Information
Routines

cs_byte_to_netcs

cs_byte_from_netcs
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 or server applications.

Format

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

void dce_cs_loc_to_rgy(
    id1_norm_char *local_code_set_name,
    unsigned32 *rgy_code_set_value,
    unsigned16 *rgy_char_sets_number,
    unsigned16 **rgy_char_sets_value,
    error_status_t *status);
```

Parameters

**Input**

*local_code_set_name* A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 character data bytes plus a terminating NULL character.

**Output**

*rgy_code_set_value* The registered integer value that uniquely identifies the code set specified by *local_code_set_name*.

*rgy_char_sets_number* The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.

*rgy_char_sets_value* A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter. The routine dynamically allocates this value.

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

The possible status codes and their meanings are as follows:

- **dce_cs_c_ok** Code set registry access succeeded.
- **dce_cs_c_cannot_allocate_memory** Cannot allocate memory for code set info.
- **dce_cs_c_cannot_open_file** Cannot open the code set registry file (path name may be incorrect or the file's permissions may have prevented access). If the code set registry is not in the default location, `/usr/lib/nls/csr/code_set_registry.db` use the **dce_cf_get_crgy_filename** routine to get the correct path name.
- **dce_cs_c_cannot_read_file** Cannot read the code set registry file
- **dce_cs_c_unknown** The requested value, for example local code set name or registry code set value, was not found in the code set registry.
dce_cs_loc_to_rgy

Usage

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 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. (See "cs_byte_from_netcs" on page 2-23 and "cs_byte_to_netcs" on page 2-32.)

Permissions Required

No permissions are required.

Return Values

None.

Related Information

Routines

- dce_cs_rgy_to_loc
- rpc_cs_char_set_compat_check
- rpc_cs_eval_with_universal
- rpc_cs_eval_without_universal
- rpc_rgy_get_code_sets
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 or server applications.

Format
#include <dce/rpc.h>

void dce_cs_rgy_to_loc(
    unsigned32 rgy_code_set_value,
    idl_norm_char *local_code_set_name,
    unsigned16 rgy_char_sets_number,
    unsigned16 *rgy_char_sets_value,
    error_status_t *status);

Parameters

Input
rgy_code_set_value The registered hexadecimal value that uniquely identifies the code set.

Output
local_code_set_name A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 character data bytes and a terminating NULL character.
rgy_char_sets_number The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.
rgy_char_sets_value A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value. The routine dynamically allocates this value.
status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

dce_cs_c_ok Code set registry access succeeded.
dce_cs_c_cannot_allocate_memory Cannot allocate memory for code set info.
dce_cs_c_cannot_open_file Cannot open the code set registry file (path name may be incorrect or the file's permissions may have prevented access. If the code set registry is not in the default location, /usr/lib/nls/csr/code_set_registry.db use the dce_cf_get_csgry_filename routine to get the correct path name.
dce_cs_c_cannot_read_file Cannot read the code set registry file
A read error occurred when reading the code set registry (probably a problem in the underlying operating system).
dce_cs_c_unknown The requested value, for example local code set name or registry code set value, was not found in the code set registry.
Usage

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 interoperability 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. (See "cs_byte_to_netcs" on page 2-32 and "cs_byte_from_netcs" on page 2-23.)

Permissions Required

No permissions are required.

Return Values

None.

Related Information

Routines

- `dce_cs_loc_to_rgy`
- `rpc_cs_char_set_compat_check`
- `rpc_cs_eval_with_universal`
- `rpc_cs_eval_without_universal`
- `rpc_rgy_get_code_sets`
idl_es_decode_buffer

Purpose
Returns a buffer decoding handle to the IDL encoding services.

Used by client or server applications.

Format

```c
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.

The possible status codes and their meanings include:

- rpc_s_ok Success.
- rpc_s_ss_bad_buffer Bad buffer operation.
- rpc_s_no_memory Insufficient memory available to complete operation.

Usage

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.

Related Information
idl_es_decode_buffer

Routines

idl_es_decode_incremental
idl_es_decode_incremental

Purpose
Returns an incremental decoding handle to the IDL encoding services.
Used by client or server applications.

Format
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.

Parameters for the read_fn function:
typedef void (*idl_es_read_fn_t)
    (idl_void_p_t state, /qc@1>4 in/out qc@1>4/
     idl_byte buffer, /qc@1>4 in qc@1>4/
     idl_ulong_int size, /qc@1>4 in qc@1>4/);

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.

The possible status code and its meaning is as follows:
rpc_s_ok Success.
rpc_s_no_memory Insufficient memory available to complete operation.
idl_es_decode_incremental

Usage

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.

Related Information

Routines

idl_es_encode_incremental idl_es_decode_buffer
idl_es_encode_dyn_buffer

**Purpose**
Returns a dynamic buffer encoding handle to the IDL encoding services.

Used by client or server applications.

**Format**
```c
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.

- **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.

  The possible status codes and their meanings are as follows:
  - `rpc_s_ok`
    Success.
  - `rpc_s_ss_bad_buffer`
    Bad buffer operation.
  - `rpc_s_no_memory`
    Insufficient memory available to complete operation.

**Usage**
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.
Related Information

Routines

idl_es_encode_dyn_buffer

idl_es_encode_fixed_buffer

idl_es_encode_incremental
idl_es_encode_fixed_buffer

Purpose
Returns a fixed buffer encoding handle to the IDL encoding services.

Used by client or server applications.

Format
```c
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.

The possible status codes and their meanings are as follows:
- `rpc_s_ok`: Success.
- `rpc_s_bad_buffer`: Bad buffer operation.
- `rpc_s_no_memory`: Insufficient memory available to complete operation.

Usage
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_esEncodeFixedBuffer()` 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_esEncodeFixedBuffer()` 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).
idl_es_encode_fixed_buffer

Return Values

None.

Related Information

Routines

idl_es_encode_dyn_buffer          idl_es_encode_incremental
idl_es_encode_incremental

Purpose
Returns an incremental encoding handle to the IDL encoding services.
Used by client or server applications.

Format
```
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 outline 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
idl_es_encode_incremental

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.

The possible status codes and their meanings are as follows:

- `rpc_s_ok` Success.
- `rpc_s_no_memory` Insufficient memory available to complete operation.

**Usage**

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.

**Related Information**

**Routines**

- `idl_es_decode_incremental`
- `idl_es_encode_fixed_buffer`
- `idl_es_encode_dyn_buffer`
idl_es_handle_free

Purpose
Frees an IDL encoding services handle.
Used by client or server applications.

Format
```c
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.

  The possible status code and its meaning is as follows:

  - `rpc_s_ok`:
    Success.

Usage

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.

Related Information

Routines

- `idl_es_decode_buffer`
- `idl_es_decode_incremental`
- `idl_es_encode_dyn_buffer`
- `idl_es_encode_fixed_buffer`
- `idl_es_encode_incremental`
Purpose
Return the attribute flags from an IDL encoding services handle.
Used by client or server applications.

Format
void idl_es_inq_attrs(
    idl_es_handle_t es_handle,
    unsigned32 *flags,
    error_status_t *status);

Parameters
Input
es_handle
An encoding services handle returned by one of the IDL encoding services routines.

Output
flags
Returns the flags stored in the IDL encoding services handle that are 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.

The possible status codes and their meanings are as follows:
rpc_s_ok
Success.

Usage
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_attrs routine returns the flags stored in the IDL encoding services handle that is associated with the encoded
data. Applications can use the idl_es_set_attrs routine to set the flags. Currently the only valid flag that can be set is
IDL_ES_NO_ENCODING_CHECK which tells the encoding services to not check the interface ID (UUID and version number)
for validity. This allows an interface that did not encode the data to decode it or parts of it, such as a common header.

Return Values
None.

Related Information
Routines

idl_es_set_attrs

idl_es_set_transfer_syntax
idl_es_inq_encoding_id

Purpose
Identifies an operation within an interface that has been called to encode data using the IDL encoding services.

Used by client or server applications.

Format
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
An encoding services handle returned by one of the IDL encoding services 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.

The possible status codes and their meanings are as follows:

rpc_s_ok Success.
rpc_s_unknown_if Interface identifier and operation number unavailable.

Usage
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.

Related Information
## Routines

- `idl_es_decode_buffer`
- `idl_es_decode_incremental`
- `idl_es_encode_dyn_buffer`
- `idl_es_encode_fixed_buffer`
- `idl_es_encode_incremental`
idl_es_set_attrs

Purpose
Set attribute flags from an IDL encoding services handle.

Used by client or server applications.

Format

void idl_es_set_attrs(
    idl_es_handle_t es_handle,
    unsigned32 flags,
    error_status_t *status);

Parameters

Input
es_handle       An encoding services handle returned by one of the IDL encoding services routines.

Output
flags           Specifies the flags to be stored in association with the IDL encoding services handle.
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.

Usage

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_set_attrs routine sets the attribute flags to be stored in the IDL encoding services handle that is associated with
the encoded data. Applications can use the idl_es_inq_attrs routine to retrieve the flags set by this routine. Currently, the
only valid flag that can be set is IDL_ES_NO_ENCODING_CHECK which tells the encoding services to not check the interface
ID (UUID and version number) for validity. This allows an interface that did not encode the data to decode it or parts of it, such
as a common header.

Return Values
None.

Related Information
Routines

idl_es_inq_attrs       idl_es_set_transfer_syntax
idl_es_set_transfer_syntax

Purpose
Set a transfer syntax.

Used by client or server applications.

Format

```c
void idl_es_set_transfer_syntax(
    idl_es_handle_t es_handle,
    idl_es_transfer_syntax_t es_transfer_syntax,
    error_status_t *status);
```

Parameters

**Input**

- `es_handle` Specifies the user’s encoding services handle.
- `es_transfer_syntax` Specifies the requested transfer syntax.

**Output**

- `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_tsyntaxes_unsupported` Transfer syntax specified is not supported.

Usage

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_set_transfer_syntax` routine is used on platforms which support encoding in more than one transfer syntax. It must be called before encoding data if a transfer syntax other than the default is desired. If only one transfer syntax is supported by the platform, this routine may optionally be used to explicitly specify this transfer syntax.

**Note:** The only transfer syntax currently supported is `idl_es_transfer_syntax_ndr`.

Return Values

None.

Related Information
Routines

idl_es_inq_attrs  idl_es_set_attrs
Purpose
Returns a copy of a binding handle.
Used by client or server applications.

Format
#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 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.

Possible status codes and their meanings:
- rpc_s_ok Successful completion.
- rpc_s_invalid_binding Binding handle not valid.
- rpc_s_wrong_kind_of_binding Wrong kind of binding for operation.
- rpc_s_no_memory No storage available.
- rpc_s_invalid_arg Argument not valid.

Usage
The rpc_binding_copy API copies the server binding information referred to 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 API if you want a change (made to binding information by one thread) not to affect the binding information used by other threads. See "Binding Handle" on page 2-9 for more details about the use of the rpc_binding_copy API.

After this routine is called, operations performed on the source Binding handle do not affect the binding information referred to by the destination Binding handle. Similarly, operations performed on the destination Binding handle do not affect the binding information referred to by the source 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 has finished using the binding handle specified by the destination Binding parameter, the application calls the rpc_binding_free API to release the storage used by the destination Binding binding handle and the binding information it refers to.
Return Values

None.

Related Information

Routines

rpc_binding_free
**rpc_binding_free**

**Purpose**
Releases server binding handle resources.

Used by client or server applications.

**Format**
```c
#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.

  Possible status codes and their meanings:
  - `rpc_s_ok`: Successful completion.
  - `rpc_s_invalid_binding`: Binding handle not valid.
  - `rpc_s_wrong_kind_of_binding`: Wrong kind of binding for operation.

**Usage**

The `rpc_binding_free` API frees the storage used by a server binding handle and the binding information that it refers to. Use this routine when your application has finished using a server binding handle that was dynamically created during the running of the program.

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**

None.

**Related Information**

2-60 Application Development Reference Volumes 1 and 2
## Routines

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_binding_copy</code></td>
<td><code>rpc_ns_binding_import_next</code></td>
<td><code>rpc_ns_binding_select</code></td>
</tr>
<tr>
<td><code>rpc_binding_from_string_binding</code></td>
<td><code>rpc_ns_binding_lookup_next</code></td>
<td><code>rpc_server_inq_bindings</code></td>
</tr>
<tr>
<td><code>rpc_binding_vector_free</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**rpc_binding_from_string_binding**

**Purpose**
Returns a server binding handle from a string representation of a binding handle.

Used by client or management applications.

**Format**

```c
#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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_invalid_arg` Argument not valid.
- `rpc_s_invalid_endpoint_format` Endpoint format not valid.
- `rpc_s_inval_net_addr` Network address not valid.
- `rpc_s_invalid_rpc_protseq` Protocol sequence not valid.
- `rpc_s_invalid_string_binding` String binding not valid.
- `rpc_s_protseq_not_supported` Protocol sequence not supported on this host.
- `uuid_s_bad_version` Incorrect UUID version.
- `uuid_s_invalid_string_uuid` Format for a string UUID not valid.
- `rpc_s_no_memory` No storage available.

**Usage**

The **rpc_binding_from_string_binding** API 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 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** API or call the **rpc_binding_to_string_binding** API or provide a character string constant.
When an application finishes using the binding parameter, the application calls the `rpc_binding_free` API to release the storage used by the binding handle.

See "Binding Handle" on page 2-9 for an explanation of partially and fully bound binding handles.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_binding_copy`
- `rpc_binding_free`
- `rpc_binding_to_string_binding`
- `rpc_string_binding_compose`
rpc_binding_inq_auth_caller

Purpose
Returns authentication and authorization information from the binding handle for an authenticated client.

Used by server applications.

Format
```
#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 is 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 is 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_handle. 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_cdmf_priv
Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than rpc_c_protect_level_pkt_privacy. This is the second highest protection level, but it will be available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.

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_handle.

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_dce_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 using the client's DCE Extended Privilege Attribute Certificate (EPAC) 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
Success.

rpc_s_invalid_binding
Invalid 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.
rpc_binding_inq_auth_caller

Usage

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()` call or the `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 IDL `auto_handle` or `implicit_handle` attribute, 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.

Related Information

Routines

- `rpc_binding_inq_auth_info`
- `rpc_binding_set_auth_info`
- `rpc_string_free`
- `sec_cred_get_initiator`
- `sec_cred_get_delegate`
rpc_binding_inq_auth_client

Purpose

Returns authentication and authorization information from the client binding handle for an authenticated client.

Used by server applications.

Note: This call is provided only for compatibility with pre-release 1.1 DCE applications. DCE release 1.1 and later applications should use the rpc_binding_inq_auth_caller() call.

Format

```
#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>
<thead>
<tr>
<th>For authz_svc value:</th>
<th>privs contains this data:</th>
<th>Use this cast:</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_authz_none</td>
<td>A NULL value.</td>
<td>None.</td>
</tr>
<tr>
<td>rpc_authz_name</td>
<td>The calling client’s principal name.</td>
<td>(unsigned_char_t *)</td>
</tr>
<tr>
<td>rpc_authz_dce</td>
<td>The calling client’s privilege attribute certificate (EPAC)</td>
<td>(sec_id_pac_t *)</td>
</tr>
</tbody>
</table>

Note: `rpc_c_authz_none` is valid only if the `authn_svc` parameter is `rpc_c_authn_none`

The data referred to by this parameter is read-only and should not be changed by the server. If the server wants to preserve any of the returned data, it must copy the data into server-allocated storage. 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 is 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` API.

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:

- **rpc_c_protect_level_default**: Use the default protection level for the specified authentication service.
- **rpc_c_protect_level_none**: Perform no protection.
- **rpc_c_protect_level_connect**: Perform protection only when the client establishes a relationship with the server.
- **rpc_c_protect_level_call**: Perform protection only at the beginning of each remote procedure call when the server receives the request.
- **rpc_c_protect_level_pkt**: Ensure that all data received is from the expected client.
- **rpc_c_protect_level_pkt_integ**: Ensure and verify that none of the data transferred between client and server was changed.
- **rpc_c_protect_level_cdmf_priv**: Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than **rpc_c_protect_level_pkt_privacy**. This is the second highest protection level, but it will be available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.
- **rpc_c_protect_level_pkt_privacy**: Perform 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:

- **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:

- **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 (EPAC) 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.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_invalid_binding**: Binding handle not valid.
Usage

The rpc_binding_inq_auth_client API 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.

The RPC runtime allocates storage for the returned server_princ_name parameter. The server is responsible for calling the rpc_string_free API for the returned parameter string.

For applications in which the client side uses the IDL auto_handle or implicit_handle attribute, 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

None.

Related Information

Routines

rpc_binding_inq_auth_info rpc_binding_set_auth_info rpc_string_free
rpc_binding_inq_auth_info

Purpose
Returns authentication and authorization information from a server binding handle.
Used by client or server applications.

Format
#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 referred to 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 API.

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.
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 upgraded to the next higher supported level.
Specifying NULL prevents the routine from returning this parameter.
The possible protection levels are:

- rpc_c_protect_level_default: Use the default protection level for the specified authentication service.
- rpc_c_protect_level_none: Perform no protection.
- rpc_c_protect_level_connect: Perform protection only when the client establishes a relationship with the server.
- rpc_c_protect_level_call: Perform protection only at the beginning of each remote procedure call when the server receives the request.
- rpc_c_protect_level_pkt: Ensure that all data received is from the expected client.
- rpc_c_protect_level_pkt_integ: Ensure and verify that none of the data transferred between client and server has been changed.
rpc_c_protect_level_cdmf_priv

Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than \texttt{rpc_c_protect_level_pkt_privacy}. This is the second highest protection level, but it will be available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.

\texttt{rpc_c_protect_level_pkt_privacy}

Perform protection as specified by all of the previous levels and also encrypt each remote procedure call parameter value.

\texttt{authn_svc}

Returns the authentication service used for remote procedure calls made with \textit{binding}.

Specifying NULL prevents the routine from returning this parameter.

The possible authentication services are:

- \texttt{rpc_c_authn_none} No authentication.
- \texttt{rpc_c_authn_dce_secret} DCE shared-secret key authentication.
- \texttt{rpc_c_authn_dce_public} DCE public key authentication (reserved for future use).
- \texttt{rpc_c_authn_default} DCE default authentication service.

\texttt{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 by \texttt{rpc_binding_set_auth_info}.

If you use the \texttt{rpc_c_authn_dce_secret} authentication service and any authorization service, you must set the \texttt{sec_login_handle_t} value obtained from one of the following routines:

- \texttt{sec_login_setup_identity}
- \texttt{sec_login_get_current_context}
- \texttt{sec_login_newgroups}

These routines are described in Chapter 6, "Security and Related Services" on page 6-1.

Specifying NULL prevents the routine from returning this parameter.

\texttt{authz_svc}

Returns the authorization service used for remote procedure calls made with \textit{binding}.

Specifying NULL prevents the routine from returning this parameter.

The possible authorization services are:

- \texttt{rpc_c_authz_none} Server performs no authorization. This is valid only if the \texttt{authn_svc} parameter is \texttt{rpc_c_authn_none}.
- \texttt{rpc_c_authz_name} Server performs authorization based on the client principal name.
- \texttt{rpc_c_authz_dce} Server performs authorization using the client's DCE privilege attribute certificate (EPAC) sent to the server with each remote procedure call made with \textit{binding}. Generally, access is checked against DCE access control lists (ACLs).

\texttt{status}

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

Possible status codes and their meanings:

- \texttt{rpc_s_ok} Successful completion.
- \texttt{rpc_s_invalid_binding} Binding handle not valid.
- \texttt{rpc_s_wrong_kind_of_binding} Wrong kind of binding for operation.
- \texttt{rpc_s_binding_has_no_auth} Caller has no authority.
rpc_binding_inq_auth_info

Usage

The rpc_binding_inq_auth_info API 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 API.

The RPC runtime allocates storage for the returned server_princ_name parameter. The caller is responsible for calling the rpc_string_free API for the returned parameter string.

Return Values

None.

Related Information

Routines

rpc_binding_set_auth_info    rpc_string_free
**rpc_binding_inq_object**

**Purpose**
Returns the object UUID from a binding handle.

Used by client or server applications.

**Format**

```c
#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.

  Possible status codes and their meanings:

  - **rpc_s_ok**
    Successful completion.
  - **rpc_s_invalid_binding**
    Binding handle not valid.

**Usage**

The `rpc_binding_inq_object` API 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**

None.

**Related Information**

**Routines**

- **rpc_binding_set_object**
rpc_binding_reset

Purpose
Resets a server binding handle so the host remains specified, but the server instance on that host is unspecified.

Used by client or management applications.

Format
#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.

Possible status codes and their meanings:
rpc_s_ok Successful completion.
rpc_s_invalid_binding Binding handle not valid.
rpc_s_wrong_kind_of_binding Wrong kind of binding for operation.

Usage
The rpc_binding_reset API disassociates a server instance from the server binding handle specified in the binding. 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. See "Binding Handle" on page 2-9 for 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 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 routine 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

None.

Related Information

Routines

rpc_ep_register  rpc_ep_register_no_replace
**rpc_binding_server_from_client**

**Purpose**
Converts a client binding handle to a server binding handle.

Used by server applications.

**Format**

```c
#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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_cant_getpeername` Cannot get peer name.
- `rpc_s_connection_closed` Connection closed.
- `rpc_s_invalid_binding` Binding handle not valid.
- `rpc_s_wrong_kind_of_binding` Wrong kind of binding.

**Usage**

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` API 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` API 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` API 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. 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 run 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 `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 `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 affects the selection of a manager for a remote procedure call. See the section about the Internals of Remote Procedure Calls in the RPC module of [z/OS 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 kept in the server binding information. This server binding information is newly constructed by the `rpc_binding_server_from_client` API. 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 non-nil type, the server must also:

1. Specify the type for the manager that uses that interface (by calling `rpc_server_register_if`).
2. 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:

- 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` API 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` API, add the object UUID from the call argument to the newly constructed server binding information (by calling `rpc_binding_set_object`).

When your application finishes using the `server_binding` parameter, it should call the `rpc_binding_free` API to release the storage used by the binding handle.

**Return Values**

None.
rpc_binding_server_from_client

Related Information

Routines

<table>
<thead>
<tr>
<th>rpc_binding_free</th>
<th>rpc_ep_register</th>
<th>rpc_ep_register_no_replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_binding_set_object</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
rpc_binding_set_auth_info

Purpose
Sets authentication and authorization information for a server binding handle.
Used by client applications.

Format

```c
#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 referred to by `binding`. The content of the name and its syntax are 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` API 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, the client does not care which server principal received the remote procedure call request. The server still verifies the identity of the client.

  **Note:** If you specify `NULL`, that is no principal name is specified, the RPC program will make a call to `rpc_mgmt_inq_server_princ_name`. The purpose of specifying `NULL` here is primarily programming convenience because it saves time calling `rpc_mgmt_inq_server_princ_name`. You can use a `NULL` server name only if the `binding` handle is either fully bound or is bound to a non-null object UUID.

- **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 upgraded to the next higher supported level. The possible protection levels are:

  - `rpc_c_protection_level_default`
    Uses the default protection level for the specified authentication service.
    
    The default protection level for the DCE shared-secret key authentication service is `rpc_c_protection_level_pkt_integ`.

  - `rpc_c_protection_level_none`
    Performs no authentication: tickets are not exchanged, *session keys* are not established, client EPACs or names are not certified, and transmissions are in the clear. Although uncertified EPACs should
not be trusted, they may be useful for debugging, tracing, and measurement purposes.

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

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

**rpc_c_protect_level_pkt**
Ensure that all data received is from the expected client.

**rpc_c_protect_level_pkt_integ**
Ensure and verify that none of the data transferred between client and server has been changed.

This is the highest protection level that is guaranteed to be present in the RPC runtime.

**rpc_c_protect_level_cdmf_priv**
Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than **rpc_c_protect_level_pkt_privacy**. This is the second highest protection level, but it will be available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.

**rpc_c_protect_level_pkt_privacy**
Perform 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 level of protection provided by the authentication service is specified by the **protect_level** parameter. The supported authentication services are:

**rpc_c_authn_none**
Performs no authentication: Tickets are not exchanged, session keys are not established, client EPACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for the remote procedure calls made using **binding**.

**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

If **rpc_c_authn_default** is specified, the RPC runtime uses the **rpc_c_authn_dce_secret** authentication service for remote procedure calls made using **binding**.

**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**

These routines are described in Chapter 6, “Security and Related Services” on page 6-1.

Specify **NULL** to use the default security login context for the current address space.
authz_svc

Specifies the authorization service carried out by the server for the interface of interest. The validity and trustworthiness of authorization data, like any application data, depend on authentication service and protection level specified. The supported authorization services are:

- **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 (EPAC) 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.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_invalid_binding**: Binding handle not valid.
- **rpc_s_invalid_arg**: Argument not valid, may have a NULL pointer to status return code.
- **rpc_s_no_memory**: No storage available.
- **rpc_s_mutex_init_fail**: POSIX error: cannot initialize mutex.
- **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 requested authentication service.
- **rpc_s_unsupported_protect_level**: Requested protection level is not supported and cannot be upgraded to a higher supported level.
- **rpc_s_exception_detected**: Unexpected exception occurs.
- **rpc_s_user_cancelled**: Call was canceled by the user.

**Usage**

The `rpc_binding_set_auth_info` API sets up the specified server binding handle so that it can be used to make authenticated remote procedure calls, including authorization information.

Unless a client calls this routine, all remote procedure calls made on the binding binding handle are unauthenticated. A client is not required to call this routine.

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 protection 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 chosen.

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 for 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 \texttt{rpc\_binding\_inq\_auth\_client()} it needs to check for a status of \texttt{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 \texttt{rpc\_binding\_inq\_auth\_client()}.

Return Values

None.

Related Information

Routines

+ \texttt{rpc\_binding\_inq\_auth\_client}
+ \texttt{rpc\_mgmt\_inq\_server\_princ\_name}
+ \texttt{sec\_login\_newgroups}
+ \texttt{rpc\_binding\_inq\_auth\_info}
+ \texttt{sec\_login\_get\_current\_context}
+ \texttt{sec\_login\_setup\_identity}
+ \texttt{rpc\_mgmt\_inq\_dflt\_protect\_level}
rpc_binding_set_object

Purpose
Sets the object UUID value into a server binding handle.
Used by client applications.

Format

#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.

Possible status codes and their meanings:
  rpc_s_ok
  Successful completion.
  rpc_s_invalid_binding
  Binding handle not valid.
  rpc_s_wrong_kind_of_binding
  Wrong kind of binding for operation.

Usage

The rpc_binding_set_object API 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

None.

Related Information

Routines
rpc_binding_from_string_binding  rpc_binding_inq_object
**rpc_binding_to_string_binding**

**Purpose**
Returns a string representation of a binding handle.

Used by client, server, or management applications.

**Format**

```c
#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.

Possible status codes and their meanings:
- **rpc_s_ok**
  Successful completion.
- **rpc_s_cant_getpeername**
  Cannot get peer name.
- **rpc_s_connection_closed**
  Connection closed.
- **rpc_s_invalid_binding**
  Binding handle not valid.

**Usage**

The `rpc_binding_to_string_binding` API converts a client or server binding handle to its string representation.

The RPC runtime allocates storage for the string returned in the `string_binding` parameter. The application calls the `rpc_string_free` API to deallocate that storage.

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 the `rpc_string_binding_parse` API.

**Return Values**

None.

**Related Information**
Routines

rpc_binding_from_string_binding      rpc_string_binding_parse      rpc_string_free
rpc_binding_vector_free

Frees the storage used to store a vector and binding handles. Used by client or server applications.

Format

```
#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.

  Possible status codes and their meanings:

  - **rpc_s_ok**: Successful completion.
  - **rpc_s_invalid_arg**: Argument not valid.
  - **rpc_s_invalid_binding**: Binding handle not valid.
  - **rpc_s_wrong_kind_of_binding**: Wrong kind of binding for operation.

Usage

The `rpc_binding_vector_free` API frees the storage used to store a vector of server binding handles. The freed storage 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` API 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

None.

Related Information

Routines

- `rpc_binding_free`
- `rpc_ns_binding_lookup_next`
- `rpc_server_inq_bindings`
rpc_cs_binding_set_tags

Purpose
Places code set tags into a server binding handle.
Used by client applications.

Format
#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 possible status codes and their meanings are as follows:
 rpc_s_ok Operation successful.
 rpc_s_no_memory Insufficient storage to perform operation.

The routine can also return status codes generated by the rpc_rgy_get_codesets() routine.

Usage
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
**rpc_cs_binding_set_tags**

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**

None.

**Related Information**

**Routines**

*rpc_cs_eval_with_universal*  
*rpc_cs_eval_without_universal*  
*rpc_cs_get_tags*
rpc_cs_char_set_compat_check

Purpose
Evaluates character set compatibility between a client and a server.

Used by client applications.

Format
#include <dce/rpc.h>

void rpc_cs_char_set_compat_check (  
    rpc_codeset_mgmt_p_t client_code_sets_array,  
    rpc_codeset_mgmt_p_t server_code_sets_array,  
    error_status_t *status);

Parameters
Input
client_rgy_code_set_array  
The array containing all of the information concerning the client machine's supported code sets.  
The first array element contains the information for the code set in which the client is currently running.

server_rgy_code_set_array  
The array containing all of the information concerning the server machine's supported code sets.  
The first array element contains the information for the code set in which the server is currently running.

Output
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
rpc_s_ss_noCompatCharsets

The routine can also return status codes from the dce_cs_rgy_to_loc() routine.

Usage
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 as input a client code sets array and a server code sets array. These arrays contain all of the code set information for the respective application's host machine. The first entry in the array contains the code set information for the code set in which the application (client or server) is currently running. This code set information contains the registered values that represent the character sets that are supported by the particular code set. If the 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_noCompatCharsets.

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.
rpc_cs_char_set_compat_check

Client and server applications that use the DCE RPC code sets evaluation routines \texttt{rpc_cs_eval_with_universal()} and \texttt{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 \texttt{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. The client application should use \texttt{rpc_rgy_get_codesets()} to obtain the array for the client's code sets information and \texttt{rpc_ns_mgmt_read_codesets()} to obtain the array for the server's code sets information.

Permissions Required

No permissions are required.

Return Values

None.

Related Information

Routines

\begin{tabular}{lll}
\texttt{rpc_cs_eval_with_universal} & \texttt{rpc_cs_get_tags} & \texttt{rpc_rgy_get_codesets} \\
\texttt{rpc_cs_eval_without_universal} & \texttt{rpc_ns_mgmt_read_codesets} & \\
\end{tabular}
**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.

**Format**
```
#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`.

**Usage**

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_cs_eval_with_universal()` 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 `rpc_ns_binding_import_next()` routine 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 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. They only need to use the `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.
rpc_cs_eval_with_universal

Return Value
None.

Related Information
Routines

rpc_cs_eval_without_universal  rpc_ns_binding_import_done  rpc_ns_import_ctx_add_eval
rpc_cs_get_tags  rpc_ns_binding_import_next  rpc_ns_mgmt_handle_set_exp_age
rpc_ns_binding_import_begin
**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.

**Format**

```c
#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`.

**Usage**

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 `rpc_ns_binding_import_next()` routine 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. They only need to use the `rpc_ns_import_ctx_add_eval()` to set the routine, for example:

```c
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.
rpc_cs_eval_without_universal

Return Values

None.

Related Information

Routines

rpc_ns_binding_import_begin       rpc_ns_binding_import_next       rpc_ns_mgmt_handle_set_exp_age
rpc_ns_binding_import_done        rpc_ns_import_ctx_add_eval
rpc_cs_get_tags

Purpose
Retrieves code set tags from a binding handle.
Used by client and server applications.

Format
#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. When called from the
server stub, this value is a pointer to the tag information that the client stub passed in the RPC
call.

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_receiving_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.

The possible status codes and their meanings are as follows:

rpc_s_ok
Success.

rpc_s_ss_incompatible_codesets
The server cannot handle the data in the code set that
the client has specified. This status code will be
returned if the application performs code set
compatibility evaluation in the server stub.
The result of the client-side evaluation used an invalid code set tag.

If code set compatibility evaluation is performed, error values can also be returned from `rpc_rgry_get_codesets()`, `rpc_ns_binding_inq_entry_name()`, and `rpc_ns_mgmt_read_codesets()`.

### Usage

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 returned by the `rpc_ns_binding_import_next()` or `rpc_ns_binding_select()` routines. 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 binding handle is a pointer to the tag data sent by the client stub.

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 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.

If you are using PDS (as opposed to HFS), modify the `include` statement in the format to be `#include <dce/codestub.h>`.

### Permissions Required

No permissions are required.
Return Values
None.

Related Information

Routines

<table>
<thead>
<tr>
<th>cs_byte_local_size</th>
<th>cs_byte_from_netcs</th>
<th>cs_byte_to_netcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs_byte_net_size</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**rpc_ep_register**

**Purpose**
Adds to, or replaces, server address information in the local endpoint map.

Used by server applications.

**Format**
```c
#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. The vector must contain fully bound server binding handles.
- **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 element of the added cross product between the binding vector and object UUID vector. The string can be up to 64 characters long, including the `NULL` ending 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 to enumerate 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.
  
  Possible status codes and their meanings:
  - **rpc_s_ok** Successful completion.
  - **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** Database Entry not valid.
  - **ept_s_update_failed** Update failed.
  - **rpc_s_auth_tkt_expired** Login context expired.
  - **rpc_s_binding_has_no_auth** Caller has no authority.
  - **rpc_s_comm_failure** Communications failure.
  - **rpc_s_invalid_binding** Binding handle not valid.
  - **rpc_s_no_bindings** No bindings.
Note to Reader

The `ept_s_not_authorized`, `ept_s_auth_tkt_expired`, and `rpc_s_binding_has_no_auth` status codes are a result of endpoint map protection that is unique to the z/OS DCE implementation. Therefore, you should code your server to respond properly if any of these status codes are returned. (See the `z/OS DCE Administration Guide` for information about controlling access to the DCED endpoint map.)

Usage

The `rpc_ep_register` API adds elements to, or replaces elements in, the local host’s endpoint map.

Each element in the local endpoint map contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information of a binding handle
- 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, only one server instance offers the same interface UUID, object UUID, and protocol sequence.

**Note:** For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the `rpc_s_binding_has_no_auth` status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

In z/OS DCE all endpoint elements registered by a server instance are removed when that server instance stops. If the server instance stops while DCE Host Daemon is down, the obsolete elements are removed on start up. Therefore, there is no accumulation of obsolete elements in the endpoint map.

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 element’s interface major and minor version numbers:

<table>
<thead>
<tr>
<th>Existing Element Relationship</th>
<th>Provided Element Major version number</th>
<th>Routine’s Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not equal to</td>
<td>Major version number</td>
<td>Ignore minor version number relationship and add a new endpoint map element. The existing element remains unchanged.</td>
</tr>
<tr>
<td>Equal to</td>
<td>Major version number</td>
<td>Act according to the minor version number relationship.</td>
</tr>
<tr>
<td>Equal to</td>
<td>Minor version number</td>
<td>Replaces the endpoint of the existing element based on the provided information.</td>
</tr>
<tr>
<td>Less than</td>
<td>Minor version number</td>
<td>Replaces the existing element based on the provided information.</td>
</tr>
<tr>
<td>Greater than</td>
<td>Minor version number</td>
<td>Ignore the provided information. The existing element remains unchanged.</td>
</tr>
</tbody>
</table>

For example, suppose 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, but does not change the element with
rpc_ep_register

version number 1.3. 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 that uses **rpc_server_use_all_protseqs**, **rpc_server_use_protseqs_ep** or **rpc_server_use_protseq** has dynamically allocated endpoints. These servers must call **rpc_ep_register** or **rpc_ep_register_no_replace** to register itself with the local endpoint map.

A server that calls only the **rpc_server_use_all_protseqs_if** or **rpc_server_use_protseq_if** APIs does not need to call this routine. The client’s runtime uses an endpoint from the client’s interface specification to fill in a partially bound binding handle. However, you should 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 that the server uses when calling the **rpc_ns_binding_export** API.

When the server is stopped, its interfaces are unregistered from the endpoint map file.

The **rpc_ep_register** API communicates with the DCE Host Daemon that, in turn, updates 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 (the DCE Host Daemon) selects an element for an interface ID and an object UUID, see the RPC information in [Z/OS 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 specifies a non-nil object UUID that is not in the endpoint map, or the client specifies 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** APIs. The **object_uuid_vec** parameter can contain both nil and non-nil 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 “String Binding” on page 2-17 for an explanation.

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 specify **NULL** to argument **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.

\[
\begin{align*}
& (\text{ifhand}, b1, u1) \quad (\text{ifhand}, b1, u2) \quad (\text{ifhand}, b1, u3) \quad (\text{ifhand}, b1, u4) \\
& (\text{ifhand}, b2, u1) \quad (\text{ifhand}, b2, u2) \quad (\text{ifhand}, b2, u3) \quad (\text{ifhand}, b2, u4) \\
& (\text{ifhand}, b3, u1) \quad (\text{ifhand}, b3, u2) \quad (\text{ifhand}, b3, u3) \quad (\text{ifhand}, b3, u4)
\end{align*}
\]

(An annotation string is part of each of these 12 elements.)

**Return Values**

None.

**Related Information**

**Routines**

- **rpc_ep_register_no_replace**
- **rpc_ep_resolve_binding**
- **rpc_ep_unregister**
- **rpc_mgmt_ep_unregister**
- **rpc_ns_binding_export**
- **rpc_server_inq_bindings**
- **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**

2-100 Application Development Reference Volumes 1 and 2
rpc_ep_register_wlb

Purpose

Adds to, or replaces, server address information in the local endpoint map, for server interfaces that support load balancing.

Used by server applications.

Format

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

void rpc_ep_register_wlb(
    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. The vector must contain fully bound server binding handles.
- `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 element of the added cross product between the binding vector and object UUID vector. The string can be up to 64 characters long, including the `NULL` ending 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 to enumerate 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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `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` Database Entry not valid.
- `ept_s_update_failed` Update failed.
- ` rpc_s_auth_tkt_expired` Login context expired.
- `rpc_s_binding_has_no_auth` Caller has no authority.
- `rpc_s_comm_failure` Communications failure.
- `rpc_s_invalid_binding` Binding handle not valid.
- `rpc_s_no_aename_set` No AENAME was set.
**rpc_ep_register_wlb**

**rpc_s_noBindings**  
No bindings.

**ept_s_not_authorized**  
Caller not authorized.

**rpc_s_wrong_kind_of_binding**  
Wrong kind of binding for operation.

**rpc_s_no_interfaces**  
No interfaces registered.

---

**Note to Reader**

The `ept_s_not_authorized`, `ept_s_auth_tkt_expired`, and `rpc_s_binding_has_no_auth` status codes are a result of endpoint map protection that is unique to the z/OS DCE implementation. Therefore, you should code your server to respond properly if any of these status codes are returned. (See the [z/OS DCE Administration Guide](#) for information about controlling access to the DCED endpoint map.)

The `rpc_s_no_aename_set` status code may be returned if `rpc_set_ae_name` has not been issued and if the `_EUV_LOAD_BALANCE` environmental variable was not set.

---

**Usage**

Use the `rpc_ep_register_wlb` API instead of `rpc_ep_register` when developing a new DCE server application which exploits DCE Workload Management. The `rpc_ep_register_wlb` API does everything the `rpc_ep_register` API does, plus it sets the activated bit for the endpoint. It also associates the AENAME in the endpoint map with the AENAME specified in the `_EUV_LOAD_BALANCE` environment variable or the `rpc_set_ae_name` API call.

Workload balancing allows DCE clients to balance their RPC workloads across a group of servers while providing the same RPC interfaces. For detailed information about DCE Workload Management see the [z/OS DCE Application Development Guide: Core Components](#) SC24-5905.

Each element in the local endpoint map contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information of a binding handle
- Object UUID (optional)
- Annotation (optional)
- Activate
- Reserved

The endpoint is registered with the endpoint map. Here is where `rpc_ep_register_wlb` differs from `rpc_ep_register`. The activate and the reserved bits are architected for the workload balancing APIs. These two elements are used to record the workload balancing information. Specifically, the **activated flag is set**.

A server uses this routine, instead of `rpc_ep_register_no_replace_wlb`, when only a single instance of the server runs on the server's host. Use this routine if, at any time, only one server instance offers the same interface UUID, object UUID, and protocol sequence.

**Note:** For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the `rpc_s_binding_has_no_auth` status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

In z/OS DCE all endpoint elements registered by a server instance are removed when that server instance stops. If the server instance stops while DCE Host Daemon is down, the obsolete elements are removed on start up. Therefore, there is no accumulation of obsolete elements in the endpoint map.

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 element's interface major and minor version numbers:
Table 2-9. How rpc_ep_register_wlb affects the EPM

<table>
<thead>
<tr>
<th>Existing Element</th>
<th>Relationship</th>
<th>Provided Element</th>
<th>Routine's Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major version number</td>
<td>Not equal to Major version number</td>
<td>Ignore minor version number relationship and add a new endpoint map element. The existing element remains unchanged.</td>
<td></td>
</tr>
<tr>
<td>Major version number</td>
<td>Equal to Major version number</td>
<td>Act according to the minor version number relationship.</td>
<td></td>
</tr>
<tr>
<td>Minor version number</td>
<td>Equal to Minor version number</td>
<td>Replaces the endpoint of the existing element based on the provided information.</td>
<td></td>
</tr>
<tr>
<td>Minor version number</td>
<td>Less than Minor version number</td>
<td>Replaces the existing element based on the provided information.</td>
<td></td>
</tr>
<tr>
<td>Minor version number</td>
<td>Greater than Minor version number</td>
<td>Ignore the provided information. The existing element remains unchanged.</td>
<td></td>
</tr>
</tbody>
</table>

For example, suppose 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, but does not change the element with version number 1.3. 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 that uses rpc_server_use_all_protseqs, rpc_server_use_protseqs_ep or rpc_server_use_protseq has dynamically allocated endpoints. These servers must call rpc_ep_register_wlb or rpc_ep_register_no_replace_wlb to register itself with the local endpoint map.

A server that calls only the rpc_server_use_all_protseqs_if or rpc_server_use_protseq_if APIs does not need to call this routine. The client's runtime uses an endpoint from the client's interface specification to fill in a partially bound binding handle. However, you should 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 that the server uses when calling the rpc_ns_binding_export API.

When the server is stopped, its interfaces are unregistered from the endpoint map file.

The rpc_ep_register_wlb API communicates with the DCE Host Daemon that, in turn, updates 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_wlb. The routine indicates this failure by placing the value rpc_s_comm_failure into status.

For information about how the endpoint map service (the DCE Host Daemon) selects an element for an interface ID and an object UUID, see the RPC information in z/OS 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 specifies a non-nil object UUID that is not in the endpoint map, or the client specifies a nil object UUID, the search can succeed — but only if the server has registered a nil object UUID using the rpc_ep_register_wlb or rpc_ep_register_no_replace_wlb APIs. The object_uuid_vec parameter can contain both nil and non-nil 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 "String Binding" on page 2-17 for an explanation. Note, however, that in this case no load balancing can occur.

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 specify NULL to argument 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)
```
**rpc_ep_register_wlb**

(An annotation string is part of each of these 12 elements.)

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_ep_register`
- `rpc_ep_register_no_replace`
- `rpc_ep_register_no_replace_wlb`
- `rpc_ep_resolve_binding`
- `rpc_ep_unregister`
- `rpc_mgmt_ep_set_activated_wlb`
- `rpc_mgmt_ep_unregister`
- `rpc_ns_binding_export`
- `rpc_server_inq_bindings`
- `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`
rpc_ep_register_no_replace

Purpose

Adds to server address information in the local endpoint map.

Used by server applications.

Format

```c
#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. Note that the vector must contain fully bound server binding handles.
- `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 element of the added cross product between the binding vector and object UUID vector. The string can be up to 64 characters long, including the `NULL` ending 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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `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` Database Entry not valid.
- `ept_s_update_failed` Update failed.
- `rpc_s_auth_tkt_expired` Login context expired.
- `rpc_s_binding_has_no_auth` Caller has no authority.
- `rpc_s_comm_failure` Communications failure.
- `rpc_s_invalid_binding` Binding handle not valid.
- `rpc_s_no_bindings` No bindings.
The ept_s_not_authorized, ept_s_auth_tkt_expired, and rpc_s_binding_has_no_auth status codes are a result of endpoint map protection that is unique to the z/OS DCE implementation. Therefore, you should code your server to respond properly if this status code is returned. (See the z/OS DCE Administration Guide for information about controlling access to the DCED endpoint map.)

Usage

The rpc_ep_register_no_replace API 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 contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information of a binding handle
- 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 more than one server instance offers the same interface UUID, object UUID, and protocol sequence.

In z/OS DCE all endpoint elements registered by a server instance are removed when that server instance stops. If the server instance stops while DCE Host Daemon is down, the obsolete elements are removed on start up. Therefore, there is no accumulation of obsolete elements in the endpoint map.

Note: For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the rpc_s_binding_has_no_auth status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

A server program that uses rpc_server_use_all_protseqs, rpc_server_use_protseq_ep or rpc_server_use_protseq has dynamically allocated endpoints. These servers must call rpc_ep_register or rpc_ep_register_no_replace to register itself with the local endpoint map.

A server that calls only the rpc_server_use_all_protseqs_if or rpc_server_use_protseq_if APIs 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. 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 that the server uses when calling the rpc_ns_binding_export API.

The rpc_ep_register_no_replace API communicates with the DCE Host Daemon that, 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 z/OS 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 specifies a non-nil object UUID that is not in the endpoint map, or the client specifies 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...
The `rpc_ep_register_no_replace` API. The `object_uuid_vec` parameter can contain both nil and non-nil 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 "String Binding" on page 2-17 for an explanation.

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 specify `NULL` to `object_uuid_vec`, the corresponding elements in the cross product contain a nil object UUID. The description of the `rpc_ep_register` API summarizes the contents of an element in the local endpoint map.

Return Values

None.

Related Information

Routines

<table>
<thead>
<tr>
<th>rpc_ep_register</th>
<th>rpc_mgmt_ep_unregister</th>
<th>rpc_server_use_all_protseqs_if</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_ep_register_wlb</td>
<td>rpc_ns_binding_export</td>
<td>rpc_server_use_protseq</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace_wlb</td>
<td>rpc_server_inq_bindings</td>
<td>rpc_server_use_protseq_ep</td>
</tr>
<tr>
<td>rpc_ep_resolve_binding</td>
<td>rpc_server_use_all_protseqs</td>
<td>rpc_server_use_protseq_if</td>
</tr>
<tr>
<td>rpc_ep_unregister</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Purpose**

Adds to server address information in the local endpoint map.

Used by server applications.

**Format**

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

void rpc_ep_register_no_replace_wlb(
    rpc_if_handle_t if_handle,
    rpc_binding_vector_t *binding_vec,
    uuid_vector_t *object_uuid_vec,
    unsigned char_t *annotation,
    unsigned32_t *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. Note that the vector must contain fully bound server binding handles.

- **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 element of the added cross product between the binding vector and object UUID vector. The string can be up to 64 characters long, including the `NULL` ending 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.
  Possible status codes and their meanings:

  - **rpc_s_ok**
    Successful completion.
  - **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**
    Database Entry not valid.
  - **ept_s_update_failed**
    Update failed.
  - **rpc_s_auth_tkt_expired**
    Login context expired.
  - **rpc_s_binding_has_no_auth**
    Caller has no authority.
  - **rpc_s_comm_failure**
    Communications failure.
  - **rpc_s_invalid_binding**
    Binding handle not valid.
  - **rpc_s_no_aename_set**
    No AENAME was set.
Note to Reader

The `ept_s_not_authorized`, `ept_s_auth_tkt_expired`, and `rpc_s_binding_has_no_auth` status codes are a result of endpoint map protection that is unique to the z/OS DCE implementation. Therefore, you should code your server to respond properly if this status code is returned. (See the z/OS DCE Administration Guide for information about controlling access to the DCED endpoint map.)

The `rpc_s_no_aename_set` status code may be returned if `rpc_set_ae_name` has not been issued and if the `_EUV_LOAD_BALANCE` environmental variable was not set.

Usage

The `rpc_ep_register_no_replace_wlb` API 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_wlb`. Like `rpc_ep_register_wlb`, this API sets the activated bit for the endpoint and associates an AENAME from the `_EUV_LOAD_BALANCE` environment variable or the `rpc_set_ae_name` API call.

Workload balancing allows DCE clients to balance their RPC workloads across a group of servers while providing the same RPC interfaces. For detailed information about DCE Workload Management see the z/OS DCE Application Development Guide: Core Components, SC24-5905.

Each element in the local endpoint map contains the following:

- Interface ID, consisting of an interface UUID and versions (major and minor)
- Binding information of a binding handle
- Object UUID (optional)
- Annotation (optional)
- Activate
- Reserved

A server uses this routine, instead of `rpc_ep_register_wlb`, when multiple instances of the server run on the same host. Use this routine if more than one server instance offers the same interface UUID, object UUID, and protocol sequence.

In z/OS DCE all endpoint elements registered by a server instance are removed when that server instance stops. If the server instance stops while DCE Host Daemon is down, the obsolete elements are removed on start up. Therefore, there is no accumulation of obsolete elements in the endpoint map.

Note: For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the `rpc_s_binding_has_no_auth` status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

A server program that uses `rpc_server_use_all_protseqs`, `rpc_server_use_protseq_ep` or `rpc_server_use_protseq` has dynamically allocated endpoints. These servers must call `rpc_ep_register` or `rpc_ep_register_no_replace` to register itself with the local endpoint map.

A server that calls only the `rpc_server_use_all_protseqs_if` or `rpc_server_use_protseq_if` APIs 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. 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 that the server uses when calling the `rpc_ns_binding_export` API.
The `rpc_ep_register_no_replace_wlb` API communicates with the DCE Host Daemon that, 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 [z/OS DCE Application Development Guide: Core Components](https://www.ibm.com/docs/zos). This guide explains how the endpoint map service searches for the endpoint of a server that is compatible with a client.

If the client specifies a non-nil object UUID that is not in the endpoint map, or the client specifies 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_wlb` or `rpc_ep_register_wlb` API. The `object_uuid_vec` parameter can contain both nil and non-nil 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 "String Binding" on page 2-17 for an explanation.

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 specify `NULL` to `object_uuid_vec`, the corresponding elements in the cross product contain a nil object UUID. The description of the `rpc_ep_register_wlb` API summarizes the contents of an element in the local endpoint map.

### Return Values

None.

### Related Information

#### Routines

- `rpc_ep_register`
- `rpc_ep_register_wlb`
- `rpc_ep_no_replace`
- `rpc_ep_resolve_binding`
- `rpc_ep_unregister`
- `rpc_mgmt_ep_unregister`
- `rpc_mgmt_ep_set_activated_wlb`
- `rpc_ns_binding_export`
- `rpc_server_inq_bindings`
- `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`
rpc_ep_resolve_binding

Purpose
Resolves a partially bound server binding handle into a fully bound server binding handle.
Used by client or management applications.

Format
```
#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.

**Input**

- **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.

Possible status codes and their meanings:
- **rpc_s_ok**
  Successful completion.
- **ept_s_not_registered**
  No entries found.
- **rpc_s_invalid_binding**
  Binding handle not valid.
- **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.

Usage

An application calls the **rpc_ep_resolve_binding** API 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, on the host that the **binding** parameter specifies, to look up an endpoint for a compatible server instance. The DCE Host Daemon 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** API 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 **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: 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 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. Because the application can then use this fully bound handle to make remote management calls, calling the rpc_ep_resolve_binding API is unnecessary.

When a high proportion of all servers in an environment offers the same interface, the interface is known as a pervasive one.

For each server instance, the RPC runtime provides routines (the rpc_mgmt_* APIs) that form an RPC interface. This management interface is registered with the runtime. A server typically does not register the management interface with the local endpoint map. Even if it does, it would also be a pervasive interface.

To unambiguously locate a compatible server instance, an application needs to call rpc_ep_resolve_binding before calling a routine in the management or a pervasive interface.

Partially Bound Handles with a Non-Nil Object UUID: If the application has a partially bound handle with a non-nil object UUID, the application can decide not to call rpc_ep_resolve_binding 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 rpc_ep_resolve_binding.

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 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 return 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 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: 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 rpc_mgmt_ep_elt_inq_begin, rpc_mgmt_ep_elt_inq_next, and rpc_mgmt_ep_elt_inq_done.
Return Values

None.

Related Information

Routines

- `rpc_ep_register`
- `rpc_ep_register_no_replace`
- `rpc_mgmt_ep_elt_inq_begin`
- `rpc_mgmt_ep_elt_inq_done`
- `rpc_mgmt_ep_elt_inq_next`
- `rpc_binding_from_string_binding`
- `rpc_binding_reset`
Purpose
Removes server address information from the local endpoint map.
Used by server applications.

Format
#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.

Possible status codes and their meanings:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>ept_s_cant_access</td>
<td>Error reading endpoint database.</td>
</tr>
<tr>
<td>ept_s_cant_create</td>
<td>Error creating endpoint database.</td>
</tr>
<tr>
<td>ept_s_cant_perform_op</td>
<td>Cannot perform requested operation.</td>
</tr>
<tr>
<td>ept_s_database_invalid</td>
<td>Endpoint map database invalid.</td>
</tr>
<tr>
<td>ept_s_invalid_entry</td>
<td>Database Entry not valid.</td>
</tr>
<tr>
<td>ept_s_update_failed</td>
<td>Update failed.</td>
</tr>
<tr>
<td>ept_s_not_registered</td>
<td>Entry is not registered in endpoint map.</td>
</tr>
<tr>
<td>rpc_s_auth_tkt_expired</td>
<td>Login context expired.</td>
</tr>
<tr>
<td>rpc_s_binding_has_no_auth</td>
<td>Caller has no authority.</td>
</tr>
<tr>
<td>rpc_s_invalid_binding</td>
<td>Binding handle not valid.</td>
</tr>
<tr>
<td>rpc_s_no_bindings</td>
<td>No bindings.</td>
</tr>
<tr>
<td>ept_s_not_authorised</td>
<td>Caller has no authority.</td>
</tr>
<tr>
<td>rpc_s_wrong_kind_of_binding</td>
<td>Wrong kind of binding for operation.</td>
</tr>
</tbody>
</table>
Note to Reader

The ept_s_not_authorized, ept_s_auth_tkt_expired, and rpc_s_binding_has_no_auth status codes are a result of endpoint map protection that is unique to the z/OS DCE implementation. Therefore, you should code your server to respond properly if any of these status codes are returned. (See the z/OS DCE Administration Guide for information about controlling access to the DCED endpoint map.)

Usage

The rpc_ep_unregister API 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 wants to remove that address information from the local endpoint map.

A server program can remove its own local endpoint map elements (server address information) based on either of the following:

- 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 argument binding_vec before calling this routine. See "rpc_binding_vector_free" on page 2-86 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 description of the rpc_ep_register API 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.

Note: For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the rpc_s_binding_has_no_auth status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

Return Values

None.

Related Information

Routines

 rpc_ep_register  rpc_mgmt_ep_unregister  rpc_server_inq_bindings
 rpc_ep_register_no_replace  rpc_ns Binding_unregister

rpc_if_id_vector_free

Purpose
Frees a vector and the storage used to store interface identifier structures.

Used by client, server, or management applications.

Format
#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.

Possible status codes and their explanations:
    rpc_s_ok Successful completion.
    rpc_s_invalid_arg Argument not valid.

Usage
The rpc_if_id_vector_free API frees the storage used to store a vector of interface identifiers. This includes storage 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
None.

Related Information
Routines
rpc_if_inq_id rpc_mgmt_inq_if_ids rpc_ns_mgmt_entry_inq_if_ids
rpc_if_inq_id

Purpose
Returns the interface identifier for an interface specification.
Used by client or server applications.

Format
#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 storage 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.

Possible status code and its meaning:
rpc_s_ok    Successful completion.

Usage
The rpc_if_inq_id API obtains 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
None.

Related Information
Routines
rpc_if_id_vector_free  rpc_mgmt_inq_if_ids  rpc_ns_mgmt_entry_inq_if_ids
rpc_mgmt_ep_elt_inq_begin

**Purpose**

Creates an inquiry context for viewing the elements in a local or remote endpoint map.

Used by management applications.

**Format**

```c
#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 non-nil 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 lists the valid inquiry types:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_ep_all_elts</td>
<td>Returns every element from the local endpoint map. The <code>if_id</code>, <code>vers_option</code>, and <code>object_uuid</code> parameters are ignored.</td>
</tr>
<tr>
<td>rpc_c_ep_match_by_if</td>
<td>Searches the local endpoint map for those elements that contain the interface identifier specified by the <code>if_id</code> and <code>vers_option</code> values. The <code>object_uuid</code> parameter is ignored.</td>
</tr>
<tr>
<td>rpc_c_ep_match_by_obj</td>
<td>Searches the local endpoint map for those elements that contain the object UUID specified by the <code>object_uuid</code> parameter. The <code>if_id</code> and <code>vers_option</code> parameters are ignored.</td>
</tr>
<tr>
<td>rpc_c_ep_match_by_both</td>
<td>Searches the local endpoint map for those elements that contain the interface identifier and object UUID specified by the <code>if_id</code>, <code>vers_option</code>, and <code>object_uuid</code> parameters.</td>
</tr>
</tbody>
</table>

*if_id*

Specifies the interface identifier of the local endpoint map elements to be returned by the `rpc_mgmt_ep_elt_inq_next` API.

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 you can specify `NULL` to get a nil UUID.
**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 you can specify a 0 (zero) value.

The following lists the valid values for this parameter:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_vers_all</td>
<td>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 <code>if_id</code>.</td>
</tr>
<tr>
<td>rpc_c_vers_compatible</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_exact</td>
<td>Returns local endpoint map elements that offer the specified version of the specified interface UUID.</td>
</tr>
<tr>
<td>rpc_c_vers_major_only</td>
<td>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 <code>if_id</code>.</td>
</tr>
<tr>
<td>rpc_c_vers_upto</td>
<td>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 <code>if_id</code> contains V2.0 and the local endpoint map contained elements with the following versions: V1.3, V2.0, and V2.1. The <code>rpc_mgmt_ep_elt_inq_next</code> API returns the elements with V1.3 and V2.0.)</td>
</tr>
</tbody>
</table>

**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 specify `NULL` to get 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` APIs.

**status**

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

Possible status codes and their meanings:

- `rpc_s_ok`  
  Successful completion.
- `rpc_s_invalid_inquiry_context`  
  Inquiry context not valid.
- `rpc_s_invalid_inquiry_type`  
  Inquiry type not valid.
- `rpc_s_invalid_vers_option`  
  Version option not valid.
- `rpc_s_wrong_kind_of_binding`  
  Wrong kind of binding for operation.
- `rpc_s_no_memory`  
  No storage available.
**Usage**

The `rpc_mgmt_ep_elt_inq_begin` API 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` API:

- 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` API to delete the inquiry context.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_ep_register`
- `rpc_ep_unregister`
- `rpc管理模式_ep_elt_inq_begin`
- `rpc管理模式_ep_elt_inq_next`
- `rpc管理模式_ep_unregister`
- `rpc管理模式_ep_register_no_replace`
- `rpc管理模式_ep_elt_inq_done`
**Purpose**
Deletes the inquiry context for viewing the elements in a local or remote endpoint map.
Used by management applications.

**Format**
```
#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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_invalid_inquiry_context` Inquiry context not valid.
- `rpc_s_comm_failure` RPC runtime is not communicating with the remote server.

**Usage**
The `rpc_mgmt_ep_elt_inq_done` API deletes an inquiry context. The `rpc_mgmt_ep_elt_inq_begin` API 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**
None.

**Related Information**

**Routines**
`rpc_mgmt_ep_elt_inq_begin` `rpc_mgmt_ep_elt_inq_next`
Purpose
Returns one element from a local or remote endpoint map.
Used by management applications.

Format
#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 API.

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
do not call the rpc_binding_free API.
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 parameter. In this case, the application
do not call the rpc_string_free API.
status         Returns the status code from this routine. This status code indicates whether the routine
completed successfully or, if not, why not.

Possible status codes and their meanings:
rpc_s_ok                  Successful completion.
ept_s_cant_perform_op     Cannot perform the requested operation.
rpc_s_comm_failure        Communications failure.
ept_s_database_invalid    Endpoint map database not valid.
rpc_s_fault_context_mismatch  Fault context mismatch.
ept_s_invalid_context    Inquiry type for this context not valid.
ept_s_invalid_entry      Database Entry not valid.
rpc_s_invalid_arg        Argument not valid.
rpc_s_invalid_inquiry_context Inquiry context not valid.
rpc_s_invalid_inquiry_type Inquiry type not valid.
rpc_s_no_more_elements  No more elements.
Usage

The `rpc_mgmt_ep_elt_inq_next` API 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 description of the `rpc_ep_register` API 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` API. 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 if 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 storage 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

None.

Related Information

Routines

<table>
<thead>
<tr>
<th>routine</th>
<th>routine</th>
<th>routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_binding_free</td>
<td>rpc_ep_register_no_replace</td>
<td>rpc_mgmt_ep_elt_done</td>
</tr>
<tr>
<td>rpc_ep_register</td>
<td>rpc_mgmt_ep_elt_begin</td>
<td>rpc_string_free</td>
</tr>
</tbody>
</table>
### Purpose
Returns one element from a local or remote endpoint map.

Used by management applications.

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

void rpc_mgmt_ep_elt_inq_next_wlb(
    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,
    boolean *activated,
    boolean *reserved,
    char **aename,
    unsigned32 *status);
```

### Parameters
**Input**
- `inquiry_context`: Specifies an inquiry context. This inquiry context is returned from the `rpc_mgmt_ep_elt_inq_begin` API.

**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` API.
- `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 parameter. In this case, the application does not call the `rpc_string_free` API.
- `activated`: Returns the state, true or false, of the activated variable for this endpoint. Specify `NULL` to prevent the routine from returning this parameter.
- `reserved`: Returns the variable if the state of the endpoint map is not null. Specify `NULL` to prevent the routine from returning this parameter.
- `aename`: Returns the AENAME associated with this endpoint. Specify `NULL` to prevent the routine from returning this parameter. In this case, the application does not need to call the `rpc_string_free` API.
- `status`: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings:
- `rpc_s_ok`: Successful completion.
- `ept_s_cant_perform_op`: Cannot perform the requested operation.
The `rpc_mgmt_ep_elt_inq_next_wlb` API, like the `rpc_mgmt_ep_elt_inq_next` API, 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 components of a selected local endpoint map element. For `rpc_mgmt_ep_elt_inq_next_wlb`, these components include three additional fields, `activated`, `reserved`, `aename`. The `activated` and `reserved` fields are boolean. The `aename` field is a null terminated string. The description of the `rpc_ep_register_wlb` API 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_wlb` API. 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 if 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 storage for the returned `binding` the `annotation` string, and the `aename` 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` and `aename` 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**

None.

**Related Information**

**Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_binding_free</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_ep_register</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_ep_register_wlb</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_ep_register_no_replace</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_ep_register_no_replace_wlb</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_mgmt_ep_elt_begin</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_mgmt_ep_elt_done</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_mgmt_ep_elt_inq_next_wlb</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_mgmt_ep_elt_inq_next</code></td>
<td></td>
</tr>
<tr>
<td><code>rpc_s_comm_failure</code></td>
<td>Communications failure.</td>
</tr>
<tr>
<td><code>ept_s_database_invalid</code></td>
<td>Endpoint map database not valid.</td>
</tr>
<tr>
<td><code>rpc_s_fault_context_mismatch</code></td>
<td>Fault context mismatch.</td>
</tr>
<tr>
<td><code>ept_s_invalid_context</code></td>
<td>Inquiry type for this context not valid.</td>
</tr>
<tr>
<td><code>ept_s_invalid_entry</code></td>
<td>Database Entry not valid.</td>
</tr>
<tr>
<td><code>rpc_s_invalid_arg</code></td>
<td>Argument not valid.</td>
</tr>
<tr>
<td><code>ept_s_invalid_inquiry_context</code></td>
<td>Inquiry context not valid.</td>
</tr>
<tr>
<td><code>rpc_s_invalid_inquiry_type</code></td>
<td>Inquiry type not valid.</td>
</tr>
<tr>
<td><code>rpc_s_no_more_elements</code></td>
<td>No more elements.</td>
</tr>
</tbody>
</table>
rpc_mgmt_ep_set_activated_wlb

Purpose
Sets or resets the value of the activated bit for the designated endpoint. Another boolean input parameter is reserved for future use and otherwise has no effect. Used by management applications.

Format
#include <dce/rpc.h>

void rpc_mgmt_ep_set_activated_wlb(
    rpc_binding_handle_t ep_binding,
    rpc_if_id_t *if_id,
    rpc_binding_handle_t binding,
    uuid_t *object_uuid,
    boolean activated,
    boolean reserved,
    unsigned32 *status );

Parameters

Input

ep_binding Specifies the host whose local endpoint map elements for which you wish to set the state of the activated flag. To modify elements from the same host as the calling application, specify NULL. To modify 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 non-nil 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 modify in the local endpoint map.

binding Specifies the binding handle to modify.

object_uuid Specifies an optional object UUID to modify. The value NULL indicates there is no object UUID to consider in selecting the endpoint.

activated Specifies the new state of the activated variable for this changed endpoint map entry.

reserved Has no effect. May be used in future releases.

Output

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

Possible status codes and their meanings:

rpc_s_ok Successful completion.
ept_s_cant_access Cannot read 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 not valid.
ept_s_invalid_entry Database Entry not valid.
ept_s_not_registered No entries found.
ept_s_update_failed Update failed.
rpc_s_auth_tkt_expired Login context expired.
rpc_s_binding_has_no_auth Caller has no authority.
Usage

The **rpc_mgmt_ep_set_activated_wlb** API is used to set or reset the state of the activated bit for the target endpoint. The target endpoint is selected by the *if_id*, the *binding*, and, if applicable, the *object_uuid*.

All endpoints associated with load balanced interfaces on the same host should have the activated variable in the same state to work the way you expect. If at least one such endpoint has activated in the true state, requests on that host will be load balanced.

Care must be used in setting the activated bit for a given *if_id*. Only those interfaces supporting load balancing should be activated. Activating interfaces for which the server does not explicitly support load balancing could result in errors.

**Note**: For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the **rpc_s_binding_has_no_auth** status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

Return Values

None.

Related Information

**Routines**

- **rpc_ep_register**
- **rpc_ep_register_wlb**
- **rpc_ep_register_no_replace**
- **rpc_ep_register_no_replace_wlb**
- **rpc_ns_binding_unexport**
- **rpc_mgmt_ep_elt_inq_done**
- **rpc_mgmt_ep_elt_inq_begin**
- **rpc_mgmt_ep_elt_inq_next_wlb**

Note to Reader

If the AENAME field in the target endpoint entry is null, status code **ept_s_cant_perform_op**, is returned.
**Purpose**

Removes server address information from a local or remote endpoint map.

Used by management applications.

**Format**

```c
#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 non-nil 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.

  Possible status codes and their meanings:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>ept_s_cant_access</td>
<td>Cannot read the endpoint database.</td>
</tr>
<tr>
<td>ept_s_cant_perform_op</td>
<td>Cannot perform the requested operation.</td>
</tr>
<tr>
<td>rpc_s_comm_failure</td>
<td>Communications failure.</td>
</tr>
<tr>
<td>ept_s_database_invalid</td>
<td>Endpoint map database not valid.</td>
</tr>
<tr>
<td>ept_s_invalid_entry</td>
<td>Database Entry not valid.</td>
</tr>
<tr>
<td>ept_s_not_registered</td>
<td>No entries found.</td>
</tr>
<tr>
<td>ept_s_update_failed</td>
<td>Update failed.</td>
</tr>
<tr>
<td>rpc_s_auth_tkt_expired</td>
<td>Login context expired.</td>
</tr>
<tr>
<td>rpc_s_binding_has_no_auth</td>
<td>Caller has no authority.</td>
</tr>
<tr>
<td>rpc_s_invalid_binding</td>
<td>Binding handle not valid.</td>
</tr>
<tr>
<td>rpc_s_no_interfaces</td>
<td>No interfaces registered.</td>
</tr>
<tr>
<td>ept_s_not_authorized</td>
<td>Caller not authorized.</td>
</tr>
</tbody>
</table>
Usage

The `rpc_mgmt_ep_unregister` API unregisters (that is, removes) an element from a local or remote 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_elt_inq_next` routine to view local endpoint map elements. The application can then remove the elements using the `rpc_mgmt_ep_unregister` API.

Note: For z/OS DCE, the endpoint map is protected by an Access Control List (ACL). Depending on how your machine is configured, the ACL may allow any user to access the endpoint map. If this is not the case, the DCE principal running the application that makes this call requires special permissions to use this API. If the principal running the application does not have sufficient permissions, the `rpc_s_binding_has_no_auth` status code is returned. Note that the ACL can be modified at any time. See "Permissions Required" on page 2-19 for more information.

Return Values

None.

Related Information

Routines

- `rpc_ep_register`
- `rpc_ep_register_no_replace`
- `rpc_mgmt_ep_elt_inq_done`
- `rpc_mgmt_ep_elt_inq_next`
- `rpc_mgmt_ep_elt_inq_begin`
- `rpc_ns_binding_unexport`
rpc_mgmt_inq_com_timeout

Purpose
Returns the communication timeout value in a binding handle.

Used by client applications.

Format
#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 communication timeout value from the binding parameter. For a list of binding timeout tables, see [Table 2-13 on page 2-147]
status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings:
- rpc_s_ok Successful completion.
- rpc_s_invalid_binding Binding handle not valid.
- rpc_s_wrong_kind_of_binding Wrong kind of binding for operation.

Usage
The rpc_mgmt_inq_com_timeout API returns the communication 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 reference information explains the timeout values returned in timeout.

To change the timeout value, a client calls rpc_mgmt_set_com_timeout.

Return Values
None.

Related Information
Routines

rpc_mgmt_set_com_timeout
Purpose
Returns the default protection level for an authentication service.
Used by client or server applications.

Format
#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 that returns the default protection level.
The supported authentication services are:
- 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:
- rpc_c_protect_level_default: Use the default protection level for the specified authentication service.
- rpc_c_protect_level_none: Perform no protection.
- rpc_c_protect_level_connect: Perform protection only when the client establishes a relationship with the server.
- rpc_c_protect_level_call: Perform protection only at the beginning of each remote procedure call when the server receives the request.
- rpc_c_protect_level_pkt: Ensure 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 changed.
- rpc_c_protect_level_cdmf_priv: Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than rpc_c_protect_level_pkt_privacy. This is the second highest protection level, but it will be available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.
**Usage**

The `rpc_mgmt_inq_dflt_protect_level` API 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 run.

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**

None.

**Related Information**

Routines

- `rpc_binding_inq_auth_client`
- `rpc_binding_set_auth_info`
rpc_mgmt_inq_if_ids

Purpose
Returns a vector of interface identifiers of interfaces a server offers.
Used by client, server, or management applications.

Format
#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 specify 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. The RPC management interface is registered with the runtime (by the run time) for all RPC servers.

You can avoid this situation by calling the rpc_ep_resolve_binding routine to obtain a fully bound server binding handle.

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.

Possible status codes and their meanings:

- rpc_s_ok: Successful completion.
- rpc_s_binding_incomplete: Binding incomplete (no object ID and no endpoint).
- rpc_s_comm_failure: Communications failure.
- rpc_s_invalid_arg: Argument not valid.
- rpc_s_invalid_binding: Binding handle not valid.
- rpc_s_no_interfaces: No interfaces registered.
- rpc_s_mgmt_op_disallowed: Management operation disallowed.
- rpc_s_wrong_kind_of_binding: Wrong kind of binding for operation.

Usage

The rpc_mgmt_inq_if_ids API obtains 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 storage 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 specifies an authorization function using the `rpc_mgmt_set_authorization_fn` API.

### Return Values

None.

### Related Information

#### Routines

- `rpc_ep_resolve_binding`
- `rpc_mgmt_set_authorization_fn`
- `rpc_server_register_if`
- `rpc_if_id_vector_free`
rpc_mgmt_inq_server_princ_name

Purpose
Returns a server’s principal name.
Used by client, server, or management applications.

Format
#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, specify a server binding handle for that server. For a server application to receive its own principal name of itself, specify the value NULL.
If the binding handle you specify 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. The DCE Host Daemon does not know which server instance to select from the local endpoint map. The RPC management interface is registered with the runtime (by the runtime) 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 authentication services are described in "rpc_binding_set_auth_info" on page 2-79 under the description for auth_info. All listed authentication services are supported for the authn_svc parameter except rpc_c_auth_none.

Output
server_princ_name Returns a principal name. This name is registered for the authentication service in parameter authn_svc by the server referred to in argument 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. If the operation fails, this server_princ_name will be set to NULL.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
rpc_s_binding_incomplete Binding incomplete (no object ID and no endpoint).
rpc_s_comm_failure Communications failure.
rpc_s_no_princ_name No principal name registered.
rpc_s_mgmt_op_disallowed Not authorized for operation.
rpc_s_no_memory No storage available.
rpc_s_unknown_authn_service Unknown authentication service.
rpc_s_wrong_kind_of_binding Wrong kind of binding for operation.
**Usage**

The `rpc_mgmt_inq_server_princ_name` API obtains 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 the client’s identity. 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 routine `rpc_server_register_auth_info`.

The RPC runtime allocates storage for the string returned in `server_princ_name`. The application calls `rpc_string_free` to deallocate that storage.

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

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_binding_inq_object`
- `rpc_binding_set_auth_info`
- `rpc_ep_resolve_binding`
- `rpc_mgmt_set_authorization_fn`
- `rpc_server_register_auth_info`
- `rpc_string_free`
- `uuid_is_nil`
rpc_mgmt_inq_stats

Purpose
Returns RPC runtime statistics.

Used by client, server, or management applications.

Format
#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 specify 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. The RPC management interface is registered with the runtime (by the run time) for all RPC servers.

You can avoid this situation by calling the rpc_ep_resolve_binding routine to obtain a fully bound server binding handle.

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.

Possible status codes and their meanings:
- rpc_s_ok: Successful completion.
- rpc_s_binding_incomplete: Binding incomplete (no object ID and no endpoint).
- rpc_s_comm_failure: Communications failure.
- rpc_s_invalid_binding: Binding handle not valid.
- rpc_s_mgmt_op_disallowed: Management operation disallowed.
- rpc_s_no_memory: No storage available.
- rpc_s_exception_detected: Unexpected exception detected.
- rpc_s_wrong_kind_of_binding: Wrong kind of binding for operation.

Usage

The rpc_mgmt_inq_stats API returns statistics from the RPC runtime about a specified server. The RPC runtime allocates storage for the statistics vector. The application calls the rpc_mgmt_stats_vector_free routine to release the storage that the statistics vector used. See Statistics Vector on page 2-17 which lists the elements of 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 specifies an authorization function using the rpc_mgmt_set_authorization_fn API.
Return Values
None.

Related Information
Routines
rpc_ep_resolve_binding  rpc_mgmt_set_authorization_fn  rpc_mgmt_stats_vector_free
Purpose
Tells whether a server is listening for remote procedure calls.
Used by client, server, or management applications.

Format

```c
#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 specify 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. The RPC management interface is registered with the runtime (by the run time) for all RPC servers.

  You can avoid this situation by calling the `rpc_ep_resolve_binding` routine to obtain a fully bound server binding handle.

**Output**

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

  Possible status codes and their meanings:

  - `rpc_s_ok`: Successful completion.
  - `rpc_s_binding_incomplete`: Binding incomplete (no object ID and no endpoint).
  - `rpc_s_comm_failure`: Communications failure.
  - `rpc_s_invalid_binding`: Binding handle not valid.
  - `rpc_s_mgmt_op_disallowed`: Management operation disallowed.
  - `rpc_s_exception_detected`: Unexpected exception detected.
  - `rpc_s_wrong_kind_of_binding`: Wrong kind of binding for operation.

Usage

The `rpc_mgmt_is_server_listening` API 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` API.

By default, the RPC runtime allows all clients to remotely call this routine. To restrict remote calls of this routine, a server application specifies an authorization function using the `rpc_mgmt_set_authorization_fn` API.
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>
<thead>
<tr>
<th>Value Returned</th>
<th>Status Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>rpc_s_ok</td>
<td>The specified server is listening for remote procedure calls.</td>
</tr>
<tr>
<td>false</td>
<td>One of the status codes listed for the status parameter</td>
<td>The specified server is not listening for remote procedure calls, or the server cannot be reached.</td>
</tr>
</tbody>
</table>

Related Information

Routines

rpc_ep_resolve_binding    rpc_mgmt_set_authorization_fn    rpc_server_listen
Purpose
Establishes an authorization function for processing remote calls to a server's management routines.

Used by server applications.

Format
```
#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 calls this routine whenever the server run time receives a client request to run one of the RPC management routines.

Specify **NULL** to unregister a previously registered authorization function. 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>
<thead>
<tr>
<th>Called Remote Routine</th>
<th>requested_mgmt_operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_mgmt_inq_if_ids</code></td>
<td><code>rpc_c_mgmt_inq_if_ids</code></td>
</tr>
<tr>
<td><code>rpc_mgmt_inq_server_princ_name</code></td>
<td><code>rpc_c_mgmt_inq_princ_name</code></td>
</tr>
<tr>
<td><code>rpc_mgmt_inq_stats</code></td>
<td><code>rpc_c_mgmt_inq_stats</code></td>
</tr>
<tr>
<td><code>rpc_mgmt_is_server_listening</code></td>
<td><code>rpc_c_mgmt_is_server_listen</code></td>
</tr>
<tr>
<td><code>rpc_mgmt_stop_server_listening</code></td>
<td><code>rpc_c_mgmt_stop_server_listen</code></td>
</tr>
</tbody>
</table>

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

Possible status code and its meaning:
```
rpc_s_ok  Successful completion.
```
Usage

The `rpc_mgmt_set_authorization_fn` API 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 following table. An enable authorization allows all clients to run the remote routine and a disable authorization prevents all clients from running the remote routine.

<table>
<thead>
<tr>
<th>Remote Routine</th>
<th>Default Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_mgmt_inq_if_ids</code></td>
<td>enabled</td>
</tr>
<tr>
<td><code>rpc_mgmt_inq_server_princ_name</code></td>
<td>enabled</td>
</tr>
<tr>
<td><code>rpc_mgmt_inq_stats</code></td>
<td>enabled</td>
</tr>
<tr>
<td><code>rpc_mgmt_is_server_listening</code></td>
<td>enabled</td>
</tr>
<tr>
<td><code>rpc_mgmt_stop_server_listening</code></td>
<td>disabled</td>
</tr>
</tbody>
</table>

A server can change the default authorizations by calling `rpc_mgmt_set_authorization_fn` to specify an authorization function. When an authorization function is provided, the RPC runtime calls that function to control the running 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 run and the function returns the `rpc_s_ok` status code to the client. If the authorization function returns `false`, the `rpc_s_mgmt_op_disallowed` status code is returned.

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 run 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 run the requested management routine.

Return Values

None.

Related Information

Routines

- `rpc_mgmt_ep_unregister`
- `rpc_mgmt_inq_if_ids`
- `rpc_mgmt_inq_server_princ_name`
- `rpc_mgmt_inq_stats`
- `rpc_mgmt_is_server_listening`
- `rpc_mgmt_stop_server_listening`
**rpc_mgmt_set_cancel_timeout**

**Purpose**
Sets the lower bound on the time to wait before timing out after forwarding a cancelation.

Used by client applications.

**Format**

```c
#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 cancelation. To specify that a client waits an infinite amount of time, use 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.

  Possible status codes and their meanings:

  - `rpc_s_ok` Successful completion.
  - `rpc_s_pthread_getspecific_fail` POSIX error.
  - `rpc_s_pthread_setspecific_fail` POSIX error.
  - `rpc_s_no_memory` No storage available.

**Usage**

The `rpc_mgmt_set_cancel_timeout` API 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 wants to change the timeout value must call this routine in each thread.

For more information about canceled threads and orphaned remote procedure calls, see the “RPC Section,” of the Z/OS DCE Application Development Guide: Core Components.

**Return Values**

None.

**Related Information**

2-144 Application Development Reference Volumes 1 and 2
Routines

pthread_cancel  pthread_setcancel
Purpose
Sets the communication timeout value in a binding handle.

Used by client applications.

Format

```c
#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 communication 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.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_invalid_binding**: Binding handle not valid.
- **rpc_s_invalid_timeout**: Timeout value not valid.
- **rpc_s_wrong_kind_of_binding**: Wrong kind of binding for operation.

Usage

The `rpc_mgmt_set_com_timeout` API resets the communication 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 can revert to not less than the default timeouts for the protocol service. This means that after a short initial timeout establishing a connection is set, calls in progress are not timed out any sooner than the default.

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 of binding timeout values describes the DCE RPC predefined values that an application can use for the `timeout` parameter:

To change the default timeout value for the process you can set the `_EUV_RPC_COMM_TIMEOUT` environment variable. Note that the value specified by the `rpc_mgmt_set_com_timeout` API overrides the value in this environment variable. See [z/OS DCE Application Development Guide: Core Components](#) for more information on setting this environment variable.
Table 2-13. Predefined Timeout Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_binding_min_timeout</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_binding_default_timeout</td>
<td>5</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_binding_max_timeout</td>
<td>9</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_binding_infinite_timeout</td>
<td>10</td>
<td>Attempts to communicate forever.</td>
</tr>
</tbody>
</table>

Note that connection-oriented RPC handles the time-out value differently from Datagram RPC. Because connection-oriented RPC is based on a reliable transport layer, communications time-outs are not as significant as they are under datagram protocol. When `rpc_mgmt_set_com_timeout` is called on a binding using connection-oriented protocol, only the input parameter `rpc_c_binding_infinite_timeout` changes the binding's behavior. All other values are ignored.

Return Values

None.

Related Information

Routines

`rpc_mgmt_inq_com_timeout`
rpc_mgmt_stats_vector_free

Purpose
Frees a statistics vector.
Used by client, server, or management applications.

Format
#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.
Possible status code and its meaning:
    rpc_s_ok Successful completion.

Usage
The rpc_mgmt_stats_vector_free API releases the storage 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
None.

Related Information
Routines
rpc_mgmt_inq_stats
**rpc_mgmt_stop_server_listening**

**Purpose**
Tells a server to stop listening for remote procedure calls.

Used by client, server, or management applications.

**Format**

```c
#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 *local server* to stop listening for remote procedure calls, specify `NULL`.

  If the binding handle you specify 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. The DCE Host Daemon does not know which server instance to select from the local endpoint map because the RPC management interface is 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.

  Possible status codes and their meanings:

  - `rpc_s_ok`  
    Successful completion.
  - `rpc_s_binding_incomplete`  
    Binding incomplete (no object ID and no endpoint).
  - `rpc_s_comm_failure`  
    Communications failure.
  - `rpc_s_invalid_binding`  
    Binding handle is not valid.
  - `rpc_s_mgmt_op_disallowed`  
    Not authorized for operation.
  - `rpc_s_unknown_if`  
    Unknown interface.
  - `rpc_s_wrong_kind_of_binding`  
    Wrong kind of binding for operation.
  - `rpc_s_not_listening`  
    The server is not listening.
  - `rpc_s_exception_detected`  
    Exception detected.
  - `rpc_s_mutex_lock_fail`  
    Mutex lock fails. POSIX error.
  - `rpc_s_mutex_unlock_fail`  
    Mutex unlock fails. POSIX error.
**Usage**

The `rpc_mgmt_stop_server_listening` API 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. Calls that are running are allowed to complete.

After all calls are completed, `rpc_server_listen` returns to the caller.

By default, the RPC runtime does not allow any client to call this routine remotely. To allow clients to run this routine, a server application specifies an authorization function using `rpc_mgmt_set_authorization_fn`.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_ep_resolve_binding`
- `rpc_mgmt_set_authorization_fn`
- `rpc_server_listen`
**rpc_get_ae_name**

**Purpose**
Used to determine, the AENAME that is currently being used. For instance, the same server application can assume different AENAME values. Which value is used depends on the value of the environmental value _EUV_LOAD_BALANCE when the server is started.

Used by server applications.

**Format**

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

void rpc_get_ae_name(
    char **ae_name,
    unsigned32 *status);
```

**Parameters**

**Output**
- **ae_name**
  Contains a pointer to a null terminated string containing the AENAME associated with the server.
- **status**
  Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings:
- **rpc_s_invalid_arg**
  Argument not valid.

**Usage**

The **rpc_get_ae_name** API provides a way for the server to determine the AENAME at runtime.

The API returns a pointer to a character string. The pointer represents the AENAME that this server will use to identify equivalent instances to WLM.

The character string returned contains up to 19 characters containing a null terminated string. The string returned will be allocated by the runtime and should be freed by the caller using **rpc_string_free**.

Several conditions would return a null pointer:
- **rpc_set_ae_name** has not been previously issued.
- The environmental variable _EUV_LOAD_BALANCE was null or not set the first time one of the following APIs were issued.
  - **rpc_ep_register**
  - **rpc_ep_register_wlb**
  - **rpc_ep_register_no_replace**
  - **rpc_ep_register_no_replace_wlb**

One use of this API, would be to issue it just before issuing **rpc_server_listen**. This checks that the right AENAME is being used by the server so you can display or write in the log for problem debugging.

**Return Values**

None.

**Related Information**

Chapter 2. Remote Procedure Call 2-151
rpc_get_ae_name

Routines

rgc_ep_register  rpc_ep_register_no_replace  rpc_mgmt_ep_elt._inq_next_wlb
rgc_ep_register_wlb  rpc_ep_register_no_replace_wlb
rpc_set_ae_name

Purpose
Sets the AENAME the server will use to identify equivalent instances of itself to Workload Management.

Used by server applications.

Format
#include <dce/rpc.h>

void rpc_set_ae_name(
    char ae_name[],
    unsigned32 *status );

Parameters

Input
ae_name[]                Specifies the AENAME of 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.
                        Possible status codes and their meanings:
                        ept_s_cant_perform_op     Cannot perform requested operation.
                        rps_s_invalid_arg         Argument not valid.

Usage

The rpc_set_ae_name API passes a character array representing the AENAME that this server will use to identify equivalent
instances to WLM.

The AENAME used for the server will default to the value of the _EUV_LOAD_BALANCE environmental variable unless this
API is issued before the first call to one of the following API's:

- rpc_ep_register ,
- rpc_ep_register_wlb ,
- rpc_ep_register_no_replace ,
- rpc_ep_register_no_replace_wlb.

If this API is issued after one of the 4 previous API's, the result will be ept_s_cant_perform_op.

The API will have no effect if status contains a null pointer. If the character array ae_name contains more than 18 characters,
only the first 18 will be used and no error signalled. Issue this API as many times as you want before the first call to one of
the following API's:

- rpc_ep_register ,
- rpc_ep_register_wlb ,
- rpc_ep_register_no_replace ,
- rpc_ep_register_no_replace_wlb.

The character array passed consists of 18 characters plus a null terminated string. When ae_name is a null pointer and none
of the following APIs were previously issued:

- rpc_ep_register ,
- rpc_ep_register_wlb ,
rpc_set_ae_name

- rpc_ep_register_no_replace
- rpc_ep_register_no_replace_wlb

Runtime will forget any ae_name value previously set and use the null string.

If this routine is successful, the environmental variable EUV_LOAD_BALANCE will be ignored. Once a server has issued one of the following API's:

- rpc_ep_register
- rpc_ep_register_wlb
- rpc_ep_register_no_replace
- rpc_ep_register_no_replace_wlb

the AENAME is fixed and cannot be changed without restarting the server.

If you prefer the environmental variable EUV_LOAD_BALANCE to override this API, issue the C function getenv, then rpc_set_ae_name. This must be done prior to issuing any of the following APIs:

- rpc_ep_register
- rpc_ep_register_wlb
- rpc_ep_register_no_replace
- rpc_ep_register_no_replace_wlb

If the ae_name returned is not an empty string, do nothing. If the ae_name returned is an empty string, issue rpc_set_ae_name. Use the default name you want. This is one way to set up your server to have a default AENAME in case the user forgets to set up the environmental variable.

If ae_name is too long, only the first 18 characters will be saved to represent the AENAME. No error status codes will be returned to indicated the entire input was not saved.

If any of the following API's were issued and an application is running, you can not reset the AENAME:

- rpc_ep_register
- rpc_ep_register_wlb
- rpc_ep_register_no_replace
- rpc_ep_register_no_replace_wlb

Return Values
None.

Related Information

Routines

rpc_ep_register            rpc_ep_register_no_replace       rpc_mgmt_ep_elt_inq_next_wlb
rpc_ep_register_wlb        rpc_ep_register_no_replace_wlb       rpc_get_ae_name
**rpc_network_inq_protseqs**

**Purpose**
Returns all protocol sequences supported by both the RPC runtime and the operating system. Used by client or server applications.

**Format**

```c
#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.

Possible status codes and their meanings:

- `rpc_s_ok`: Successful completion.
- `rpc_s_no_protseqs`: No supported protocol sequences.
- `rpc_s_no_memory`: No storage available.

**Usage**

The `rpc_network_inq_protseqs` API 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 storage used by the vector.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_network_is_protseq_valid`
- `rpc_protseq_vector_free`
rpc_network_is_protseq_valid

### Purpose
Indicates whether the specified protocol sequence is supported by both the RPC runtime and the operating system.

Used by client or server applications.

### Format
```c
#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 [Table 2-5 on page 2-16](#) for a list of acceptable values.

  The `rpc_network_is_protseq_valid` API 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.

  Possible status codes and their meanings:
  - `rpc_s_ok`  
    Successful completion.
  - `rpc_s_invalid_rpc_protseq`  
    Protocol sequence is not valid.
  - `rpc_s_protseq_not_supported`  
    Protocol sequence not supported on this host.

### Usage
The `rpc_network_is_protseq_valid` API 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.

### Related Information
Routines

rpc_network_inq_protseqs        rpc_string_binding_parse
rpc_ns_binding_export

Purpose
Establishes a name service database entry with binding handles or object UUIDs for a server.
Used by server applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify the value rpc_c_ns_syntax_default.
For more information, see the name_syntax section in [Frequently Used Routine Arguments" on page 2-20.

entry_name Specifies the entry name to which binding handles and object UUIDs are exported. It 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 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.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
rpc_s_incomplete_name Incomplete name.
rpc_s_invalid_binding Binding handle is not valid.
rpc_s_invalid_name_syntax Name syntax is not valid.
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.
Usage

The rpc_ns_binding_export API 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 two items: an interface, and the server binding handles that can be used by a client 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. 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 API 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_if
  - rpc_server_use_protseq_ep
  - 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. For more information on binding handles, see “Binding Handle” on page 2-9.

  A server that uses the rpc_server_use_protseq or rpc_server_use_all_protseqs has dynamic endpoints. The bindings exported to the name service database will contain a NULL endpoint. The servers must also register with the local endpoint map by calling rpc_ep_register or rpc_ep_register_no_replace.

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” on page 2-17.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

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

None.
Related Information

Routines

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_ep_register</td>
<td>rpc_ns_mgmt_entry_create</td>
<td>rpc_server_use_protseq</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace</td>
<td>rpc_server_inq_bindings</td>
<td>rpc_server_use_protseq_ep</td>
</tr>
<tr>
<td>rpc_ns_binding_unexport</td>
<td>rpc_server_use_all_protseqs</td>
<td>rpc_server_use_protseq_if</td>
</tr>
<tr>
<td>rpc_ns_mgmt_binding_unexport</td>
<td>rpc_server_use_all_protseqs_if</td>
<td></td>
</tr>
</tbody>
</table>
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.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable,
specify the value rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20

entry_name Specifies the entry name where the search for compatible binding handles begins. It can be
the global or cell-relative name.
To use the entry name found in the RPC_DEFAULT_ENTRY environment variable, specify
NULL or a null string (\0) for this parameter. When this entry name is used, the RPC runtime
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. 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 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 non-nil 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 non-nil object UUID.

Output
import_context Returns the name service handle for use with the rpc.ns_binding_import_next and
rpc.ns_binding_import_done APIs.
status Returns the status code from this routine. This status code indicates whether the routine
completed successfully or, if not, why not.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
### Usage

The `rpc_ns_binding_import_begin` API 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 parameters to this routine control the operation `rpc_ns_binding_import_next`.

After importing binding handles, the client calls `rpc_ns_binding_import_done` to delete the import context.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

### Permissions Required

None.

### Return Values

None.

### Related Information

#### Routines

- `rpc_ns_binding_import_done`
- `rpc_ns_binding_import_next`
- `rpc_ns_mgmt_handle_set_exp_age`
**rpc_ns_binding_import_done**

**Purpose**
Deletes the import context for searching the name service database.

Used by client applications.

**Format**

```c
#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`.

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

  Possible status codes and their meanings:

  - `rpc_s_ok`: Successful completion.
  - `rpc_s_invalid_ns_handle`: Name service handle is not valid.

**Usage**

The `rpc_ns_binding_import_done` API 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 of the success in making remote procedure calls.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

**Permissions Required**

None.

**Return Values**

None.

**Related Information**

Chapter 2. Remote Procedure Call  2-163
rpc_ns_binding_import_done

Routines

rpc_ns_binding_import_begin  rpc_ns_binding_import_next
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.

Format

```c
#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 `rpc_ns_binding_import_begin`.

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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_class_version_mismatch` RPC class version mismatch.
- `rpc_s_entry_not_found` Name service entry not found.
- `rpc_s_invalid_ns_handle` Name service handle is not valid.
- `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.
- `sec_rgy_server_unavailable` Register server unavailable.

Usage

The `rpc_ns_binding_import_next` API 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 `rpc_ns_binding_import_next` routine calls `rpc_ns_binding_lookup_next`, that, in turn, obtains a vector of server binding handles from the name service database. Next, the `rpc_ns_binding_import_next` routine randomly selects one of the elements from the vector.

The `rpc_ns_binding_import_next` API 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` API as follows:

- If `obj_uuid` contains a non-nil 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. This object UUID is selected in an unspecified way.

Multiple calls to `rpc_ns_binding_import_next` may not return different object UUIDs to the application. 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 return one binding for each compatible server address, not the cross product of all compatible server addresses with all exported UUIDs.

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

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` API again.

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` API allocates storage for the `returned binding` parameter. When a client application finishes with the binding handle, it must call the `rpc_binding_free` API to deallocate the storage. Each call to the `rpc_ns_binding_import_next` routine requires a corresponding call to `rpc_binding_free`.

The client calls the `rpc_ns_binding_import_done` API after it has satisfactorily used one or more returned server binding handles. The `rpc_ns_binding_import_done` API 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.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

### 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
None.

### Related Information

#### Routines
- `rpc_ns_binding_import_begin`
- `rpc_ns_binding_import_done`
- `rpc_ns_binding_inq_entry_name`
- `rpc_ns_binding_lookup_begin`
- `rpc_ns_binding_lookup_done`
- `rpc_ns_binding_lookup_next`
- `rpc_ns_binding_select`
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.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify the value rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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 was completed successfully or, if not, the reason.

Possible status codes and their meanings:
- rpc_s_ok: Successful completion.
- rpc_s_incomplete_name: Incomplete name.
- rpc_s_invalid_binding: Binding handle is not valid.
- rpc_s_invalid_name_syntax: Name syntax is not valid.
- rpc_s_no_entry_name: No entry name for binding.
- rpc_s_unsupported_name_syntax: Unsupported name syntax.
- rpc_s_no_env_setup: RPC_DEFAULT_ENTRY not setup.

Usage
The rpc_ns_binding_inq_entry_name API 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 storage for the string returned in the entry_name parameter. Your application calls rpc_string_free to deallocate that storage.

An entry name is associated only with binding handles returned from the rpc_ns_binding_import_next, rpc_ns_binding_lookup_next, and rpc_ns_binding_select APIs.
rpc_namespace_binding_inq_entry_name

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.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

Permissions Required

None.

Return Values

None.

Related Information

Routines

rpc_binding_from_string_binding  rpc_ns_binding_lookup_next  rpc_string_free
rpc_ns_binding_import_next  rpc_ns_binding_select
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.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify the value rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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, specify NULL or a null string (\0) for this parameter. When this entry name is used, the RPC runtime 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. 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 non-nil 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 non-nil 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 rpc_ns_binding_lookup_next and rpc_ns_binding_lookup_done routines.
### rpc_ns_binding_lookup_begin

**status**

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

Possible status codes and their meanings:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>rpc_s_incomplete_name</td>
<td>Incomplete name.</td>
</tr>
<tr>
<td>rpc_s_invalid_name_syntax</td>
<td>Name syntax is not valid.</td>
</tr>
<tr>
<td>rpc_s_invalid_object</td>
<td>Object is not valid.</td>
</tr>
<tr>
<td>rpc_s_no_env_setup</td>
<td>RPC_DEFAULT_ENTRY not setup.</td>
</tr>
<tr>
<td>rpc_s_no_memory</td>
<td>No storage available.</td>
</tr>
<tr>
<td>rpc_s_not_rpc_entry</td>
<td>Not an RPC entry.</td>
</tr>
<tr>
<td>rpc_s_unsupported_name_syntax</td>
<td>Unsupported name syntax.</td>
</tr>
</tbody>
</table>

### Usage

The **rpc_ns_binding_lookup_begin** API 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 **rpc_ns_binding_lookup_next** API.

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

**Note:** With z/OS, any thread calling this API cannot be canceled with the **pthread_cancel()** routine. This avoids potential resource clean up problems with this routine.

### Permissions Required

None.

### Return Values

None.

### Related Information

#### Routines

- **rpc_ns_binding_lookup_done**
- **rpc_ns_binding_lookup_next**
- **rpc_ns_mgmt_handle_set_exp_age**
**rpc_ns_binding_lookup_done**

**Purpose**
Deletes the lookup context for searching the name service database.

Used by client applications.

**Format**

```c
#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`.

- **status**
  Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.
  Possible status codes and their meanings:

  - `rpc_s_ok`:
    Successful completion.

  - `rpc_s_invalid_ns_handle`:
    Name service handle is not valid.

**Usage**

The `rpc_ns_binding_lookup_done` API 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 of success in making remote procedure calls.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

**Permissions Required**

None.

**Return Values**

None.

**Related Information**
rpc_ns_binding_lookup_done

Routines
rpc_ns_binding_lookup_begin            rpc_ns_binding_lookup_next
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.

Format
#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 API.

Output
binding_vec
Returns a vector of compatible 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.

Possible status codes and their meanings:

- rpc_s_ok
  - Successful completion.
- rpc_s_class_version_mismatch
  - RPC class version mismatch.
- rpc_s_entry_not_found
  - Name service entry not found.
- rpc_s_invalid_ns_handle
  - Name service handle is not valid.
- 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.
- sec_rgy_server_unavailable
  - Register server unavailable.

Usage
The rpc_ns_binding_lookup_next API 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 parameter binding_max_count in the rpc_ns_binding_lookup_begin API.

A similar routine is rpc_ns_binding_import_next that returns one compatible server binding handle.

The rpc_ns_binding_lookup_next API communicates only with the name service database, not directly with servers.

This routine searches entries in the name service database, returning compatible server binding handles from each entry. Search operations obey the following rules for searching the entries:

- At each entry visited, the search operations randomly process binding information, then group members, and then profile members. Profile members (but not group members) with different priorities are returned according to their priorities, highest priority first.
rpc_ns_binding_lookup_next

- 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 [Parameters on page 2-169](#)) contains binding handles as well as an RPC group or a profile, rpc_ns_binding_lookup_next returns the binding handles from entry_name before searching the group or profile. The routine 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 parameter object_uuid of rpc_ns_binding_lookup_begin as follows:

- If object_uuid contains a non-nil 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. This object UUID is selected in an unspecified way.

The binding handles returned to the application from a given entry may not contain different object UUIDs. Each name service entry stores server address information separately from exported object UUIDs. One or more calls to rpc_ns_binding_lookup_next returns one binding for each compatible server address, not the cross product of all compatible server addresses with all exported 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 APIs 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 of 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 rpc_ns_binding_lookup_next looks 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 is 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 API allocates storage for the returned binding_vec. When a client application finishes with the vector, it must call the rpc_binding_vector_free API to deallocate the storage. 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, to delete 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 rpc_ns_binding_lookup_begin). The order of binding handles returned can be different for each new search. The order in which binding handles are returned to an application can be different each time the application is run.
Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

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

None.

Related Information

Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_binding_to_string_binding</code></td>
<td><code>rpc_ns_binding_inq_entry_name</code></td>
<td><code>rpc_ns_binding_select</code></td>
</tr>
<tr>
<td><code>rpc_binding_vector_free</code></td>
<td><code>rpc_ns_binding_lookup_begin</code></td>
<td><code>rpc_string_binding_parse</code></td>
</tr>
<tr>
<td><code>rpc_ns_binding_import_next</code></td>
<td><code>rpc_ns_binding_lookup_done</code></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 2. Remote Procedure Call 2-175
rpc_ns_binding_select

 rpc_ns_binding_select

Purpose
Returns a binding handle from a list of compatible server binding handles.
Used by client applications.

Format
#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 refers to 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. This status code indicates whether the routine
completed successfully or, if not, why not.

Possible status codes and their meanings:

rpc_s_ok                  Successful completion.
rpc_s_no_moreBindings     No more bindings.

Usage

The rpc_ns_binding_select API 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_moreBindings
and returns the value NULL in binding. The binding vector is freed, and an error occurs if rpc_binding_vector_free is called.

The select operation allocates storage for the data referred to by the returned binding parameter. When a client finishes with
the binding handle, it calls the rpc_binding_free API 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. The
rpc_binding_to_string_binding and rpc_string_binding_parse APIs are useful to the applications because the routines work
together to extract the individual fields of a binding handle for examination.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential
resource clean up problems with this routine.

Permissions Required

None.
Return Values

None.

Related Information

Routines

rpc_binding_free  rpc_ns_binding_lookup_next  rpc_string_binding_parse
rpc_binding_to_string_binding
Purpose
Removes the binding handles for an interface or object UUIDs from an entry in the name service database.
Used by server applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify the value rpc_c_ns_syntax_default.
For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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.
Possible status codes and their meanings:

table
<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>rpc_s_class_version_mismatch</td>
<td>RPC class version mismatch.</td>
</tr>
<tr>
<td>rpc_s_entry_not_found</td>
<td>Name service entry not found.</td>
</tr>
<tr>
<td>rpc_s_incomplete_name</td>
<td>Incomplete name.</td>
</tr>
<tr>
<td>rpc_s_interface_not_found</td>
<td>Interface not found.</td>
</tr>
<tr>
<td>rpc_s_invalid_name_syntax</td>
<td>Name syntax is not valid.</td>
</tr>
<tr>
<td>rpc_s_invalid_vers_option</td>
<td>Version option is not valid.</td>
</tr>
<tr>
<td>rpc_s_name_service_unavailable</td>
<td>Name service unavailable.</td>
</tr>
<tr>
<td>rpc_s_no_ns_permission</td>
<td>No permission for name service operation.</td>
</tr>
<tr>
<td>rpc_s_not_all_objs_unexported</td>
<td>Not all objects unexported.</td>
</tr>
<tr>
<td>rpc_s_nothing_to_unexport</td>
<td>Nothing to unexport.</td>
</tr>
</tbody>
</table>
Usage

The `rpc_ns_binding_unexport` API 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` API 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 the `rpc_ns_mgmt_binding_unexport` API.

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, the routine 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, routine `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 the `rpc_ns_binding_export` and `rpc_ns_binding_unexport` causes the name service to repeatedly remove and replace the same entry and can cause performance problems.

Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

Routines

`rpc_ep_unregister`  `rpc_ns_binding_export`  `rpc_ns_mgmt_binding_unexport`
rpc_ns_entry_expand_name

Purpose
Expands the name of a name service entry.
Used by client, server, or management applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify value of rpc_c_ns_syntax_default.
For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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 because 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.
Possible status codes and their meanings:

rpc_s_ok Successful completion.
rpc_s_incomplete_name Incomplete name.

Usage

An application calls the rpc_ns_entry_expand_name API to obtain a fully expanded entry name.
The RPC runtime allocates storage for the returned expanded_name. The application is responsible for calling the rpc_string_free API 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

On 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 that introduces the DCE Directory Service in z/OS DCE Administration Guide.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.
Permissions Required
None.

Return Values
None.

Related Information
Routines
rpc_string_free
Purpose
Resolves the cell namespace components of a name and returns partial results.
Used by client applications.

Format

```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, supply a value of `rpc_c_ns_syntax_default`.
  
  For more information, see the `name_syntax` section in "Frequently Used Routine Arguments" on page 2-20.

- `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 form.

**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.
  
  The possible status codes and their meanings are as follows:

  - `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.
Usage

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 (i.e., 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 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 (non-empty) 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 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.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Return Values

None.

Related Information

Routines

`rpc_ns_binding_*`
Purpose
Creates an inquiry context for viewing the objects of an entry in the name service database.

Used by client, server, or management applications.

Format

```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, specify `rpc_c_ns_syntax_default`.
  For more information, see the *name_syntax* section in "Frequently Used Routine Arguments" on page 2-20.

- **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 `rpc_ns_entry_object_inq_next` and `rpc_ns_entry_object_inq_done` APIs.

- **status**
  Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.
  Possible status codes and their meanings:
  - `rpc_s_ok` Successful completion.
  - `rpc_s_incomplete_name` Incomplete name.
  - `rpc_s_invalid_name_syntax` Name syntax is not valid.
  - `rpc_s_unsupported_name_syntax` Unsupported name syntax.

Usage

The `rpc_ns_entry_object_inq_begin` API 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` API to delete the inquiry context.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

None.
Return Values

None.

Related Information

Routines

rpc_ns_binding_export  rpc_ns_entry_object_inq_next  rpc_ns_mgmt_handle_set_exp_age
rpc_ns_entry_object_inq_done
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.

Format
#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.
Possible status codes and their meanings:
   rpc_s_ok Successful completion.
   rpc_s_invalid_ns_handle Name service handle is not valid.

Usage
The rpc_ns_entry_object_inq_done API 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 API.
Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential
resource clean up problems with this routine.

Permissions Required
None.

Return Values
None.

Related Information
Routines

rpc_ns_entry_object_inq_begin  rpc_ns_entry_object_inq_next
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.

Format
#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 API.

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.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
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 Name service handle is not valid.
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.

Usage
The rpc_ns_entry_object_inq_next API 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 API 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 specifies the storage 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 API 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.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential
resource clean up problems with this routine.
Permissions Required

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

Return Values

None.

Related Information

Routines

rpc_ns_binding_export  rpc_ns_entry_object_inq_begin  rpc_ns_entry_object_inq_done
Purpose
Deletes a group attribute.
Used by client, server, or management applications.

Format
#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 group_name parameter.
To use the syntax specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.
For more information, see the name_syntax section in Frequently Used Routine Arguments on page 2-20.

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.
Possible status codes and their meanings:

    rpc_s_ok Successful completion.
    rpc_s_entry_not_found Name service entry not found.
    rpc_s_incomplete_name Incomplete name.
    rpc_s_invalid_name_syntax Name syntax is not valid.
    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.

Usage

The rpc_ns_group_delete API 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.
Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

Permissions Required
You need write permission to the CDS object entry (the target group entry).
Return Values

None.

Related Information

Routines

rpc_ns_group_member_add    rpc_ns_group_member_delete
 rpc_ns_group_mbr_add

Purpose
Adds an entry name to a group; if necessary, creates the group entry.

Used by client, server, or management applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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

Possible status codes and their meanings:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>rpc_s_class_version_mismatch</td>
<td>RPC class version mismatch.</td>
</tr>
<tr>
<td>rpc_s_incomplete_name</td>
<td>Incomplete name.</td>
</tr>
<tr>
<td>rpc_s_invalid_name_syntax</td>
<td>Name syntax is not valid.</td>
</tr>
<tr>
<td>rpc_s_name_service_unavailable</td>
<td>Name service unavailable.</td>
</tr>
<tr>
<td>rpc_s_no_ns_permission</td>
<td>No permission for name service operation.</td>
</tr>
<tr>
<td>rpc_s_unsupported_name_syntax</td>
<td>Unsupported name syntax.</td>
</tr>
</tbody>
</table>
Usage

The `rpc_ns_group_mbr_add` API 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 group 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. 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 without creating the group entry.

Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

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

None.

Related Information

Routines

- `rpc_ns_group_mbr_remove`
- `rpc_ns_mgmt_entry_create`
Purpose
Creates an inquiry context for viewing group members.

Used by client, server, or management applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.
  For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

  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 API.
  To use the syntax specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify the rpc_c_ns_syntax_default.
  For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

Output

  inquiry_context Returns a name service handle for use with the rpc_ns_group_mbr_inq_next and rpc_ns_group_mbr_inq_done APIs.

  status Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.
  Possible status codes and their meanings:

    rpc_s_ok Successful completion.
    rpc_s_incomplete_name Incomplete name.
    rpc_s_invalid_name_syntax Name syntax is not valid.
    rpc_s_unsupported_name_syntax Unsupported name syntax.

Usage

The rpc_ns_group_mbr_inq_begin API 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.
**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

**Permissions Required**

None.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_ns_group_mbr_add`
- `rpc_ns_group_mbr_inq_next`
- `rpc_ns_mgmt_handle_set_exp_age`
- `rpc_ns_group_mbr_inq_done`
rpc_ns_group_mbr_inq_done

Purpose
Deletes the inquiry context for a group.
Used by client, server, or management applications.

Format
#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.
Possible status codes and their meanings:
    rpc_s_ok Successful completion.
    rpc_s_invalid_ns_handle Name service handle is not valid.

Usage
The rpc_ns_group_mbr_inq_done API 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 API.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

Permissions Required
None.

Return Values
None.

Related Information
Routines

rpc_ns_group_mbr_inq_begin  rpc_ns_group_mbr_inq_next
rpc_ns_group_mbr_inq_next

Purpose
Returns one member name at a time from a group.
Used by client, server, or management applications.

Format
#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/Output
inquiry_context Specifies a name service handle. This handle is returned from the
rpc_ns_group_mbr_inq_begin API.

Output
member_name Returns a pointer to a (global) RPC group member name.
The syntax of the returned name is specified by the member_name_syntax parameter in
rpc_ns_group_mbr_inq_begin.
Specify NULL to prevent the routine from returning this parameter. The application does not call
the rpc_string_free API.

status Returns the status code from this routine. This status code indicates whether the routine
completed successfully or, if not, why not.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
 rpc_s_class_version_mismatch RPC class version mismatch.
 rpc_s_entry_not_found Name service entry not found.
 rpc_s_invalid_ns_handle Name service handle is not valid.
 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.

Usage
The rpc_ns_group_mbr_inq_next API returns one member of the RPC group specified by the group_name parameter in the
rpc_ns_group_mbr_inq_begin API.

An application can view all the members of an RPC group by repeatedly calling the rpc_ns_group_mbr_inq_next API. 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 storage for the
returned member_name. The application calls rpc_string_free API for each returned member_name string.
After viewing the RPC group’s members, the application must call the `rpc_ns_group_mbr_inq_done` API to delete the inquiry context.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

### Permissions Required

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

### Return Values

None.

### Related Information

#### Routines

- `rpc_ns_group_mbr_inq_begin`
- `rpc_ns_group_mbr_inq_done`
- `rpc_string_free`
Purpose
Removes a member from a group.
Used by client, server, or management applications.

Format
#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 parameter group_name.

    To use the syntax specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

    For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

    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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

    For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

    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.

Possible status codes and their meanings:

    rpc_s_ok Successful completion.
    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 Name syntax is not valid.
    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.
Usage

The `rpc_ns_group_mbr_remove` API removes a member from the NSI group attribute in the `group_name` entry.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

Routines

`rpc_ns_group_mbr_add`
rpc_ns_import_ctx_add_eval

Purpose
Adds an evaluation routine to an import context.

Used by client applications.

Format
#include <dce/rpc.h>

void rpc_ns_import_ctx_add_eval(
    rpc_ns_handle_t *import_context,
    unsigned32 func_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_c_eval_type_codesets or rpc_c_custom_eval_type_codesets.
eval_args An opaque data type containing data used by the evaluation routine.

Client applications adding one of the DCE RPC code sets evaluation routines (such as rpc_cs_eval_with_universal() or rpc_cs_eval_without_universal()) should specify the server's NSI entry name (which is stored in the I18N_SERVER_ENTRY environment variable) 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 this parameter indicates that it does not return a value.

Client applications adding one of the DCE RPC code sets evaluation routines (such as rpc_cs_eval_with_universal() or rpc_cs_eval_without_universal()) should 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 that performs application-specific cleanup.

Client applications adding one of the DCE RPC code sets evaluation routines (such as rpc_cs_eval_with_universal() or rpc_cs_eval_without_universal()) should specify NULL in this parameter.

Input/Output
import_context Returns the name service handle which contains the rpc_ns_binding_import_next() and rpc_ns_binding_import_done() routines.

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

rpc_s_ok Success.
rpc_s_invalid_function_type Function type specified in func_type parameter is invalid.
rpc_s_no_memory The RPC runtime could not allocate heap storage.
rpc_s_invalid_ns_handle The import_context parameter was not valid.
Usage

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()`.

At the OSF DCE 1.1 level (which includes z/OS DCE) 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 the DCE intermediate code set, which is the ISO 10646 (or "universal") code set. If this is chosen, the `free_func` parameter should be specified as `NULL`.

- **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 will return `rpc_s_no_compat_codesets` to the `rpc_ns_binding_import_next()` routine. If this is chosen, the `free_func` parameter should be specified as `NULL`.

- **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**

None

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_cs_eval_with_universal`
- `rpc_cs_eval_without_universal`
- `rpc_ns_binding_import_begin`
- `rpc_ns_binding_import_done`
- `rpc_ns_binding_import_next`
- `rpc_ns_mgmt_handle_set_exp_age`
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.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in [Frequently Used Routine Arguments" on page 2-20]

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 lists the accepted values for this parameter:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_vers_all</td>
<td>Unexports (that is, 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.</td>
</tr>
<tr>
<td>rpc_c_vers_compatible</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_exact</td>
<td>Removes those bindings for the interface UUID in if_id with the same major and minor versions as in if_id.</td>
</tr>
<tr>
<td>rpc_c_vers_major_only</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_upto</td>
<td>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...</td>
</tr>
</tbody>
</table>
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.

Possible status codes and their meanings:

- **rpc_s_ok** — Successful completion.
- **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** — Name syntax is not valid.
- **rpc_s_incomplete_name** — Interface not found.
- **rpc_s_invalid_vers_option** — Version option is not valid.
- **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_not_rpc_entry** — Not an RPC entry.
- **rpc_s_unsupported_name_syntax** — Unsupported name syntax.

**Usage**

The `rpc_ns_mgmt_binding_unexport` API 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` API 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, routine `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` API 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` APIs causes the name service to repeatedly remove and replace the same entry and can cause performance problems.

**Note:** With z/OS, any thread calling this API cannot be canceled with the **pthread_cancel()** routine. This avoids potential resource clean up problems with this routine.
Permissions Required
You need both read permission and write permission to the CDS object entry (the target name service entry).

Return Values
None.

Related Information
Routines
rpc_mgmt_ep_unregister   rpc_ns_binding_export   rpc_ns_binding_unexport
rpc_ns_mgmt_entry_create

Purpose
Creates an entry in the name service database.

Used by management applications.

Format
#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 specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

entry_name Specifications 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.

Possible status codes and their meanings:
- rpc_s_ok Successful completion.
- rpc_s_entry_already_exists Name service entry already exists.
- rpc_s_incomplete_name Incomplete name.
- rpc_s_invalid_name_syntax Name syntax is not valid.
- 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.

Usage
The rpc_ns_mgmt_entry_create API 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.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

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.
rpc_ns_mgmt_entry_create

Return Values

None.

Related Information

Routines

rpc_ns_mgmt_entry_delete
**rpc_ns_mgmt_entry_delete**

**Purpose**
Deletes an entry from the name service database.

Used by management applications.

**Format**

```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, provide `rpc_c_ns_syntax_default`.
  
  For more information, see the *name_syntax* section in "Frequently Used Routine Arguments" on page 2-20.

- **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.

  Possible status codes and their meanings:
  
  - **rpc_s_ok**
    Successful completion.
  - **rpc_s_entry_not_found**
    Name service entry not found.
  - **rpc_s_incomplete_name**
    Incomplete name.
  - **rpc_s_invalid_name_syntax**
    Name syntax is not valid.
  - **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.

**Usage**

The **rpc_ns_mgmt_entry_delete** API removes an RPC entry from the name service database.

Management applications use this routine only when an entry is no longer needed, for example, 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. Because 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**
rpc_ns_mgmt_entry_delete

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.

Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

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

None.

Related Information

Routines

rpc_ns_mgmt_entry_create
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.

Format

```c
#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 the `entry_name` parameter.
  
  To use the syntax specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, specify `rpc_c_ns_syntax_default`.
  
  For more information, see the `name_syntax` section in [“Frequently Used Routine Arguments” on page 2-20](#).

- **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. This status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_entry_not_found**: Name service entry not found.
- **rpc_s_incomplete_name**: Incomplete name.
- **rpc_s_invalid_name_syntax**: Name syntax is not valid.
- **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.

Usage

The `rpc_ns_mgmt_entry_inq_if_ids` API returns an interface identifier vector containing the interfaces of binding handles exported to parameter `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 description of the `rpc_ns_mgmt_inq_exp_age` routine contains an explanation of the expiration age.

The application calls `rpc_if_id_vector_free` to release storage used by the returned vector.
Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

Routines

rpc_if_id_vector_free    rpc_if_inq_id    rpc_ns_binding_export
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.

Format
```
#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 the
  `rpc_ns_mgmt_read_codesets()` routine or the `rpc_rgy_get_codesets()` routine.

**Output**

- **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.

Usage

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 registry. 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

None.

Related Information
rpc_ns_mgmt_free_codesets

Routines

rpc_ns_mgmt_read_codesets  rpc_rgy_get_codesets
**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.

**Format**
```c
#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 (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` API; 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.

Possible status codes and their meanings:
- `rpc_s_ok`: Successful completion.
- `rpc_s_invalid_ns_handle`: Name service handle is not valid.

**Usage**
The `rpc_ns_mgmt_handle_set_exp_age` API 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 is the expiration age set for the name service handle). If the actual age exceeds the handle's expiration age, the operation 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` API operates as follows:

1. An RPC NSI begin operation, such as the one that routine `rpc_ns_group_mbr_inq_begin` performs, 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.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

### 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 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. 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. The client can retry the import operation and, after `rpc_ns_binding_import_begin` ends, include a `rpc_ns_mgmt_handle_set_exp_age` call 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.

### Permissions Required

None.

### Return Values

None.

### Related Information

**Routines**

- `rpc_ns_binding_import_begin`
- `rpc_ns_binding_lookup_begin`
- `rpc_ns_entry_object_inq_begin`
- `rpc_ns_group_mbr_inq_begin`
- `rpc_ns_mgmt_inq_exp_age`
- `rpc_ns_mgmt_set_exp_age`
- `rpc_ns_mgmt_set_exp_age`
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.

Format
#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.

Possible status code and its meaning:
rpc_s_ok
Successful completion.

Usage
The rpc_ns_mgmt_inq_exp_age API 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 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 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-specified global expiration age for the default by calling rpc_ns_mgmt_set_exp_age. The rpc_ns_mgmt_inq_exp_age API returns the current global expiration age, whether it is a default or a user-specified value.

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

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

Permissions Required
None.
rpc_ns_mgmt_inq_exp_age

Return Values
None.

Related Information
Routines
rpc_ns_mgmt_handle_set_exp_age  rpc_ns_mgmt_set_exp_age
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.

Format
```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, provide `rpc_c_ns_syntax_default`.
  
  For more information, see the `name_syntax` section in "Frequently Used Routine Arguments" on page 2-20.

- **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.

The possible status codes and their meanings are as follows:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_ok</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>rpc_s_invalid_name_syntax</td>
<td>Name syntax is not valid.</td>
</tr>
<tr>
<td>rpc_s_mgmt_bad_type</td>
<td>Unsupported attribute type was given to NSI; it should be one of the pre-architected types.</td>
</tr>
<tr>
<td>rpc_s_name_service_unavailable</td>
<td>Name service unavailable.</td>
</tr>
<tr>
<td>rpc_s_no_permission</td>
<td>No permission for name service operation.</td>
</tr>
<tr>
<td>rpc_s_incomplete_name</td>
<td>Incomplete name.</td>
</tr>
<tr>
<td>rpc_s_no_memory</td>
<td>No memory for RPC runtime. It was either unable to allocate heap storage, or a fixed buffer supplied to IDL Encoding Services was not large enough.</td>
</tr>
</tbody>
</table>

Usage

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 set 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.

Applications that call `rpc_ns_mgmt_read_codesets()` to allocate the `code_sets_array` should call `rpc_ns_mgmt_free_codesets()` to free this storage after the application has finished processing it.

**Permissions Required**

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

**Return Values**

None.

**Related Information**

**Routines**

- `dce_cs_loc_to_rgy`
- `dce_rgy_to_loc`
- `rpc_ns_mgmt_free_codesets`
- `rpc_cs_eval_with_universal`
- `rpc_cs_eval_without_universal`
- `rpc_ns_mgmt_remove_attribute`
- `rpc_ns_mgmt_set_attribute`
- `rpc_rgy_get_codesets`
- `rpc_rgy_get_max_bytes`
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.

Format
```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, provide `rpc_c_ns_syntax_default`.
  
  For more information, see the `name_syntax` section in "Frequently Used Routine Arguments" on page 2-20.

- `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 OSF DCE 1.1 (which includes z/OS DCE), 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.

  The possible status codes and their meanings are as follows:

  - `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_mgmnt_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.
**Usage**

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 OSF DCE 1.1 (which includes z/OS DCE), 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. This might be done if a new version of the server was installed that no longer supported some codeset in the codesets array.

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**

None.

**Related Information**

**Routines**

- `rpc_ns_mgmt_read_codesets`
- `rpc_ns_mgmt_set_attribute`
- `rpc_rgy_get_codesets`
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.

Format
```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, provide `rpc_c_ns_syntax_default`. For more information, see the `name_syntax` section in "Frequently Used Routine Arguments" on page 2-20.

- `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 OSF DCE 1.1 (which includes z/OS DCE), this value must be `rpc_c_attr_codesets`.

- `attr_value` A pointer to the value to be stored in the attribute specified by the `attr_type` parameter. If `attr_type` is specified as `rpc_c_attr_codesets`, this parameter must point to an `rpc_codeset_mgmt_t` structure (see page 2-12). The pointer must be cast to a `void*` on input to this routine.

**Output**

- `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:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_s_ok</code></td>
<td>Success.</td>
</tr>
<tr>
<td><code>rpc_s_invalid_name_syntax</code></td>
<td>The name syntax specified in the call is not valid.</td>
</tr>
<tr>
<td><code>rpc_s_mgmt_bad_type</code></td>
<td>The attribute type specified in the call does not match that of the attribute to be added to the name service database.</td>
</tr>
<tr>
<td><code>rpc_s_no_memory</code></td>
<td>The routine was unable to allocate memory to encode the value.</td>
</tr>
<tr>
<td><code>rpc_s_name_service_unavailable</code></td>
<td>The routine was unable to communicate with the name service.</td>
</tr>
<tr>
<td><code>rpc_s_no_ns_permission</code></td>
<td>The routine’s caller does not have the proper permission for an NSI operation.</td>
</tr>
</tbody>
</table>
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 OSF DCE 1.1 (which includes z/OS DCE), 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). The rpc_ns_mgmt_set_attribute() routine 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).

Return Values
None.

Related Information
Routines
rpc_ns_mgmt_read_codesets     rpc_ns_mgmt_remove_attribute     rpc_rgy_get_codesets
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.

Format

```c
#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.
  
  Possible status code and its meaning:
  
  - `rpc_s_ok`: Successful completion.

Usage

The `rpc_ns_mgmt_set_exp_age` API changes the global expiration age that the application is using. 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.

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 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.

Note: With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.
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.

Permissions Required

None.

Return Values

None.

Related Information

Routines

rpc_ns_mgmt_handle_set_exp_age
rpc_ns_profile_delete

Purpose
Deletes a profile attribute.

Used by client, server, or management applications.

Format
```
#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 specified in the *RPC_DEFAULT_ENTRY_SYNTAX* environment variable, specify *rpc_c_ns_syntax_default*.

For more information, see the *name_syntax* section in "Frequently Used Routine Arguments" on page 2-20.

**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.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_entry_not_found**: Name service entry not found.
- **rpc_s_incomplete_name**: Incomplete name.
- **rpc_s_invalid_name_syntax**: Name syntax is not valid.
- **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.

Usage

The **rpc_ns_profile_delete** API deletes the profile attribute from the specified entry in the name service database (the *profile_name* parameter).

Neither the specified entry nor the entries included as members in each profile element are deleted.

Use this routine cautiously; deleting a profile may break a hierarchy of profiles.

**Note:** With z/OS, any thread calling this API cannot be canceled with the *pthread_cancel()* routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

Routines
rpc_ns_profile_elt_add         rpc_ns_profile_elt_remove
**rpc_ns_profile_elt_add**

**Purpose**
Adds an element to a profile. If necessary, creates the profile entry.

Used by client, server, or management applications.

**Format**

```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, specify `rpc_c_ns_syntax_default`.
  For more information, see the `name_syntax` section in ["Frequently Used Routine Arguments" on page 2-20](#).

- **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, specify `rpc_c_ns_syntax_default`.
  For more information, see the `name_syntax` section in ["Frequently Used Routine Arguments" on page 2-20](#).

- **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 (0-7) 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. 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.
rpc_ns_profile_elt_add

Output

status

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

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **rpc_s_class_version_mismatch**: RPC class version mismatch.
- **rpc_s_incomplete_name**: Incomplete name.
- **rpc_s_invalid_name_syntax**: Name syntax is not valid.
- **rpc_s_invalid_priority**: Profile element priority is not valid.
- **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.

Usage

The `rpc_ns_profile_elt_add` API 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. The application must have permission to create the entry. Otherwise, a management application with the necessary permissions creates the entry by calling the `rpc_ns_mgmt_entry_create` API before the application is run.

If an element with the specified member name and interface identifier is already in the profile, this routine updates the element’s priority and annotation string using the values specified in the `priority` and `annotation` parameters.

An application can add the entry in parameter `member_name` to a profile before it creates the entry itself.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

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

None.

Related Information

Routines

- `rpc_if_inq_id`
- `rpc_ns_mgmt_entry_create`
- `rpc_ns_profile_elt_remove`
**Purpose**
Creates an inquiry context for viewing the elements in a profile.

Used by client, server, or management applications.

**Format**
```c
#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 specified in the `RPC_DEFAULT_ENTRY_SYNTAX` environment variable, specify `rpc_c_ns_syntax_default`.

  For more information, see the `name_syntax` section in "Frequently Used Routine Arguments" on page 2-20.

- **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 list describes the valid inquiry types:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_c_profile_default_elt</code></td>
<td>Searches the profile for the default profile element, if any.</td>
</tr>
<tr>
<td></td>
<td>The <code>if_id</code>, <code>vers_option</code>, and <code>member_name</code> parameters are ignored.</td>
</tr>
<tr>
<td><code>rpc_c_profile_all_elts</code></td>
<td>Returns every element from the profile.</td>
</tr>
<tr>
<td></td>
<td>The <code>if_id</code>, <code>vers_option</code>, and <code>member_name</code> parameters are ignored.</td>
</tr>
<tr>
<td><code>rpc_c_profile_match_by_if</code></td>
<td>Searches the profile for those elements that contain the interface identifier specified by the <code>if_id</code> and <code>vers_option</code> values.</td>
</tr>
<tr>
<td></td>
<td>The <code>member_name</code> parameter is ignored.</td>
</tr>
<tr>
<td><code>rpc_c_profile_match_by_mbr</code></td>
<td>Searches the profile for those elements that contain the member name specified by the <code>member_name</code> parameter.</td>
</tr>
<tr>
<td></td>
<td>The <code>if_id</code> and <code>vers_option</code> parameters are ignored.</td>
</tr>
<tr>
<td><code>rpc_c_profile_match_by_both</code></td>
<td>Searches the profile for those elements that contain the interface identifier and member name specified by the <code>if_id</code>, <code>vers_option</code>, and <code>member_name</code> parameters.</td>
</tr>
</tbody>
</table>

Chapter 2. Remote Procedure Call 2-231
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 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 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 are the valid values for this parameter:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_vers_all</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_compatible</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_exact</td>
<td>Returns profile elements that offer the specified version of the specified interface UUID.</td>
</tr>
<tr>
<td>rpc_c_vers_major_only</td>
<td>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.</td>
</tr>
<tr>
<td>rpc_c_vers_upto</td>
<td>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.)</td>
</tr>
</tbody>
</table>

member_name_syntax
An integer value that specifies the syntax of parameter member_name in this routine and the syntax of parameter member_name in rpc_ns_profile_elt_inq_next.
To use the syntax specified in the RPC_DEFAULT_ENTRY_SYNTAX environment variable, specify rpc_c_ns_syntax_default.
For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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 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 rpc_ns_profile_elt_inq_next and rpc_ns_profile_elt_inq_done APIs.

status
Returns the status code from this routine, indicating whether the routine completed successfully or, if not, why not.
Possible status codes and their meanings:
- rpc_s_ok: Successful completion.
- rpc_s_incomplete_name: Incomplete name.
- rpc_s_invalid_inquiry_type: Inquiry type is not valid.
Usage
The `rpc_ns_profile_elt_inq_begin` API 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` API to delete the inquiry context.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required
None.

Return Values
None.

Related Information

Routines

- `rpc_if_inq_id`
- `rpc_ns_profile_elt_inq_done`
- `rpc_ns_profile_elt_inq_next`
- `rpc_ns_mgmt_handle_set_exp_age`
rpc_ns_profile_elt_inq_done

Purpose
Deletes the inquiry context for a profile.
Used by client, server, or management applications.

Format
#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.
Possible status codes and their meanings:
    rpc_s_ok Successful completion.
    rpc_s_invalid_ns_handle Name service handle is not valid.

Usage
The rpc_ns_profile_elt_inq_done API 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 API.

Note: With z/OS, any thread calling this API cannot be canceled with the pthread_cancel() routine. This avoids potential resource clean up problems with this routine.

Permissions Required
None.

Return Values
None.

Related Information
Routines

`rpc_ns_profile_elt_inq_begin`  `rpc_ns_profile_elt_inq_next`
rpc_ns_profile_elt_inq_next

**Purpose**

Returns one element at a time from a profile.

Used by client, server, or management applications.

**Format**

```
#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` API.

**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 `member_name_syntax` parameter in `rpc_ns_profile_elt_inq_begin`. Specify `NULL` to prevent the routine from returning this parameter. The application does not call the `rpc_string_free` API.
- **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. The application does not need to call the `rpc_string_free` API.
- **status**: Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

Possible status codes and their meanings:

- **rpc_s_ok**: Successful completion.
- **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**: Name service handle is not valid.
- **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.
Usage

The `rpc_ns_profile_elt_inq_next` API returns one element from the profile specified by the `profile_name` parameter in the `rpc_ns_profile_elt_inq_begin` API.

The selection criteria for the element returned are based on the `inquiry_type` parameter in the `rpc_ns_profile_elt_inq_begin` API. The `rpc_ns_profile_elt_inq_next` API 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 storage for the returned `member_name` (which points to a global name) and `annotation` strings. The application is responsible for calling the `rpc_string_free` API 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` API to delete the inquiry context.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

**Routines**

`rpc_ns_profile_elt_begin`  `rpc_ns_profile_elt_done`  `rpc_string_free`
rpc_ns_profile_elt_remove

Purpose
Removes an element from a profile.

Used by client, server, or management applications.

Format
#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, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

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.

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, specify rpc_c_ns_syntax_default.

For more information, see the name_syntax section in "Frequently Used Routine Arguments" on page 2-20.

member_name Specifies the name service entry name in the profile element to remove. 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.

Possible status codes and their meanings:

rpc_s_ok Successful completion.
rpc_s_entry_not_found Name service entry not found.
rpc_s_incomplete_name Incomplete name.
rpc_s_invalid_name_syntax Name syntax is not valid.
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.
Usage

The `rpc_ns_profile_elt_remove` API removes a profile element from the profile specified by `profile_name`. When `if_id` is not `NULL` the `member_name` parameter and the `if_id` parameter must match the corresponding profile elements 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 entry (member_name) referred to as a member in the profile element is not deleted.

Use this routine cautiously; removing elements from a profile may break a hierarchy of profiles.

**Note:** With z/OS, any thread calling this API cannot be canceled with the `pthread_cancel()` routine. This avoids potential resource clean up problems with this routine.

Permissions Required

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

Return Values

None.

Related Information

Routines

- `rpc_ns_profile_delete`
- `rpc_ns_profile_elt_add`
rpc_object_inq_type

Purpose
Returns the type of an object.
Used by server applications.

Format
```c
#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 specified in the `obj_uuid` parameter.
Specifying NULL here prevents the return of a type UUID. An application, by specifying NULL, can determine from the value returned in `status` whether parameter `obj_uuid` is registered. This determination occurs without the application specifying an output type UUID variable. If the `obj-uuid` is nil, a nil UUID is returned.

`status` Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.
Possible status codes and their meanings:
- `rpc_s_ok` Successful completion.
- `rpc_s_object_not_found` Object not found.
- `uuid_s_bad_version` Incorrect UUID version.

Usage
The `rpc_object_inq_type` API obtains the type UUID of an object.

If the object is registered with the RPC runtime using the `rpc_object_set_type` API, the registered type is returned.

Optionally, an application can maintain an object/type registration privately. If the application provides an object inquiry function (see "rpc_object_set_inq_fn" on page 2-242), the RPC runtime uses that function to determine an object’s type.

The following table summarizes how routine `rpc_object_inq_type` obtains the returned type UUID.

<table>
<thead>
<tr>
<th>Was object UUID registered (using <code>rpc_object_set_type</code>)?</th>
<th>Was an object inquiry function registered (using <code>rpc_object_set_inq_fn</code>)?</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>ignored</td>
<td>Return the object’s registered type UUID.</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>Return the type UUID returned from calling the inquiry function.</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>Return the nil UUID.</td>
</tr>
</tbody>
</table>
Return Values

None.

Related Information

Routines

rpc_object_set_inq_fn  

rpc_object_set_type
rpc_object_set_inq_fn

Purpose
Registers an object inquiry function.
Used by server applications.

Format
#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
rpc_object_inq_type and the RPC runtime finds that the specified object is not registered, the
runtime calls the registered inquiry function 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 (qc@1>4rpc_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 parameters from the
rpc_object_inq_type API.
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.
Possible status codes and their meanings:

rpc_s_ok                                      Successful completion.
rpc_s_object_not_found                      Object not found.

Usage

A server application calls the rpc_object_set_inq_fn API 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 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 calls the inquiry function for every remote procedure call it receives
if the object was not previously registered.

Note: Use this routine with caution. When the RPC runtime 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. The inquiry function must not unwind because of an exception. In general, the inquiry function should not call
back into the RPC runtime. You can call `rpc_object_set_type` or any of the `uuid_*` APIs. Failure to comply with these restrictions will result in undefined behavior.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_object_inq_type`
- `rpc_object_set_type`
**rpc_object_set_type**

**Purpose**
Registers the type of an object with the RPC runtime.

Used by server applications.

**Format**
```
#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.

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

  Possible status codes and their meanings:
  - rpc_s_ok: Successful completion.
  - rpc_s_already_registered: Object already registered.
  - rpc_s_invalid_object: Object is not valid.
  - uuid_s_bad_version: Incorrect UUID version.

**Usage**
The *rpc_object_set_type* API 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 description of routine "rpc_server_register_if" on page 2-262 for more information on registering a server interface.

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 non-nil 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 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**

None.

**Related Information**

**Routines**

| `rpc_object_set_inq_fn` | `rpc_server_register_if` |
**Purpose**
Frees the storage used by a vector and its protocol sequences.
Used by client or server applications.

**Format**
```c
#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.
    
    Possible status code and its meaning:
    - `rpc_s_ok`
      Successful completion.

**Usage**
The `rpc_protseq_vector_free` API frees the storage used to store a vector of protocol sequences. The freed storage 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**
None.

**Related Information**

- **Routines**
  - `rpc_network_inq_protseqs`
rpc_rgy_get_codesets

Purpose
Gets supported code sets information from the local host.
Used by client and server applications.

Format
#include <dce/rpc.h>

void rpc_rgy_get_codesets(
    rpc_codeset_mgmt_p_t *code_sets_array,
    error_status_t *status);

Parameters
Input
None.
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.
The possible status codes and their meanings are as follows:

dce_cs_c_cannot_open_file
Cannot open the code set registry file (path name may
be incorrect or the file's permissions may have
prevented access). If the code set registry is not in the
default location,
/usr/lib/nls/csr/code_set.registry.db
use the dce_cf_get_csrgy_filename routine to get the
correct path name.
dce_cs_c_cannot_read_file
Cannot read the code set registry file.
A read error occurred when reading the code set
registry (probably a problem in the underlying operating
system).
rpc_s_ok
Successful completion.
rpc_s_no_memory
No storage available.

Usage
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).
rpc_rgy_get_codesets

Server applications use the \texttt{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 \texttt{rpc\_ns\_mgmt\_set\_attribute()}. Client applications use the \texttt{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 \texttt{rpc\_cs\_eval\_with\_universal()} and \texttt{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 \texttt{rpc\_rgy\_get\_codesets()} in their user-written evaluation routines.

Applications that call \texttt{rpc\_rgy\_get\_codesets()} to allocate the \texttt{code\_sets\_array} should call \texttt{rpc\_ns\_mgmt\_free\_codesets()} to free this storage after the application has finished processing it.

\section*{Permissions Required}

None.

\section*{Return Values}

None.

\section*{Related Information}

\textbf{Routines}

\begin{itemize}
  \item \texttt{rpc\_ns\_mgmt\_free\_codesets()}
  \item \texttt{rpc\_ns\_mgmt\_remove\_attribute}
  \item \texttt{rpc\_ns\_mgmt\_set\_attribute}
  \item \texttt{rpc\_ns\_mgmt\_read\_codesets}
\end{itemize}
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.

Format
#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 maximum 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.

The possible status codes and their meanings are as follows:

dce_cs_c_cannot_open_file
Cannot open the code set registry file (path name may be incorrect or the file’s permissions may have prevented access). If the code set registry is not in the default location, /usr/lib/nls/csr/code_set_registry.db
use the dce_cf_get_csrgy_filename routine to get the correct path name.

dce_cs_c_cannot_read_file
Cannot read the code set registry file.
A read error occurred when reading the code set registry (probably a problem in the underlying operating system).

dce_cs_c_notfound
No local code set name exists in the code set registry (implies the code set is not supported in your environment).

dce_cs_c_unknown
The requested value, for example local code set name or registry code set value, was not found in the code set registry.

rpc_s_ok
Successful completion.

Usage
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 maximum number of bytes that the code set uses to encode one character.
rpc_rgy_get_max_bytes

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 additional buffers are necessary or whether the conversion can be done “in place”.

Client and server applications that use the DCE RPC buffer sizing routines cs_byte_net_size() and cs_byte_local_size() do not need to call this routine explicitly because these DCE RPC stub support routines call it on their behalf. 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

None.

Return Values

None.

Related Information

Routines

cs_byte_local_size  dce_cs_loc_to_rgy  rpc_ns_mgmt_read_code_sets
cs_byte_net_size    dce_cs_rgy_to_loc  rpc_rgy_get_code_sets
rpc_server_inq_bindings

Purpose
Returns binding handles for communication with a server.
Used by server applications.

Format
#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.

Possible status codes and their meanings:
rpc_s_ok Successful completion.
rpc_s_no_bindings No bindings.
rpc_s_no_memory No storage available.
rpc_s_invalid_arg The argument is not valid.

Usage
The rpc_server_inq_bindings API 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 above routines the server application called. See "Binding Handle" on page 2-9 for 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 parameter binding_vector.

The server is responsible for calling the rpc_binding_vector_free routine to deallocate the storage used by the vector.

Return Values
None.
rpc_server_inq_bindings

Related Information

Routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_binding_vector_free</td>
<td>rpc_ns_binding_export</td>
<td>rpc_server_use_protseq</td>
</tr>
<tr>
<td>rpc_ep_register</td>
<td>rpc_server_use_all_protseqs</td>
<td>rpc_server_use_protseq_ep</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace</td>
<td>rpc_server_use_all_protseqs_if</td>
<td>rpc_server_use_protseq_if</td>
</tr>
</tbody>
</table>
rpc_server_inq_if

Purpose
Returns the manager entry point vector registered for an interface.
Used by server applications.

Format
#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.
Possible status codes and their meanings:

- rpc_s_ok: Successful completion.
- rpc_s_unknown_if: Unknown interface.
- rpc_s_unknown_mgr_type: Unknown manager type.
- uuid_s_bad_version: Incorrect UUID version.

Usage
The rpc_server_inq_if API determines the manager EPV for a registered interface and type UUID of the manager.

Return Values
None.

Related Information
rpc_server_inq_if

Routines

rpc_server_register_if
rpc_server_listen

Purpose
Tells the RPC runtime to listen for remote procedure calls.

Used by server applications.

Format
```
#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 running remote procedure calls.

Use the value `rpc_c_listens_max_calls_default` to specify the maximum default value. If you specify `rpc_c_listens_max_calls_default`, z/OS sets the maximum number of concurrent calls at 50% of the process thread limit for the process. You can change the process thread limit by setting both the `MAXTHREADS` and `MAXTHREADTASKS` parameters in the BPXPRMxx parmlib member of the SYS1.PARMLIB data set to the required value. The parameters contained in BPXPRMxx control the z/OS UNIX environment, the hierarchical file system, and the sockets file systems. The system uses these parameter values to initialize the kernel. For more information on the above, refer to `z/OS C/C++ Run-Time Library Reference`, SA22-7821.

To calculate the number of initial executor threads, use the following formulae:

<table>
<thead>
<tr>
<th>Number of Concurrent Calls (x)</th>
<th>Number of Initial Threads (y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ≤ 10</td>
<td>y = x</td>
</tr>
<tr>
<td>10 &lt; x ≤ 50</td>
<td>y = ((x - 10) × 0.25) + 10</td>
</tr>
<tr>
<td>x &gt; 50</td>
<td>y = ((x - 50) × 0.20) + 10</td>
</tr>
</tbody>
</table>

Also, the five `rpc_server_use_*protseq` APIs 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.

Possible status codes and their meanings:
- `rpc_s_ok` Successful completion.
- `rpc_s_already_listening` Server already listening.
- `rpc_s_no_protseqs_registered` No protocol sequences registered.
- `rpc_s_cannot_create_listener` Cannot create listener.
- `rpc_s_cannot_cancel_listener` Cannot cancel listener.
- `rpc_s_max_calls_too_small` Max calls value too small.
- `rpc_s_no_memory` No storage available.
- `rpc_s_socket_failure` Server `socket` failure.
- `rpc_s_select_error` Server selection error.
rpc_server_listen

Usage

The rpc_server_listen API makes a server listen for remote procedure calls. RPC allows a server to simultaneously process multiple calls. Each remote procedure call runs in a call execution thread, and when the server application calls rpc_server_listen, RPC runtime will create and maintain a number of executor threads.

All RPC executor threads are classified as z/OS heavy-weight threads, and each thread has its own task control block (TCB). When an incoming call is received, the call is dispatched to one of the executor threads, which then calls the corresponding operation manager. When the call is completed, the executor thread returns to the internal RPC runtime executor pool; the same executor thread TCB can execute calls from multiple clients. RPC executor threads are destroyed when the application issues rpc_server_stop_listening and all TCBs associated with these threads are detached.

The implementation of the RPC architecture determines whether it reuses call execution threads for the running of subsequent remote procedure calls or whether it creates a new thread each time it runs a subsequent remote procedure call.

Note that to improve scalability and increase the performance of the RPC runtime, z/OS dynamically allocates and deallocates system resources such as the number of executor threads required by the server application. For more information on the RPC dynamic thread pool, refer to the [z/OS DCE Application Development Guide: Core Components](https://www.zos.com/zos/bksamples/zosdceadm.html).

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* APIs advises the RPC runtime about its request of network resources.
- Enough call threads must be available to run 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 because each runs in a separate thread.

If the server receives more remote procedure calls than it can run (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 running. 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 is not completed within the cancel timeout limits.

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_mgmgt_stop_server_listening.
- A client is allowed to, and makes, a remote rpc_mgmgt_stop_server_listening call to the server.
- The communication function has an unrecoverable error.

On receiving a request to stop listening, the RPC runtime stops accepting new remote procedure calls for all registered interfaces. Calls that are running 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, see the RPC module in the [z/OS DCE Application Development Guide: Core Components](https://www.zos.com/zos/bksamples/zosdceadm.html) This module also contains information about canceled threads.

Return Values

None.
Related Information

Routines

- `rpc_mgmt_stop_server_listening`
- `rpc_server_register_if`
- `rpc_server_use_all_protseqs`
- `rpc_server_use_all_protseqs_if`
- `rpc_server_use_protseq_ep`
- `rpc_server_use_protseq`
- `rpc_server_use_protseq_if`
Purpose
Registers authentication information with the RPC runtime.
Used by server applications.

Format
#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 name and its syntax are 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 supported authentication services are:

    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_retrieval_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 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 type in the key parameter.

Note: The key_type parameter specifies a Kerberos encryption key type. Because DCE supports only Data Encryption Standard (DES) encryption, you can ignore this parameter.

If the key acquisition routine, when called from the rpc_server_register_auth_info API, 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 a parameter to pass to the `get_key_fn` key acquisition routine, if specified. See [Format](#) on page 2-258 for the description of the `get_key_fn` parameter.

Specify **NULL** for `arg` to use the default key table file, `/krb5/v5srvtab`. The calling server must be run as the root to access this file.

If `arg` is a key-table file name, the file must have been created with the `ktadd` command. If the specified key-table file resides in `/krb5`, you can specify only the file name. If the file does not reside in `/krb5`, you must specify the full path name. You can add **FILE:** as a prefix to the absolute name of the file for compatibility with other DCE implementations, but this is not a requirement with z/OS DCE.

**Output**

**status**

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

Possible status codes and their meanings:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rpc_s_ok</strong></td>
<td>Successful completion.</td>
</tr>
<tr>
<td><strong>rpc_s_unknown_authn_service</strong></td>
<td>Unknown authentication service.</td>
</tr>
<tr>
<td><strong>rpc_s_key_func_not_allowed</strong></td>
<td><code>auth_svc</code> is <strong>rpc_c_authn_default</strong> and a non-null value was specified for <code>get_key_fn</code> parameter.</td>
</tr>
</tbody>
</table>

**Usage**

The `rpc_server_register_auth_info` API registers an authentication service to use for authenticating remote procedure calls to a particular server principal. 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 `rpc_binding_set_auth_info` or `rpc_if_register_auth_info`) 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>
<thead>
<tr>
<th>authn_svc</th>
<th>get_key_fn</th>
<th>arg</th>
<th>Runtime Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rpc_c_authn_default</strong></td>
<td><strong>NULL</strong></td>
<td><strong>NULL</strong></td>
<td>Use the default method of encryption key acquisition from the default key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_default</strong></td>
<td><strong>NULL</strong></td>
<td>non-NULL</td>
<td>Use the default method of encryption key acquisition from the default key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_default</strong></td>
<td>non-NULL</td>
<td>ignored</td>
<td>Error returned.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_none</strong></td>
<td>ignored</td>
<td>ignored</td>
<td>No authentication performed.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td><strong>NULL</strong></td>
<td><strong>NULL</strong></td>
<td>Use the default method of encryption key acquisition from the default key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td><strong>NULL</strong></td>
<td>non-NULL</td>
<td>Use the default method of encryption key acquisition from the specified key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td>non-NULL</td>
<td><strong>NULL</strong></td>
<td>Use the specified encryption key acquisition routine to obtain keys from the default key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td>non-NULL</td>
<td>non-NULL</td>
<td>Use the specified encryption key acquisition routine to obtain keys from the specified key table.</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_public</strong></td>
<td>ignored</td>
<td>ignored</td>
<td>(Reserved for future use.)</td>
</tr>
</tbody>
</table>
Return Values

None.

Examples

The following example shows how you can use the `rpc_server_register_auth_info` API to obtain keys from a specified key acquisition table using a specified key acquisition routine:

```c
#include <dce/rpc.h>
#include <dce/keymgmt.h>
#include <dce/rpcbase.h>
#include <dce/passwd.h>
#include <locale.h>
#include <krb5/krb5.h>

void get_key_func (void *arg, unsigned_char_t *server_princ_name,
                   unsigned32 key_type, unsigned32 key_ver,
                   void **key, unsigned32 *status )
{
    krb5_keyblock *kamkey;

    void get_key_func (void *arg,
                       unsigned_char_t *server_princ_name,
                       unsigned32 key_type,
                       unsigned32 key_ver,
                       void **key,
                       unsigned32 *status )
    {
        sec_passwd_rec_t *retrieved_key;
        kamkey = (krb5_keyblock *) malloc(sizeof(krb5_keyblock));
        *status = rpc_s_ok;
        printf("\n\n\n======================\n");
        printf("Inside get_key_func: server_princ_name: %s\n",server_princ_name);
        printf("Inside get_key_func: arg is: %s\n", (char *) arg);
        printf("Inside get_key_func: calling sec_key_mgmt_get_key\n");
        printf("\n\n\n" );
        sec_key_mgmt_get_key(rpc_c_authn_dce_private,
                             arg,
                             server_princ_name,
                             key_ver,
                             &retrieved_key,
                             status);
        printf("Get_key_func: sec_key_mgmt_get_key status: %d\n", *status);
        kamkey->contents = (char *)malloc(8);
        memcpy(kamkey->contents,retrieved_key->key.tagged_union.des_key,8);
        kamkey->length = 8;
        kamkey->keytype = sec_passwd_des;
        *key = kamkey;
        if (*key != NULL)
        {
            printf("\n\n\n\n======================\n");
            printf("key does not equal NULL\n");
            printf("\n\n\n" );
        }
        printf("\n\n\n======================\n");
        printf("Inside get_key_func: leaving sec_key_mgmt_get_key\n");
        printf("\n\n\n" );
    }
```

2-260  Application Development Reference Volumes 1 and 2
int main(int argc, char *argv[]) {
    error_status_t st;
    setlocale(LC_ALL,"");
    rpc_server_register_auth_info("my_svr",
        rpc_c_authn_dce_secret,
        &get_key_func,
        "FILE:/krb5/v5srvtab",
        &st);
    printf("Status from register auth info was >=d<w\n",st);
}

Related Information

Routines

rpc_binding_set_auth_info     rpc_if_register_auth_info
**rpc_server_register_if**

**Purpose**
Registers an interface with the RPC runtime.

Used by server applications.

**Format**
```c
#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 EPV of the manager routine. 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.

Possible status codes and their meanings:
- **rpc_s_ok**
  Successful completion.
- **rpc_s_type_already_registered**
  An interface with the given type of UUID is already registered.
- **rpc_s_no_memory**
  No storage available.
- **rpc_s_no_mepv**
  No manager EPV available.
- **rpc_s_unknown_ifspec_vers**
  Version of interface is not known.

**Usage**
The `rpc_server_register_if` API 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 with 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 runs 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. When a non-nil type UUID is specified, the server must also call the `rpc_object_set_type` API to register objects of this non-nil type.
A server that only offers a single manager for an interface calls `rpc_server_register_if` once for that interface. 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: 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 Calling Manager Routines:** The RPC runtime sends 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 calling manager routines.

<table>
<thead>
<tr>
<th>Object UUID of Call</th>
<th>Has Server Set Type of Object UUID?</th>
<th>Has Server Registered Type for Manager EPV?</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>Not applicable</td>
<td>yes</td>
<td>Uses the manager with the nil type UUID.</td>
</tr>
<tr>
<td>nil</td>
<td>Not applicable</td>
<td>no</td>
<td>Error (<code>rpc_s_unknown_mgr_type</code>). Reject the remote procedure call.</td>
</tr>
<tr>
<td>non-nil</td>
<td>yes</td>
<td>yes</td>
<td>Uses the manager with the same type UUID.</td>
</tr>
<tr>
<td>non-nil</td>
<td>no</td>
<td>ignored</td>
<td>Uses the manager with the nil type UUID. If no manager with the nil type UUID was registered, error (<code>rpc_s_unknown_mgr_type</code>). Rejects the remote procedure call.</td>
</tr>
<tr>
<td>non-nil</td>
<td>yes</td>
<td>no</td>
<td>Error (<code>rpc_s_unknown_mgr_type</code>). Reject the remote procedure call.</td>
</tr>
</tbody>
</table>

Notes:

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 assigned the nil type UUID. You cannot specify a nil object UUID in `rpc_object_set_type`.

For more information about registering server interfaces and calling manager routines, see the RPC module of the [z/OS DCE Application Development Guide: Core Components](Z/OS DCE Application Development Guide: Core Components).

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_binding_set_object`
- `rpc_ep_register`
- `rpc_ep_register_no_replace`
- `rpc_ns_binding_export`
- `rpc_object_set_type`
- `rpc_server_unregister_if`
Purpose
Removes an interface from the RPC runtime.
Used by server applications.

Format
#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 API.
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. All manager EPVs registered with a non-nil 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.
Possible status codes and their meanings:

rpc_s_ok
Successful completion.

rpc_s_unknown_if
Unknown interface.

rpc_s_unknown_mgr_type
Unknown manager type.

Usage

The rpc_server_unregister_if API 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. Calls that are running (on that interface) are allowed to complete.

The following table summarizes the actions of this routine.

<table>
<thead>
<tr>
<th>if_handle</th>
<th>mgr_type_uuid</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-NULL</td>
<td>non-NULL</td>
<td>Removes the manager EPV associated with the specified parameters.</td>
</tr>
<tr>
<td>non-NULL</td>
<td>NULL</td>
<td>Removes all manager EPVs associated with parameter if_handle.</td>
</tr>
</tbody>
</table>
Table 2-17 (Page 2 of 2). Rules for Removing an Interface

<table>
<thead>
<tr>
<th>if_handle</th>
<th>mgr_type_uuid</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>non-NULL</td>
<td>Removes all manager EPVs associated with parameter mgr_type_uuid.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>Removes all manager EPVs.</td>
</tr>
</tbody>
</table>

**Note:** When both of the parameters if_handle and mgr_type_uuid are given the value NULL, this call prevents the server from receiving any new remote procedure calls because all the manager EPVs for all interfaces have been removed.

**Return Values**

None.

**Related Information**

**Routines**

rpc_server_register_if
rpc_server_use_all_protseqs

Purpose
Registers all supported protocol sequences with the RPC runtime.

Used by server applications.

Format
#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: Any number you specify is replaced by the default value.

Also, routine rpc_server_listen limits (according to its max_calls_exec parameter) the amount of concurrent remote procedure call processing. See "rpc_server_listen" on page 2-255 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.

Possible status codes and their meanings:

- rpc_s_ok: Successful completion.
- rpc_s_cant_create_socket: Cannot create socket.
- rpc_s_max_descs_exceeded: Exceeded maximum number of descriptors for network sockets that can be stored internally.
- rpc_s_no_protseqs: No supported protocol sequences.
- rpc_s_no_memory: No storage available.

Usage
The rpc_server_use_all_protseqs API 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 explanation of "String Binding" on page 2-17.

**Return Values**

None.

**Related Information**

**Routines**

- rpc.binding_from_string_binding
- rpc.binding_to_string_binding
- rpc.binding.vector.free
- rpc.ep.register
- rpc.ep.register_no_replace
- rpc_ns.binding.export
- rpc.server.inq_bindings
- rpc.server.listen
- rpc.server.register.if
- rpc.server.listen
- rpc.server.use.all.protseqs.if
- rpc.server.use.protseq
- rpc.server.use.protseq.ep
- rpc.server.use.protseq.if
rpc_server_use_all_protseqs_if

Purpose
Registers protocol sequences and endpoint address information with the RPC runtime.
Used by server applications.

Format
#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 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: Any number you specify is replaced by the default value.
Also, routine rpc_server_listen limits (according to its max_calls_exec parameter) the amount of concurrent remote procedure call processing. See "rpc_server_listen" on page 2-255 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 has 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.
Possible status codes and their meanings:

rpc_s_ok
Successful completion.

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
Endpoint format is not valid.

rpc_s_max_descs_exceeded
Exceeded maximum number of descriptors for network sockets that can be stored internally.

rpc_s_no_protseqs
No supported protocol sequences.

rpc_s_no_memory
No storage available.
Usage

The `rpc_server_use_all_protseqs_if` API 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" on page 2-266 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 explanation of "String Binding" on page 2-17.

Return Values

None.

Related Information

Routines

- `rpc_binding_vector_free`
- `rpc_ep_register`
- `rpc_ep_register_no_replace`
- `rpc_ns_binding_export`
- `rpc_server_inqBindings`
- `rpc_server_listen`
- `rpc_server_register_if`
- `rpc_server_use_protseq`
- `rpc_server_use_protseq_ep`
- `rpc_server_use_all_protseqs`
Purpose
Registers the specified protocol sequence for receiving remote procedure calls.
Used by server applications.

Format
```c
#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. See [Table 2-5 on page 2-16](#) for a list of string identifiers.

- **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:** Any number you specify is replaced by the default value.

  Also, routine `rpc_server_listen` limits (according to its `max_calls_exec` parameter) the amount of concurrent remote procedure call processing. See [“rpc_server_listen” on page 2-255](#) 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.
  Possible status codes and their meanings:
  - `rpc_s_ok` Successful completion.
  - `rpc_s_cant_create_socket` Cannot create socket.
  - `rpc_s_invalid_rpc_protseq` Protocol sequence is not valid.
  - `rpc_s_max_descs_exceeded` Exceeded maximum number of descriptors for network sockets that can be stored internally.
  - `rpc_s_protseq_not_supported` Protocol sequence not supported on this host.
  - `rpc_s_cant_bind_socket` Cannot bind to socket.
  - `rpc_s_invalid_arg` The argument is not valid.

Usage

The `rpc_server_use_protseq` API 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.

To register all protocol sequences, a server calls the `rpc_server_use_all_protseqs` API.

The explanation of "rpc_server_use_all_protseqs" on page 2-266 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 explanation of "String Binding" on page 2-17.

**Return Values**

None.

**Related Information**

**Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_binding_vector_free</code></td>
<td><code>rpc_ns_binding_export</code></td>
<td><code>rpc_server_use_all_protseqs</code></td>
</tr>
<tr>
<td><code>rpc_ep_register</code></td>
<td><code>rpc_server_inqBindings</code></td>
<td><code>rpc_server_use_all_protseqs_if</code></td>
</tr>
<tr>
<td><code>rpc_ep_register_no_replace</code></td>
<td><code>rpc_server_list</code></td>
<td><code>rpc_server_use_protseq_ep</code></td>
</tr>
<tr>
<td><code>rpc_network_is_protseq_valid</code></td>
<td><code>rpc_server_register_if</code></td>
<td><code>rpc_server_use_protseq_if</code></td>
</tr>
</tbody>
</table>
rpc_server_use_protseq_ep

**Purpose**
Registers a specified protocol sequence combined with the specified endpoint for receiving remote procedure calls.

Used by server applications.

**Format**
```
#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. See Table 2-5 on page 2-16 for a list of acceptable values.

- **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:** Any number you specify is replaced by the default value.
  Also, routine `rpc_server_listen` limits (according to its `max_calls_exec` parameter) the amount of concurrent remote procedure call processing. See the description section of "rpc_server_listen" on page 2-255 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.
  Possible status codes and their meanings:
  - `rpc_s_ok` Successful completion.
  - `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` Endpoint format is not valid.
  - `rpc_s_invalid_net_addr` Network address not valid.
  - `rpc_s_invalid_rpc_protseq` Protocol sequence is not valid.
  - `rpc_s_max_descs_exceeded` Exceeded maximum number of descriptors for network sockets that can be stored internally.
  - `rpc_s_protseq_not_supported` Protocol sequence not supported on this host.
  - `rpc_s_invalid_arg` The argument is not valid.
  - `rpc_s_no_memory` No storage available.
Usage

The **rpc_server_use_protseq_ep** API 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" on page 2-266] 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 explanation of ["String Binding" on page 2-17].

Return Values

None.

Related Information

Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_binding_vector_free</td>
<td>rpc_server_inqbindings</td>
<td>rpc_server_use_all_protseqs_if</td>
</tr>
<tr>
<td>rpc_ep_register</td>
<td>rpc_server_listen</td>
<td>rpc_server_use_protseq</td>
</tr>
<tr>
<td>rpc_ep_register_no_replace</td>
<td>rpc_server_register_if</td>
<td>rpc_server_use_protseq_ep</td>
</tr>
<tr>
<td>rpc_ns_binding_export</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**rpc_server_use_protseq_if**

**Purpose**
Registers a specified protocol sequence combined with the endpoints in the interface specification for receiving remote procedure calls.

Used by server applications.

**Format**
```
#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. See [Table 2-5 on page 2-16](#) for a list of string identifiers.

- **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:** Any number you specify is replaced by the default value.

  Also, routine `rpc_server_listen` limits (according to its `max_calls_exec` parameter) the amount of concurrent remote procedure call processing. See the description section of "rpc_server_listen" on page 2-255 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.

  Possible status codes and their meanings:

  - `rpc_s_ok` : Successful completion.
  - `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` : Endpoint format is not valid.
  - `rpc_s_invalid_rpc_protseq` : Protocol sequence is not valid.
  - `rpc_s_max_descs_exceeded` : Exceeded maximum number of descriptors for network sockets that can be stored internally.
  - `rpc_s_protseq_not_supported` : Protocol sequence not supported on this host.
  - `rpc_s_invalid_arg` : The argument is not valid.
  - `rpc_s_no_memory` : No storage available.
Usage

The `rpc_server_use_protseq_if` API 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` API.

The explanation of "rpc_server_use_all_protseqs" on page 2-266 contains a list of the routines a server typically calls after calling this routine. However, a server that uses only routine `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 explanation of "String Binding" on page 2-17.

Return Values

None.

Related Information

Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_binding_vector_free</code></td>
<td><code>rpc_server_inq_bindings</code></td>
<td><code>rpc_server_use_all_protseqs_if</code></td>
</tr>
<tr>
<td><code>rpc_ep_register</code></td>
<td><code>rpc_server_listen</code></td>
<td><code>rpc_server_use_protseq</code></td>
</tr>
<tr>
<td><code>rpc_ep_register_no_replace</code></td>
<td><code>rpc_server_register_if</code></td>
<td><code>rpc_server_use_protseq_ep</code></td>
</tr>
<tr>
<td><code>rpc_ns_binding_export</code></td>
<td><code>rpc_server_use_all_protseqs</code></td>
<td></td>
</tr>
</tbody>
</table>
###.rpc_sm_allocate

**Purpose**

Allocates storage within the RPC stub memory management scheme.

Used by server or possibly by client applications.

**Format**

```c
#include <dce/rpc.h>
idl_void_p_t rpc_sm_allocate(
    unsigned long size,
    unsigned32 *status);
```

**Parameters**

**Input**

- `size` Specifies, in bytes, the size of storage 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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_no_memory` No storage available.
- `rpc_s_posix_error` The number of POSIX errors detected by the runtime exceeded the limit.

**Usage**

The `rpc_sm_allocate` API allocates storage within the RPC stub memory management scheme. Before a call can be made 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 storage allocated by `rpc_sm_allocate`. The application can free such storage before returning to the calling stub by calling `rpc_sm_free`.

When the application establishes the memory management environment, it must free any storage 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 storage 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 storage.

**Related Information**

2-276  Application Development Reference Volumes 1 and 2
Routines

rpc_sm_free  rpc_sm_disable.allocate  rpc_sm_set_thread_handle
rpc_sm_enable.allocate  rpc_sm_get_thread_handle
rpc_sm_client_free

.rpc_sm_client_free

**Purpose**
Frees storage returned from a client stub.
Used by client applications.

**Format**
```c
#include <dce/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 storage 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.

  Possible status codes and their meanings:
  - `rpc_s_ok` Successful completion.
  - `rpc_s_no_memory` No storage available.
  - `rpc_s_posix_error` The number of POSIX errors detected by the runtime exceeded the limit.

**Usage**
The `rpc_sm_client_free` API releases storage 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. Application 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 storage returned by an RPC call without knowledge of the
storage management environment from which it was called.

**Return Values**
None.

**Related Information**

**Routines**
- `rpc_sm_free`
- `rpc_sm_set_client_alloc_free`
- `rpc_sm_swap_client_alloc_free`
- `rpc_sm_get_thread_handle`
- `rpc_sm_set_thread_handle`
**rpc_sm_destroy_client_context**

**Purpose**
Reclaims the client storage resources for a context handle, and sets the context handle to null.

Used by client applications.

**Format**
```
#include <dce/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.

Possible status codes and their meanings:

- **rpc_s_ok** Successful completion.
- **rpc_s_posix_error** The number of POSIX errors detected by the runtime exceeded the limit.

**Usage**
The `rpc_sm_destroy_client_context` API is used by client application 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 storage resources, it also sets the context handle to null.

**Return Values**
None.
rpc_sm_disable_allocate

Purpose
Releases resources and allocated storage within the stub memory management scheme.
Used by client applications.

Format
#include <dce/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.
Possible status codes and their meanings:
  rpc_s_ok
  rpc_sposix_error

Usage
The rpc_sm_disable_allocate API releases all resources acquired by a call to rpc_sm_enable_allocate, and any storage
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 APIs must be used in matching pairs.

Related Information
Routines
rpc_sm_allocate           rpc_sm_enable_allocate
rpc_sm_enable_allocate

Purpose
Establishes the stub memory management environment.
Used by client applications.

Format
#include <dce/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.
Possible status codes and their meanings:
- rpc_s_ok: Successful completion.
- rpc_s_posix_error: The number of POSIX errors detected by the runtime exceeded the limit.

Usage
The rpc_sm_enable_allocate API establishes a stub memory management environment in cases where one is not established
by the stub itself. For server manager code called from the stub, the stub memory management environment is normally
established by the stub itself. A stub memory management environment must be established before any calls are made to
rpc_sm_allocate. 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
"rpc_sm_get_thread_handle" on page 2-283 and "rpc_sm_set_thread_handle" on page 2-286.

Return Values
None.

Related Information
Routines
rpc_sm_allocate                   rpc_sm_disable_allocate
rpc_sm_free

Purpose
Frees storage allocated by the rpc_sm_allocate API.
Used by server or possibly by client applications.

Format
#include <dce/rpc.h>
void rpc_sm_free(idl_void_p_t node_to_free,
                 unsigned32 *status);

Parameters

Input
node_to_free Specifies a pointer to storage 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.
Possible status codes and their meanings:
rpc_s_ok Successful completion.
rpc_s_posix_error The number of POSIX errors detected by the runtime exceeded the limit.

Usage
The rpc_sm_free API releases storage allocated by rpc_sm_allocate.
When the stub allocates storage within the stub memory management environment, manager code called from the stub can
also use rpc_sm_free to release storage allocated by the stub.
The thread calling rpc_sm_free must have the same thread handle as the thread that allocated the storage 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.

Related Information
Routines
rpc_sm_allocate rpc_sm_get_thread_handle rpc_sm_set_thread_handle
rpc_sm_get_thread_handle

Purpose
Gets a thread handle for the stub memory management environment.

Used by server or possibly by client applications.

Format

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

Possible status codes and their meanings:

- `rpc_s_ok`: Successful completion.
- `rpc_s_posix_error`: The number of POSIX errors detected by the runtime exceeded the limit.

Usage

The `rpc_sm_get_thread_handle` API gets a thread handle for the current stub memory management environment. A thread that is managing storage within the stub memory management 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 its handle can use the same memory management environment.

When multiple threads call `rpc_sm_allocate` and `rpc_sm_free` to manage the same storage, 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 storage. 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 can 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.

Related Information

**Routines**

- `rpc_sm_allocate`
- `rpc_sm_free`
- `rpc_sm_set_thread_handle`
rpc_sm_set_client_alloc_free

Purpose
Sets the storage allocation and freeing mechanisms used by the client stubs.

Used by client applications.

Format
```
#include <dce/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 storage allocator routine.
- `p_free` Specifies a storage free routine. This routine is used to free storage 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.

Possible status codes and their meanings:

- `rpc_s_ok` Successful completion.
- `rpc_s_no_memory` No storage available.
- `rpc_s_posix_error` The number of POSIX errors detected by the runtime exceeded the limit.
- `rpc_s_invalid_arg` The argument is not valid. The storage allocator and free routines are not specified.

Usage

The `rpc_sm_set_client_alloc_free` API overrides the default routines that the client stub uses to manage storage.

**Note:** The default memory management routines are **ISO malloc** and **ISO free** except when the remote call occurs within manager code, then default memory management routines are **rpc_sm_allocate** and **rpc_sm_free**.

Return Values

None.

Related Information
Routines

rpc_sm_allocate        rpc_sm_free
rpc_sm_set_thread_handle

Purpose
Sets a thread handle for the stub memory management environment.
Used by server or possibly by client applications.

Format
#include <dce/rpc.h>
void rpc_sm_set_thread_handle(
    rpc_sm_thread_handle_t id,
    unsigned32 *status );

Parameters

Input
idi
Specifies a thread handle returned by a call to 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.
Possible status codes and their meanings:
  rpc_s_ok
       Successful completion.
  rpc_s_no_memory
       No storage available.
  rpc_s_posix_error
       The number of POSIX errors detected by the runtime exceeded the limit.

Usage
The rpc_sm_set_thread_handle API sets a thread handle for managing storage within the stub memory management
environment. A thread that is managing storage within the stub memory management 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 can use the same memory management environment.

When multiple threads call rpc_sm_allocate and rpc_sm_free to manage the same storage, they must share the same thread
handle. The thread that established the stub memory management calls rpc_sm_get_thread_handle to get a thread handle
before spawning new threads that will manage the same storage. 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 can also use rpc_sm_get_thread_handle and rpc_sm_set_thread_handle to save and
restore its memory management environment.

Return Values
None.

Related Information
Routines

rpc_sm_allocate  rpc_sm_free  rpc_sm_get_thread_handle
Purpose
Exchanges the current storage allocation and freeing mechanism used by the client stubs with one specified by the client.

Used by client applications.

Format

```c
#include <dce/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_old_allocate) (unsigned long size),
    void (**p_old_free) (idl_void_p_t ptr),
    unsigned32 *status);
```

Parameters

**Input**

- `p_allocate`: Specifies a new storage allocation routine.
- `p_free`: Specifies a new storage free routine.

**Output**

- `p_p_old_allocate`: Returns the storage allocation routine in use before the call to this routine.
- `p_p_old_free`: Returns the storage 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.

Possible status codes and their meanings:

- `rpc_s_ok`: Successful completion.
- `rpc_s_no_memory`: No storage available.
- `rpc_sposix_error`: The number of POSIX errors detected by the runtime exceeded the limit.
- `rpc_s_invalid_arg`: The argument is not valid. The storage allocator and free routines are not specified.

Usage

The `rpc_sm_swap_client_alloc_free` API exchanges the current allocate and free mechanisms used by the client stubs for routines specified by the caller.

Return Values

None.

Related Information
### Routines

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_sm_allocate</code></td>
</tr>
<tr>
<td><code>rpc_sm_free</code></td>
</tr>
<tr>
<td><code>rpc_sm_swap_client_alloc_free</code></td>
</tr>
</tbody>
</table>
### rpc_ss_allocate

**Purpose**
Allocates memory within the RPC stub memory management scheme.

Used by server or possibly by client applications.

**Format**

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

idl_void_p_t rpc_ss_allocate (idl_size_t size);
```

**Parameters**

**Input**

- `size`
  
  Specifies, in bytes, the size of storage to be allocated.

  **Note:** In ANSI standard C environments, such as C/C++ or ILE C/400, `idl_void_p_t` is defined as `void *`, and in other environments is defined as `char *`.

**Usage**

Usually, the `rpc_ss_allocate` API is used in the manager code that is called from a server stub. Storage 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 storage was allocated. If you want to release storage 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 storage 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 `rpc_ss_allocate` 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. The generated server stub then sets up the necessary environment.

**Note:** Storage allocated by allocators other than `rpc_ss_allocate` is not released when the operation running on the server side is complete.

If you want to use `rpc_ss_allocate` outside the code called from a server stub, you must create an environment for it by first calling `rpc_ss_enable_allocate`.

See “Rules for Using Memory Management Routines” in the Interface Definition Language chapter in [Z/OS DCE Application Development Guide: Core Components](#).

**Return Value**

A pointer to the allocated storage.

**Exceptions**

The following exceptions may be raised. To capture the exceptions, use *TRY* and *CATCH* around the routines.

- `rpc_x_no_memory` No storage available.
- `rpc_x_ss_posix_error` A POSIX error is detected.
### Related Information

#### Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_ss_free</code></td>
<td><code>rpc_ss_disable_allocate</code></td>
<td><code>rpc_ss_set_thread_handle</code></td>
</tr>
<tr>
<td><code>rpc_ss_enable_allocate</code></td>
<td><code>rpc_ss_get_thread_handle</code></td>
<td></td>
</tr>
</tbody>
</table>
rpc_ss_bind_authn_client

Purpose
Authenticates a client's identity to a server from a client stub.

Used by client applications.

Format
#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.

Possible status codes and their meanings include:

error_status_ok Success.
rpc_s_no_more_bindings Directs the client stub not to look for another server binding.

Usage
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 rpc_binding_set_auth_info 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. See the z/OS DCE Application Development Guide: Core Components for more information on using the binding_callout ACF attribute.
Return Values
None.

Related Information

Routines

rpc_binding_set_auth_info        rpc_ep_resolve_binding        rpc_mgmt_inq_server_princ_name
**rpc_ss_client_free**

**Purpose**
Frees storage returned from a client stub.

Used by client applications.

**Format**

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

void rpc_ss_client_free (idl_void_p_t node_to_free);
```

**Parameters**

**Input**

`node_to_free` Identifies a pointer to storage returned from a client stub.

**Usage**

The `rpc_ss_client_free` API releases storage 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 storage returned by an RPC call without knowledge of the storage management environment from which it was called.

**Note:** While this routine is always called from client code, the code can be running as part of another server.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_ss_free`
- `rpc_ss_get_thread_handle`
- `rpc_ss_set_client_alloc_free`
- `rpc_ss_set_thread_handle`
- `rpc_ss_swap_client_alloc_free`
**rpc_ss_destroy_client_context**

**Purpose**
Reclaims the client storage resources for the context handle, and sets the context handle to **NULL**.

Used by client applications.

**Format**
```c
#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.

**Usage**

The **rpc_ss_destroy_client_context** API 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 because this call makes the context handle unusable. When **rpc_ss_destroy_client_context** reclaims the storage resources, it also sets the context handle to null.

**Return Values**

None.

The **rpc_ss_destroy_client_context** API raises no exceptions.
**rpc_ss_disable_allocate**

**Purpose**
Releases resources and allocated storage.
Used by client applications.

**Format**

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

void rpc_ss_disable_allocate (void);
```

**Usage**

The `rpc_ss_disable_allocate` API releases (disables) all resources acquired by a call to `rpc_ss_enable_allocate`, and any storage 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 the memory management routines, see [z/OS DCE Application Development Guide: Core Components](#).

**Return Values**

The following exception may be raised. To capture the exceptions, use `TRY` and `CATCH` around the routines.

- `rpc_x_ss_posix_error` A POSIX error is detected.

**Related Information**

**Routines**

- `rpc_ss_allocate`
- `rpc_ss_enable_allocate`
rpc_ss_enable_allocate

Purpose
Enables the allocation of storage by the rpc_ss_allocate routine when not in manager code.

Used by client applications.

Format
#include <dce/rpc.h>

void rpc_ss_enable_allocate(void);

Usage
In sophisticated servers, it may be necessary to call manager code routines from different environments, 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 rpc_ss_allocate it must first call rpc_ss_enableAllocate to initialize the storage management environment that rpc_ss_allocate uses.

For information about rules for using memory management routines, see Z/OS DCE Application Development Guide: Core Components.

Return Values
The following exception may be raised. To capture the exceptions, use TRY and CATCH around the routines.

rpc_x_no_memory Insufficient storage is available to set up necessary data structures.

Related Information
Routines
rpc_ss_allocate rpc_ss_disable_allocate
rpc_ss_free

Purpose
Frees storage allocated by the rpc_ss_allocate API.
Used by server or possibly by client applications.

Format
#include <dce/rpc.h>

void rpc_ss_free (idl_void_p_t node_to_free);

Parameters
Input
node_to_free Specifies a pointer to storage allocated by rpc_ss_allocate.

Note: In ANSI standard C environments, such as C/C++ or ILE C/400, idl_void_p_t is defined as void *, and in other environments is defined as char *.

Usage
The rpc_ss_free API releases storage allocated by rpc_ss_allocate. The thread calling rpc_ss_free must have the same thread handle as the thread that allocated the storage with rpc_ss_allocate. Use it only in an environment where rpc_ss_allocate is used.

If the manager code allocates storage with rpc_ss_allocate and the storage is not released by rpc_ss_free during manager code processing, then the server stub releases the storage when the manager code completes running and returns control to the stub.

Manager code can also use rpc_ss_free to release storage that is pointed to by a full pointer in an input parameter.

For information about rules for using memory management routines, see z/OS DCE Application Development Guide: Core Components.

Return Values
The following exception may be raised. To capture the exceptions, use TRY and CATCH around the routines.

rpc_x_ss_posix_error A POSIX error is detected.

Related Information
Routines
rpc_ss_allocate rpc_ss_get_thread_handle rpc_ss_set_thread_handle
**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.

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

rpc_ss_thread_handle_t rpc_ss_get_thread_handle(void);
```

**Parameters**
None.

**Usage**
The `rpc_ss_get_thread_handle` API 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 the `rpc_ss_allocate` and `rpc_ss_free` routines for managing storage must first call the `rpc_ss_set_thread_handle` routine, using the handle obtained by the original manager thread.

The `rpc_ss_get_thread_handle` API 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`, specifying the previously obtained handle as a parameter.

**Return Value**
A thread handle.

If the returned thread handle is `NULL`, the `rpc_ss_get_thread_handle` API has failed.

**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.

This example catches specific exceptions. You can substitute `rpc_x_ss_pipe_memory` and `rpc_x_ss_pipe_closed` with the exceptions that the API raised.
#include <pthread.h>
#include <dce/rpc.h>

pthread_t Launch_thread(
    int *(routine_to_launch)(
        pthread_addr_t th
    )
)
{
    TRY
    /* calling API that may have exceptions raised */
    rpc_ss_thread_handle_t th =
    rpc_ss_get_thread_handle();
    pthread_t t;
    CATCH(rpc_x_ss_pipe_memory)
    /* do cleanup1 */
    CATCH(rpc_x_ss_pipe_closed)
    /* do cleanup2 */
    /* 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);
    ENDTry
    return t;
}

Related Information

Routines
rpc_ss_allocate                  rpc_ss_free                  rpc_ss_set_thread_handle
rpc_ss_set_client_alloc_free

Purpose
Sets the storage allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage storage for pointed-to nodes.

Used by client applications.

Format

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

void rpc_ss_set_client_alloc_free(
    idl_void_p_t (*p_allocate) (idl_size_t 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 C language `malloc` API and that is used by the client stub to allocate storage.
- **p_free** specifies a pointer to a routine that has the same procedure declaration as the C language `free` API and that is used to free storage that was allocated using the routine pointed at by `p_allocate`.

**Note:** In ANSI standard C environments, such as C/C++ or ILE C/400, `idl_void_p_t` is defined as `void *`, and in other environments is defined as `char *`.

Usage

The `rpc_ss_set_client_alloc_free` API overrides the default routines that the client stub uses to manage storage 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 [z/OS DCE Application Development Guide: Core Components](#).

Return Values

The following exceptions may be raised. To capture the exceptions, use TRY and CATCH around the routines.

- **rpc_x_no_memory** Insufficient storage is available to set up necessary data structures.
- **rpc_x_ss_posix_error** A POSIX error is detected.
- **rpc_x_ss_invalid_argument** The argument is not valid.

Related Information

rpc_ss_set_client_alloc_free

Routines

rpc_ss_allocate  rpc_ss_free
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.

Format

```c
#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`.

Usage

The `rpc_ss_set_thread_handle` API 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 the `rpc_ss_set_thread_handle` API, using the handle that the main RPC manager thread obtained through `rpc_ss_get_thread_handle`.

The `rpc_ss_set_thread_handle` API 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, specifying the handle it received (through `rpc_ss_get_thread_handle`) prior to becoming a server, as a parameter.

Return Values

The following exceptions may be raised. To capture the exceptions, use TRY and CATCH around the routines.

- `rpc_x_no_memory`  
  Insufficient storage is available to set up necessary data structures.

- `rpc_x_ss_posix_error`  
  A POSIX error is detected.

Examples

When this function is called within a spawned thread, its parameter is the thread handle of the calling thread. This example assumes the data passed to the thread consists of only the middle thread.

This example catches all the exceptions. Substitute the APIs that you would like to call inside the TRY and do the necessary cleanup after the CATCH_ALL.
rpc_ss_set_thread_handle

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

int helper_thread (pthread_addr_t th)
{
    /*
     * Set the memory management environment to match
     * the parent environment.
     */

    # TRY
    /* calling API that may have exceptions raised */
    register_ifs();
    rpc_server_listen(1, &status);
    rpc_ss_set_thread_handle(rpc_ss_thread_handle_tth);

    # CATCH_ALL
    /* cleanup action */
    unregister_ifs();

    # ENDTRY
    /*
     * Real work of this thread follows here
     */
}

Related Information

Routines

rpc_ss_get_thread_handle   rpc_ss_allocate   rpc_ss_free
**rpc_ss_swap_client_alloc_free**

**Purpose**
Exchanges the current storage allocation and freeing mechanism used by the client stubs with one specified by the client.

Used by client applications.

**Format**
```c
#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` API 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` API 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` API. 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` API. A pointer to the routine that was previously used to free client stub memory is returned in this parameter.

**Note:** In ANSI standard C environments, such as C/C++ or ILE C/400, `idl_void_p_t` is defined as `void *`, and in other environments is defined as `char *`.

**Usage**
The `rpc_ss_swap_client_alloc_free` API exchanges the current client allocate and free mechanism used by the client stubs for one specified by the caller. If it is appropriate for the client code called by an application to use a certain storage 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 exchange the caller’s mechanism back into place prior to returning.

For information about rules for using memory management routines, see [Z/OS DCE Application Development Guide: Core Components](#).

**Return Values**
The following exceptions may be raised. To capture the exceptions, use TRY and CATCH around the routines.
- `rpc_x_no_memory` Insufficient storage is available to set up necessary data structures.
- `rpc_x_ss posix_error` A POSIX error is detected.
rpc_ss_swap_client_alloc_free

rpc_x_ss_invalid_argument
  The argument is not valid.

Related Information

Routines

crpc_ss_allocate  rpc_ss_free  rpc_ss_set_client_alloc_free
**rpc_string_binding_compose**

**Purpose**
Combines the components of a string binding into a string binding.

Used by client or server applications.

**Format**

```c
#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 parameter. 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.
  Possible status codes and their meanings:
  - `rpc_s_ok`: Successful completion.
  - `rpc_s_no_memory`: No storage available.

**Usage**

The `rpc_string_binding_compose` API combines string binding handle components into a string binding handle.

The RPC runtime allocates storage for the string returned in the `string_binding` parameter. The application calls `rpc_string_free` to deallocate that storage.

Specify NULL or provide a null string (`\0`) for each input string that has no data.

`rpc_string_binding_compose` does not check the validity of the parsed parameters.

**Return Values**

None.
rpc_string_binding_compose

Related Information

Routines
rpc_binding_from_string_binding   rpc_string_binding_parse   uuid_to_string
rpc_binding_to_string_binding    rpc_string_free
rpc_string_binding_parse

Purpose
Returns, as separate strings, the components of a string binding.

Used by client or server applications.

Format

```
#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. 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. 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. 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. 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. 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.

Possible status codes and their meanings:

- **rpc_s_ok** Successful completion.
- **rpc_s_invalid_string_binding** String binding is not valid.
- **rpc_s_no_memory** No storage available.
**rpc_string_binding_parse**

**Usage**

The `rpc_string_binding_parse` API parses a string representation of a binding handle into its component fields.

The RPC runtime allocates storage for each component string the routine returns. The application calls `rpc_string_free` once for each returned string to deallocate the storage for that string.

If any field of the `string_binding` is empty, `rpc_string_binding_parse` returns the empty string in the corresponding output parameter.

**Return Values**

None.

**Related Information**

**Routines**

- `rpc_binding_from_string_binding`
- `rpc_string_binding_compose`
- `uuid_from_string`
- `rpc_binding_to_string_binding`
- `rpc_string_free`
**rpc_string_free**

**Purpose**
Frees a character string allocated by the runtime.

Used by client, server, or management applications.

**Format**

```c
#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.
  Possible status code and its meaning:
  - **rpc_s_ok**
    Successful completion.

**Usage**

The `rpc_string_free` API deallocates the storage 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 in the “Related Information” on this page.

**Return Values**

None.

**Related Information**

**Routines**

- `dce_error_inq_text`
- `rpc_binding_inq_auth_client`
- `rpc_binding_inq_auth_info`
- `rpc_binding_to_string_binding`
- `rpc_mgmt_ep_elt_inq_next`
- `rpc_mgmt_inq_server_princ_name`
- `rpc_ns_binding_inq_entry_name`
- `rpc_ns_entry_expand_name`
- `rpc_ns_group_mbr_inq_next`
- `rpc_string_binding_compose`
- `rpc_string_binding_parse`
- `uuid_to_string`
**rpc_tower_to_binding**

**Purpose**
Returns a binding handle from a tower representation.

Used by client applications.

**Format**
```
#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.

The possible status codes and their meanings are as follows:
- **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.

**Usage**
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.

"Binding Handle" on page 2-9 contains an explanation of binding handles.

**Return Values**
None.

**Related Information**

Routines:
- **rpc_binding_copy**
- **rpc_binding_free**
- **rpc_tower_vector_from_binding**
- **rpc_tower_vector_free**
**rpc_tower_vector_free**

**Purpose**
Releases memory associated with a tower vector.

Used by server applications.

**Format**

```c
#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.

**Usage**

The `rpc_tower_vector_free()` routine releases memory associated with a tower vector, including the towers as well as the vector.

**Return Values**

None.

**Related Information**

Routines:

- `rpc_binding_copy`
- `rpc_binding_free`
- `rpc_tower_vector_from_binding`
- `rpc_tower_to_binding`
Purpose
Creates a tower vector from a binding handle.

Used by server applications.

Format
#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 an 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.

Usage
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
None.

Related Information
Routines:
rpc_binding_copy rpc_binding_free rpc_tower_vector_free
rpc_tower_to_binding
**uuid_compare**

**Purpose**
Compares two UUIDs to determine their order.

Used by client, server, or management applications.

**Format**

```c
#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.

  Possible status codes and their meanings:

  - `uuid_s_ok` Successful completion.
  - `uuid_s_bad_version` Incorrect UUID version.

**Usage**

The `uuid_compare` API compares two UUIDs to determine their order. A nil UUID is considered the first element in order. The order of UUIDs is not related to time. 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**

This routine returns one of the following constants:

- `-1` `uuid1` precedes `uuid2` in order.
- `0` `uuid1` is equal to `uuid2` in order.
- `1` `uuid1` follows `uuid2` 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 non-nil, routine returns `-1`.
- If `uuid1` is `NULL` and `uuid2` is `NULL`, routine returns `0`.
- If `uuid1` is non-nil and `uuid2` is `NULL`, routine returns `1`. 

Chapter 2. Remote Procedure Call 2-315
Related Information

Routines

- uuid_compare
- uuid_equal
- uuid_is_nil
uuid_create

Purpose
Creates a new UUID.

Used by client, server, or management applications.

Format

```c
#include <dce/uuid.h>

void uuid_create(
    uuid_t *uuid,
    unsigned32 *status);
```

Parameters

Input
None.

Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>uuid</td>
<td>Returns the new UUID.</td>
</tr>
<tr>
<td>status</td>
<td>Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes and their meanings:</td>
</tr>
<tr>
<td></td>
<td>uuid_s_ok: Successful completion.</td>
</tr>
<tr>
<td></td>
<td>uuid_s_no_address: Cannot get Ethernet hardware address.</td>
</tr>
<tr>
<td></td>
<td>uuid_s_internal_error: Internal coding error.</td>
</tr>
</tbody>
</table>

Usage

The `uuid_create` API creates a new UUID.

Return Values

None.

Related Information

Routines

- `uuid_create_nil`
- `uuid_from_string`
- `uuid_to_string`
uuid_create_nil

Purpose
Creates a nil UUID.
Used by client, server, or management applications.

Format
#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.

Possible status code and its meaning:

  uuid_s_ok            Successful completion.

Usage
The uuid_create_nil API creates a nil UUID.

Return Values
None.

Related Information
Routines
uuid_create
uuid_equal

Purpose
Determines whether two UUIDs are equal.

Used by client, server, or management applications.

Format

```c
#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.

Possible status codes and their meanings:

- `uuid_s_ok` Successful completion.
- `uuid_s_bad_version` Incorrect UUID version.

Usage

The `uuid_equal` API compares two UUIDs to determine if they are equal.

Return Values

The possible return values and their meanings are:

- `true` `uuid1` is equal to `uuid2`. Parameter `status` contains the status code `uuid_s_ok`.
- `false` `uuid1` is not equal to `uuid2`.

Related Information

Routines

`uuid_compare`
**Purpose**

Converts a string UUID to its binary representation.

Used by client, server, or management applications.

**Format**

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

Possible status codes and their meanings:

- `uuid_s_ok`: Successful completion.
- `uuid_s_bad_version`: Incorrect UUID version.
- `uuid_s_invalid_string_uuid`: Format is not valid for a string UUID

**Usage**

The `uuid_from_string` API converts string UUID to its binary representation.

**Return Values**

None.

**Related Information**

Routines

- `uuid_to_string`
uuid_hash

Purpose
Creates a hash value for a UUID.
Used by client, server, or management applications.

Format
#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.
Possible status codes and their meanings:

    uuid_s_ok            Successful completion.
    uuid_s_bad_version   Incorrect UUID version.

Usage
The uuid_hash API generates a hash value for a specified UUID.
The return value for a single uuid value may differ across platforms.

Return Values
Returns a hash value for the specified UUID.
Purpose
Determines whether a UUID is nil.
Used by client, server, or management applications.

Format
#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.
Possible status codes and their meanings:
- uuid_s_ok
  Successful completion.
- uuid_s_bad_version
  Incorrect UUID version.

Usage
The uuid_is_nil API determines whether the specified UUID is a nil UUID. This routine gives the same result as if an application did the following:
- Called the uuid_create_nil API
- Called the uuid_equal API to compare the returned nil UUID to the UUID specified in the uuid parameter

Return Values
The possible return values and their meanings are:
- true
  uuid is a nil UUID. Parameter status contains the status code uuid_s_ok.
- false
  uuid is not a nil UUID.

Related Information
Routines
- uuid_compare
- uuid_create_nil
- uuid_equal
uuid_to_string

Purpose
Converts a UUID from a binary representation to a string representation.

Used by client, server, or management applications.

Format

```c
#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.

Possible status codes and their meanings:

- **uuid_s_ok**: Successful completion.
- **uuid_s_bad_version**: Incorrect UUID version.
- **uuid_s_no_memory**: No storage available.

Usage

The `uuid_to_string` API converts a UUID from its binary representation to its string representation.

The RPC runtime allocates storage for the string returned in the `string_uuid` parameter. The application calls the `rpc_string_free` API to deallocate that storage. It is not necessary to call `rpc_string_free` when you specify NULL for the `string_uuid` parameter.

Return Values

None.

Related Information

Routines

- `rpc_string_free`
- `uuid_from_string`
Chapter 3. Threads

DCE Threads is a set of APIs that you can call to create a multithreaded program. This chapter provides an overview of the Threads APIs and discusses each one in detail.

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 with an \_np suffix. These APIs can only be called by DCE applications.

The following threads APIs are not supported in z/OS DCE and are not discussed in detail in this chapter. If they are called, -1 is returned, and \_errno is set to ENOSYS.

- pthread_attr_getinheritsched
- pthread_attr_getprio
- pthread_attr_getsched
- pthread_attr_setinheritsched
- pthread_attr_setprio
- pthread_attr_setsched
- pthread_getprio
- pthread_getscheduler
- pthread_setprio
- pthread_setscheduler

The Threads APIs can be divided into ten different functional groups:

- Threads Management
- Attributes Object Routines
- Mutex Routines
- Condition Variable Routines
- Thread-Specific Data Routines
- Cancelation Routines
- Priority and Scheduling Routines
- Cleanup Routines
- Exceptions in DCE Threads.
- The sigwait Routine

The following tables show which Threads APIs belong to the various functional groups.

## Threads Management

### Table 3-1. Threads Management

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Creates a thread.</td>
</tr>
<tr>
<td>pthread_delay_np</td>
<td>Causes a thread to wait for a period of time.</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>Marks a thread for deletion.</td>
</tr>
<tr>
<td>pthread_equal</td>
<td>Compares one thread identifier to another thread identifier.</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminates the calling thread.</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Causes the calling thread to wait for the termination of a specified thread.</td>
</tr>
<tr>
<td>pthread_once</td>
<td>Calls an initialization routine to be run only once.</td>
</tr>
<tr>
<td>pthread_self</td>
<td>Obtains the identifier of the current thread.</td>
</tr>
<tr>
<td>pthread_yield</td>
<td>Notifies the scheduler that the current thread will release its processor to other threads of the same or higher priority.</td>
</tr>
</tbody>
</table>
### Attributes Object Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_create</td>
<td>Creates a thread attributes object.</td>
</tr>
<tr>
<td>pthread_attr_delete</td>
<td>Deletes a thread attributes object.</td>
</tr>
<tr>
<td>pthread_attr_getinheritsched</td>
<td>Obtains the inherit scheduling attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_getprio</td>
<td>Obtains the scheduling priority attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_getsched</td>
<td>Obtains the scheduling policy attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_getstacksize</td>
<td>Obtains the stacksize attribute.</td>
</tr>
<tr>
<td>pthread_attr_setinheritsched</td>
<td>Changes the inherit scheduling attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_setprio</td>
<td>Changes the scheduling priority attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_setsched</td>
<td>Changes the scheduling policy attribute. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_attr_setstacksize</td>
<td>Changes the stacksize attribute.</td>
</tr>
<tr>
<td>pthread_condattr_create</td>
<td>Creates a condition variable attributes object.</td>
</tr>
<tr>
<td>pthread_condattr_delete</td>
<td>Deletes a condition variable attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_create</td>
<td>Creates a mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_delete</td>
<td>Deletes a mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_getkind_np</td>
<td>Obtains the mutex type attribute.</td>
</tr>
<tr>
<td>pthread_mutexattr_setkind_np</td>
<td>Changes the mutex type attribute.</td>
</tr>
</tbody>
</table>

### Mutex Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_lock_global_np</td>
<td>Locks the global mutex.</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Deletes a mutex.</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Creates a mutex.</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Locks a mutex and waits if mutex is already locked.</td>
</tr>
<tr>
<td>pthread_mutex_trylock</td>
<td>Locks a mutex and returns if mutex is already locked.</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlocks a mutex.</td>
</tr>
<tr>
<td>pthread_unlock_global_np</td>
<td>Unlocks the global mutex.</td>
</tr>
</tbody>
</table>
Condition Variable Routines

Table 3-4. Condition Variable Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cond_broadcast</td>
<td>Wakes all threads waiting on a condition variable.</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Deletes a condition variable.</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Creates a condition variable.</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Wakes one thread waiting on a condition variable.</td>
</tr>
<tr>
<td>pthread_cond_timedwait</td>
<td>Causes a thread to wait for a specified period of time for a condition variable to be signaled or broadcast.</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Causes a thread to wait for a condition variable to be signaled or broadcast.</td>
</tr>
<tr>
<td>pthread_get_expiration_np</td>
<td>Obtains a value representing a desired expiration time.</td>
</tr>
</tbody>
</table>

Thread-Specific Data Routines

Table 3-5. Thread-Specific Data Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_getspecific</td>
<td>Obtains the thread-specific data associated with the specified key.</td>
</tr>
<tr>
<td>pthread_keycreate</td>
<td>Generates a unique thread-specific data key value.</td>
</tr>
<tr>
<td>pthread_setspecific</td>
<td>Sets the thread-specific data associated with the specified key.</td>
</tr>
</tbody>
</table>

Cancellation Routines

Table 3-6. Cancellation Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cancel</td>
<td>Allows a thread to request termination.</td>
</tr>
<tr>
<td>pthread_setasynccancel</td>
<td>Allows the current thread to be canceled asynchronously.</td>
</tr>
<tr>
<td>pthread_setcancel</td>
<td>Allows the current thread to be canceled generally.</td>
</tr>
<tr>
<td>pthread_testcancel</td>
<td>Requests delivery of a pending cancelation.</td>
</tr>
</tbody>
</table>

Priority and Scheduling Routines

Table 3-7. Priority and Scheduling Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_getprio</td>
<td>Obtains the current priority of a thread. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_getscheduler</td>
<td>Obtains the current scheduling policy of a thread. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_setprio</td>
<td>Changes the current priority of a thread. This API is not supported in z/OS DCE.</td>
</tr>
<tr>
<td>pthread_setscheduler</td>
<td>Changes the current scheduling policy and priority of a thread. This API is not supported in z/OS DCE.</td>
</tr>
</tbody>
</table>

Cleanup Routines

Table 3-8. Cleanup Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cleanup_pop</td>
<td>Removes a cleanup handler from the stack.</td>
</tr>
<tr>
<td>pthread_cleanup_push</td>
<td>Establishes a cleanup handler.</td>
</tr>
</tbody>
</table>
### Exceptions in DCE Threads

**Table 3-9. Exceptions in DCE Threads**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exc_first_catch</td>
<td>Returns nonzero when the current exception is first raised.</td>
</tr>
<tr>
<td>exc_set_status</td>
<td>Creates a status exception.</td>
</tr>
<tr>
<td>exc_get_status</td>
<td>Retrieves the status from a status exception.</td>
</tr>
<tr>
<td>exc_raise_status</td>
<td>Raise system defined status as an exception.</td>
</tr>
<tr>
<td>exc_get_reason_code</td>
<td>Retrieves the reason code.</td>
</tr>
<tr>
<td>exc_matches</td>
<td>Compares two exception objects.</td>
</tr>
</tbody>
</table>

### The sigwait Routine

**Table 3-10. The sigwait Routine**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigwait</td>
<td>Causes thread to wait for asynchronous signal.</td>
</tr>
</tbody>
</table>
Exceptions in DCE Threads

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 one of these methods of receiving status. They 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 information documents these status return values. 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 dce/exc_handling.h file. The dce/exc_handling.h header file is included when you include dce/pthread_exc.h.

To use the DCE threads package, define the C language macro constant _DCE_THREADS. For information on defining this constant, see z/OS DCE Application Development Guide: Core Components.

To use the exception-returning interface, replace #include <pthread.h> with #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
```
EXCEPTION_INIT

Purpose
Initializes an exception object.

Format

```c
#include <dce/exc_handling.h>

EXCEPTION_INIT( EXCEPTION error )
```

Parameters

Input

- `error` Exception object to be initialized.

Usage

The `EXCEPTION_INIT` macro dynamically initializes an exception object that has already been declared and defined.

Return Values

None.

Related Information

Macros

- exception APIs
  - CATCH
  - RERAISE
- RAISE
  - CATCH_ALL
- TRY-ENDTRY
  - FINALLY
RAISE

Purpose
Raises an exception.

Format
#include <dce/exc_handling.h>

RAISE( EXCEPTION error )

Parameters
Input
error Exception to be raised.

Usage
The RAISE macro causes the exception package to search backwards through all active scopes which have declared an interest in catching exceptions. Within each scope, the particular exception may be caught; otherwise the exception is ignored and the scope is removed from the stack as the exception moves backwards. Once a RAISE has been made, the program can no longer run beyond this point and resumes within the scope where it is caught.

Return Values
None.

Related Information
Macros

<table>
<thead>
<tr>
<th>exception APIs</th>
<th>CATCH</th>
<th>RERAISE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_INIT</td>
<td>CATCH_ALL</td>
<td>FINALLY</td>
</tr>
<tr>
<td>TRY-ENDTRY</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TRY-ENDTRY

Purpose
Defines the beginning and end of the scope of the exceptions.

Format
#include <dce/exc_handling.h>

TRY {
    /* function calls that may raise an exception */
}
ENDTRY

Parameters
None.

Usage
The TRY-ENDTRY macros define a scope where exceptions may be caught. An exception raised within this block or within the functions that are in this scope will have control passed through it. The TRY macro defines the beginning of the scope and the ENDTRY macro defines the end of the scope.

Important note to Users:
Do not place a RETURN or nonlocal GOTO between TRY and ENDTRY as this destroys the integrity of the exception handling facility and may cause indeterminate problems. It is not valid to use RETURN or GOTO, or to leave by any other means, a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro, and it must be run.

For more information on the rules for using exceptions see z/OS DCE Application Development Guide: Core Components

Return Values
None.

Related Information

Macros

<table>
<thead>
<tr>
<th>exception APIs</th>
<th>CATCH</th>
<th>RERAISE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_INIT</td>
<td>CATCH_ALL</td>
<td>FINALLY</td>
</tr>
<tr>
<td>RAISE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CATCH

Purpose
Defines an interest in catching a particular exception.

Format

```c
#include <dce/exc_handling.h>

TRY {
    /* EXCEPTION error raised */
}
CATCH (error) {
    /* exception handling functions */
}
ENDTRY
```

Parameters
Input
- `error` Exception to be caught when executing within the scope defined.

Usage
The CATCH macro defines an interest in catching a particular exception within a TRY-ENDTRY scope. When an exception is raised, the first CATCH macro that contains the matching exception will gain control.

---

Important note to Users:

Do not place a RETURN or nonlocal GOTO between TRY and ENDTRY as this destroys the integrity of the exception handling facility and may cause indeterminate problems. It is not valid to use RETURN or GOTO, or to leave by any other means, a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro, and it must be run.


---

Return Values
None.

Related Information

Macros

- `exception APIs`
- `TRY-ENDTRY`
- `RERAISE`
- `EXCEPTION_INIT`
- `CATCH_ALL`
- `FINALLY`
- `RAISE`
CATCH_ALL

Purpose
Defines an interest in catching all exceptions.

Format
#include <dce/exc_handling.h>

TRY {
  /* EXCEPTION error raised */
}
CATCH (error) {
  /* exception handling functions */
}
CATCH_ALL {
  /* generic exception handling functions */
}
ENDTRY

Usage
The CATCH_ALL macro defines an interest in catching all exceptions within a TRY-ENDTRY scope. It is generally used to catch an exception not explicitly caught. An exception caught within a CATCH_ALL could be reraised to allow other scopes to catch this exception.

Important note to Users:

Do not place a RETURN or nonlocal GOTO between TRY and ENDTRY as this destroys the integrity of the exception handling facility and may cause indeterminate problems. It is not valid to use RETURN or GOTO, or to leave by any other means, a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro, and it must be run.

For more information on the rules for using exceptions see z/OS DCE Application Development Guide: Core Components.

Return Values
None.

Related Information

Macros

<table>
<thead>
<tr>
<th>exception APIs</th>
<th>TRY-ENDTRY</th>
<th>RERAISE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_INIT</td>
<td>CATCH</td>
<td>FINALLY</td>
</tr>
</tbody>
</table>

RAISE
**Purpose**

Raises the current exception.

**Format**

```c
#include <dce/exc_handling.h>

TRY {
    /* EXCEPTION error raised */
}
CATCH ( error ) {
    /* exception handling functions */
}
CATCH_ALL {
    /* generic exception handling functions */
    RERAISE:
}
ENDTRY
```

**Parameters**

None.

**Usage**

The **RERAISE** macro raises the current exception being handled in the catch block. The current exception is found in a local variable **THIS_CATCH**, that is accessible only from within a **CATCH/CATCH_ALL** block.

**Return Values**

None.

**Related Information**

**Macros**

<table>
<thead>
<tr>
<th>exception APIs</th>
<th>TRY-ENDTRY</th>
<th>CATCH_ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_INIT</td>
<td>CATCH</td>
<td>FINALLY</td>
</tr>
<tr>
<td>RAISE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FINALLY

Purpose
Defines a set of actions to execute at exit of the TRY-ENDTRY scope.

Format
#include <dce/exc_handling.h>
TRY {
   /* EXCEPTION error raised */
} 
FINALLY {
   /* exception handling functions */
} ENDTRY

Parameters
None.

Usage
The FINALLY macro defines a set of actions which will execute whether or not an exception occurred within the scope. If an exception occurs, the FINALLY macro will catch and then reraise the exception for outer scopes to process. To avoid duplicate code, this macro will also perform the actions when the scope exits normally without an exception. The FINALLY macro cannot be combined with the CATCH and CATCH_ALL clauses.

Important note to Users:
Do not place a RETURN or nonlocal GOTO between TRY and ENDTRY as this destroys the integrity of the exception handling facility and may cause indeterminate problems. It is not valid to use RETURN or GOTO, or to leave by any other means, a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro, and it must be run.

For more information on the rules for using exceptions see z/OS DCE Application Development Guide: Core Components.

Return Values
None.

Related Information
Macros

exception APIs TRY-ENDTRY CATCH_ALL
EXCEPTION_INIT CATCH RERAISE
RAISE
exc_first_catch

Purpose
Returns nonzero when the current exception is raised for the first time.

Format
#include <dce/exc_handling.h>

int exc_first_catch ( void )

Parameters
None.

Usage
The exc_first_catch API returns nonzero when the current exception is raised for the first time and is not caught as a RERAISE. This call can only be used within the CATCH or CATCH_ALL scope, and is not defined outside this scope, resulting in compiling errors.

Return Values
0 Indicates that no exception was raised, or that the exception was reraised.
non-0 Indicates that the exception was not caused by a RERAISE.

Related Information
Routines
exc_set_status exc_get_reason_code exc_matches
exc_get_status
Exc_get_reason_code

Purpose
Retrieves the reason code.

Format
#include <dce/exc_handling.h>

int exc_get_reason_code
(EXCEPTION *exc, int *rc)

Parameters
Input
exc Specifies the exception for which the reason code is to be obtained.

Output
rc Specifies the reason code value to be returned to the caller.

Usage
The exc_get_reason_code API retrieves the reason code associated with the exception caused by a z/OS abend.

This call can be used after an abend exception has been caught so that you can obtain both the abend code, which is a part of the exception status value, and the reason code. If exc_get_reason_code is called when an exception has not occurred, or the exception is not an abend, then the reason code returned is set to 0.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that the routine cannot set the rc parameter.

Related Information
Routines
exc_first_catch exc_get_status exc_matches
exc_set_status
exc_get_status

Purpose
Retrieves the status value from a status exception.

Format
#include <dce/exc_handling.h>

int exc_get_status
( EXCEPTION *error, int *status )

Parameters
Input
error Specifies the status object to retrieve the status value from.
status Specifies the status value to be returned to the caller.

Usage
The exc_get_status API retrieves the status value from a status exception.

If the exception was caused by an unmapped z/OS abend, then the corresponding abend code can be obtained from the status value returned from this call. The status value is 0x000ssuuu, where sss is the system abend code or uuu is the user abend code. If the reason code is required with the abend code, exc_get_reason_code is called.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that the routine cannot get the status parameter.

Related Information
Routines
exc_first_catch   exc_get_reason_code   exc_matches
exc_set_status
exc_matches

Purpose
Compares two exception objects.

Format

```c
#include <dce/exc_handling.h>

int exc_matches
    ( EXCEPTION *my_error,
    EXCEPTION *c_error )
```

Parameters

**Input**

- `my_error` Specifies the exception object.
- `c_error` Specifies the exception object to be compared.

Usage

The `exc_matches` API compares two exception objects to see if they have the same status value.

Return Values

- **0** Indicates that the exceptions have different status values
- **non-0** Indicates that the exceptions have the same status value.

Related Information

**Routines**

- `exc_first_catch`
- `exc_get_status`
- `exc_get_reason_code`
- `exc_set_status`
exc_raise_status

Purpose
Raises a status exception containing the specified status.

Format

```c
#include <dce/exc_handling.h>

void exc_raise_status ( int *status )
```

Parameters

Input
status Specifies the status value of the status exception to be raised.

Usage

The `exc_raise_status` API raises a status exception containing the status value specified as the input parameter.

The call does not return to the caller and control is passed to a CATCH or CATCH_ALL area.

Return Values
None

Related Information

Routines

- exc_first_catch
- exc_get_reason_code
- exc_matches
- exc_set_status
exc_set_status

Purpose
Sets an exception as status type.

Format
#include <dce/exc_handling.h>

int exc_set_status
( EXCEPTION *error, int status )

Parameters
Input
error Specifies the exception object that is to be assigned an associated status.
status Specifies the status exception type that is to be created for the object.

Usage
The exc_set_status API sets the exception as status type and assigns a status value.

This call is used to set the status of an exception in order to catch the specific exception specified. The exception object error must first be initialized with EXCEPTION_INIT.

The status value of an exception caused by an abend is a hexadecimal value based on the z/OS abend code, $0x00sssuuu$, where sss is the system abend code or uuu is the user abend code. Other exception types have a status value that does not conflict with the abend code values, that is the low bit value is set to 1.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that the routine cannot get the status parameter.

Related Information
Routines
exc_first_catch exc_get_reason_code exc_matches
exc_get_status
Threads Routines

This section describes in alphabetical order each DCE Threads routine that is supported by z/OS.
pthread_attr_create

Purpose
Creates a thread attributes object.

Format

```
#include <pthread.h>

int pthread_attr_create( 
    pthread_attr_t *attr );
```

Parameters

*attr  
Thread attributes object created.

Usage

The `pthread_attr_create` API creates a thread attributes object 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 query and change the individual attributes:

- `pthread_attr_setstacksize`
- `pthread_attr_getstacksize`

When an attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new object. Attributes objects perform in a manner similar to additional parameters. Changing individual attributes does not affect any objects that were previously created using the attributes object.

Return Values

If the routine fails -1 is returned and `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to create the thread attributes object.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <em>attr</em> is not valid.</td>
</tr>
</tbody>
</table>

Related Information

Routines

- `pthread_attr_delete`
- `pthread_attr_getstacksize`
- `pthread_attr_setstacksize`
- `pthread_create`
`pthread_attr_delete`

**Purpose**
Deletes a thread attributes object.

**Format**
```c
#include <pthread.h>

int pthread_attr_delete(
    pthread_attr_t *attr);
```

**Parameters**
- `attr`: Thread attributes object deleted.

**Usage**
The `pthread_attr_delete` API deletes a thread attributes object and gives permission to reclaim storage for the thread attributes object. Threads that were created using the 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>attr</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**
- **Routines**
  - `pthread_attr_create`
**pthread_attr_getstacksize**

**Purpose**
Obtains the value of the stacksize attribute in the specified thread attributes object.

**Format**

```c
#include <pthread.h>

unsigned long pthread_attr_getstacksize(
    pthread_attr_t attr);
```

**Parameters**

- **attr**  
  Thread attributes object whose stacksize attribute is obtained.

**Usage**

The `pthread_attr_getstacksize` API 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td><code>EINVAL</code> The value specified by attr is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_attr_create`
- `pthread_attr_setstacksize`
- `pthread_create`
**pthread_attr_setstacksize**

**Purpose**
Changes the stacksize attribute of thread creation.

**Format**

```c
#include <pthread.h>

int pthread_attr_setstacksize(
    pthread_attr_t *attr,
    long stacksize);
```

**Parameters**

- `attr`: Thread attributes object to be changed.
- `stacksize`: New value for the stacksize attribute. The stacksize parameter specifies the minimum size (in bytes) of the stack needed for a thread.

**Usage**

The `pthread_attr_setstacksize` API 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>attr</code> is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>stacksize</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_attr_create`
- `pthread_attr_getstacksize`
- `pthread_create`
pthread_cancel

Purpose
Allows a thread to request that it or another thread terminate execution.

Format
#include <pthread.h>

int pthread_cancel(
    pthread_t thread);

Parameters
thread Thread that receives a cancelation request.

Usage
The pthread_cancel API sends a cancelation request to the specified thread. A cancelation request is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancelation request does not guarantee that the canceled thread receives or handles the cancelation. The canceled thread can delay processing the cancelation after receiving it, for instance, if a cancelation arrives during an important operation.

Termination processing when a cancelation is delivered to a thread is similar to pthread_exit. Outstanding cleanup APIs are run in the context of the target thread, and a status of -1 is made available to any threads joining with the target thread.

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 routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The specified thread is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>ERSCH</td>
<td>The specified thread does not refer to a currently existing thread.</td>
</tr>
</tbody>
</table>

Related Information

Routines
pthread_exit  pthread_setasynccancel  pthread_testcancel
pthread_join  pthread_setcancel
**pthread_cleanup_pop**

**Purpose**
Removes the cleanup handler at the top of the cleanup stack and optionally runs it.

**Format**
```
#include <pthread.h>

void pthread_cleanup_pop(
    int execute);
```

**Parameters**
- **execute**  
  Integer that specifies whether the cleanup routine that is popped should be run or just discarded. If the value is nonzero, the cleanup routine runs.

**Usage**
The `pthread_cleanup_pop` API removes the API specified in `pthread_cleanup_push` from the top of the calling thread’s cleanup stack and runs it if the value specified in `execute` is nonzero.

This API and `pthread_cleanup_push` must be displayed as statements and in pairs within the same lexical scope.

This routine must be used as a statement.

**Return Values**
None.

**Related Information**

**Routines**
`pthread_cleanup_push`
**pthread_cleanup_push**

**Purpose**
Establishes a cleanup handler that is run when the thread exits or is canceled.

**Format**

```c
#include <pthread.h>

void pthread_cleanup_push(
    pthread_cleanup_t routine,
    pthread_addr_t arg);
```

**Parameters**

- `routine` Routine ran as the cleanup handler.
- `arg` Parameter ran with the cleanup routine.

**Usage**

The `pthread_cleanup_push` API pushes the specified routine onto the calling thread's cleanup stack. The cleanup API is popped from the stack and run 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 API and `pthread_cleanup_pop` must be displayed as statements and in pairs within the same lexical scope.

This API must be used as a statement.

**Return Values**

None.

**Related Information**

**Routines**

- `pthread_cancel`
- `pthread_exit`
- `pthread_cleanup_pop`
- `pthread_testcancel`
**pthread_cond_broadcast**

**Purpose**
Wakes all threads that are waiting on a condition variable.

**Format**
```c
#include <pthread.h>

int pthread_cond_broadcast(
    pthread_cond_t *cond);
```

**Parameters**

`cond`  
Condition variable that is broadcast.

**Usage**
The `pthread_cond_broadcast` API wakes all threads waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed. This allows for one or more waiting threads to proceed. If any thread is waiting to proceed, call `pthread_cond_signal`.

You can call this routine when the associated mutex is either locked or unlocked.

**Return Values**
If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>cond</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_cond_destroy`
- `pthread_cond_init`
- `pthread_cond_signal`
- `pthread_cond_wait`
- `pthread_cond_timedwait`
Purpose
Deletes a condition variable.

Format
#include <pthread.h>

int pthread_cond_destroy(
    pthread_cond_t *cond);

Parameters
cond Condition variable that is deleted.

Usage
The pthread_cond_destroy API destroys a condition variable. Call this API when a condition variable is no longer referred to. The effect of calling this routine is to give permission to reclaim system storage for the condition variable. Note that if you created the mutex in dynamically allocated memory, you must reclaim that storage to avoid memory leaks.

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 as it is destroyed.

Return Values
If the routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by cond is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EBUSY</td>
<td>A thread is currently running a pthread_cond_wait or pthread_cond_timedwait on the condition variable specified in cond.</td>
</tr>
</tbody>
</table>

Related Information

Routines
pthread_cond_broadcast  pthread_cond_signal  pthread_cond_wait
pthread_cond_init        pthread_cond_timedwait
**pthread_cond_init**

**Purpose**
Creates a condition variable.

**Format**
```c
#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.

**Usage**
The `pthread_cond_init` API initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data, whereas 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 re-evaluate 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 releases the mutex so that other threads can change or examine the shared data. When a thread changes 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, a client can store condition variables and mutexes and later re-use 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 system is not deallocated when the creating thread terminates.

**Return Values**
If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EAGAIN</td>
<td>The system lacks the necessary resources to initialize another condition variable.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>Invalid attributes object.</td>
</tr>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to initialize the condition variable.</td>
</tr>
</tbody>
</table>
pthread_cond_init

Related Information

Routines

<table>
<thead>
<tr>
<th>pthread_cond_broadcast</th>
<th>pthread_cond_signal</th>
<th>pthread_cond_wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cond_destroy</td>
<td>pthread_cond_timedwait</td>
<td></td>
</tr>
</tbody>
</table>
**pthread_cond_signal**

**Purpose**
Wakes one thread that is waiting on a condition variable.

**Format**

```c
#include <pthread.h>

int pthread_cond_signal(
    pthread_cond_t *cond );
```

**Parameters**

- `cond`: Condition variable that is signaled.

**Usage**
The `pthread_cond_signal` API wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so a single waiting thread can proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true, but only if one thread needs to proceed.

You can call this routine when the associated mutex is either locked or unlocked.

**Return Values**

If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>cond</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_cond_broadcast`
- `pthread_cond_init`
- `pthread_cond_wait`
- `pthread_cond_destroy`
- `pthread_cond_timedwait`
pthread_cond_timedwait

Purpose
Causes a thread to wait for a condition variable to be signaled or broadcast for a specified period of time.

Format

```c
#include <pthread.h>

int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    struct timespec *abstime);
```

Parameters

- `cond`  Condition variable that is 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. You can use `pthread_get_expiration_np` on page 3-44, to obtain a value for this parameter.

Usage

The `pthread_cond_timedwait` API 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 API 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 the mutex first being locked.

Note to Users

This API uses the system time.

Return Values

If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>cond, mutex</code> or <code>abstime</code> is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EAGAIN</td>
<td>The time specified by <code>abstime</code> expired.</td>
</tr>
<tr>
<td>-1</td>
<td>EDEADLK</td>
<td>A deadlock condition is detected.</td>
</tr>
</tbody>
</table>

Related Information
Routines

- pthread_cond_broadcast
- pthread_cond_destroy
- pthread_cond_init
- pthread_cond_signal
- pthread_cond_wait
- pthread_get_expiration_np
**pthread_cond_wait**

**Purpose**
Causes a thread to wait for a condition variable to be signaled or broadcast.

**Format**
#include <pthread.h>

int pthread_cond_wait(
   pthread_cond_t *cond ,
   pthread_mutex_t *mutex );

**Parameters**
- **cond**: Condition variable that is waited on.
- **mutex**: Mutex associated with the condition variable specified in **cond**.

**Usage**
The **pthread_cond_wait** API 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 API are unpredictable if this API is called without first locking the mutex.

This API 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 re-acquired 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 will, with low probability, return when the condition variable has not been signaled or broadcast. When a spurious wakeup occurs, the mutex is re-acquired before the routine returns. (To handle this type of situation, enclose this routine in a loop that checks the predicate.)

**Return Values**
If the routine fails, **errno** may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <strong>cond</strong> or <strong>mutex</strong> is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EDEADLK</td>
<td>A deadlock condition is detected.</td>
</tr>
</tbody>
</table>

**Related Information**
### Routines

<table>
<thead>
<tr>
<th>pthread_cond_broadcast</th>
<th>pthread_cond_init</th>
<th>pthread_cond_timedwait</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cond_destroy</td>
<td>pthread_cond_signal</td>
<td></td>
</tr>
</tbody>
</table>
pthread_condattr_create

pthread_condattr_create

Purpose
Creates a condition variable attributes object.

Format
#include <pthread.h>

int pthread_condattr_create(
    pthread_condattr_t *attr);

Parameters
attr              Condition variable attributes object that is created.

Usage
The pthread_condattr_create API 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 routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to create the condition variable attributes object.</td>
</tr>
</tbody>
</table>

Related Information
Routines
pthread_condattr_delete     pthread_cond_init
**pthread_condattr_delete**

**Purpose**
Deletes a condition variable attributes object.

**Format**

```
#include <pthread.h>

int pthread_condattr_delete(
    pthread_condattr_t *attr);
```

**Parameters**

`attr`  
Condition variable attributes object that is deleted.

**Usage**

The `pthread_condattr_delete` API deletes a condition variable attributes object. Call this API when a condition variable attributes object created by `pthread_condattr_create` is no longer referred to.

This API 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>attr</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

`pthread_condattr_create`
pthread_create

Purpose
Creates a thread object and thread.

Format

```c
#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 to be run as the new thread's start routine.
- `arg` Address value copied and passed to the thread's start routine.

Usage

The `pthread_create` API creates a thread object and a thread. A thread is a single, sequential flow of control within a program. It is the active running of a designated routine, including any nested routine calls. A thread object defines and controls the running 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 ID.
- The `start_routine` API is called when this API completes successfully.

The thread is created in the ready state, and therefore might immediately begin running the routine specified by the `start_routine` parameter. The new thread begins running at its turn. With sufficient processors the new thread might run before `pthread_create` returns.

The `start_routine` 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` API (or any other mutexes or condition variables they agree to use).

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` API.
- The thread is canceled.

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 canceled, 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 API.

- 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

On 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:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EAGAIN</td>
<td>The system lacks the necessary resources to create another thread.</td>
</tr>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to create the thread object. This is not a temporary condition.</td>
</tr>
</tbody>
</table>

Related Information

Routines

pthread_attr_create  pthread_cancel  pthread_detach  pthread_exit  pthread_join
pthread_delay_np

Purpose
Causes a thread to wait for a specified period of time before continuing to run.

Format
#include <pthread.h>

int pthread_delay_np(
    struct timespec *interval);

Parameters
interval Number of seconds and nanoseconds that the calling thread waits before continuing to run. The value specified must be greater than or equal to zero.

Usage
The pthread_delay_np API causes a thread to delay running 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 zero seconds and zero nanoseconds is allowed and can result in the thread giving up the processor or delivering a pending cancelation.

The struct timespec structure contains two fields, as follows:

tv_sec Is an integer number of seconds.
tv_nsec Is an integer number of nanoseconds.

This routine can only be called by DCE applications.

Note: This API uses system time.

Return Values
If the routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by interval is not valid.</td>
</tr>
</tbody>
</table>

Related Information

Routines
pthread_yield
**pthread_detach**

**Purpose**
Marks a thread object for deletion.

**Format**

```c
#include <pthread.h>

int pthread_detach(
    pthread_t *thread);
```

**Parameters**

`thread` Thread object marked for deletion.

**Usage**

The `pthread_detach` API 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 referred to. 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>thread</code> is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>ESRCH</td>
<td>The value specified by <code>thread</code> does not refer to an existing thread.</td>
</tr>
</tbody>
</table>

**Related Information**

Routines

- `pthread_cancel`
- `pthread_create`
- `pthread_exit`
- `pthread_join`
pthread_equal

Purpose
Compares one thread identifier to another thread identifier.

Format
```
#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.

Usage
The `pthread_equal` API 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.

Return Values
If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Values of <code>thread1</code> and <code>thread2</code> do not designate the same object.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Values of <code>thread1</code> and <code>thread2</code> designate the same object.</td>
</tr>
</tbody>
</table>

Related Information

Routines
- `pthread_create`
- `pthread_self`
**pthread_exit**

**Purpose**
Terminates the calling thread.

**Format**

```
#include <pthread.h>

void pthread_exit(
    pthread_addr_t status
);
```

**Parameters**

- **status**
  Address value copied and returned to the caller of `pthread_join`.

**Usage**

The `pthread_exit` API 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 routine’s return value serves as the thread’s exit status. The process exits when the last running thread calls `pthread_exit`, with an undefined exit status.

**Return Values**

None.

**Related Information**

- **Routines**
  - `pthread_create`
  - `pthread_detach`
  - `pthread_join`
**Purpose**
Obtains a value representing a desired expiration time.

**Format**
#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.

**Usage**
The *pthread_get_expiration_np* API 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*.

The *struct timespec* structure contains two fields, as follows:

*tv_sec*  
Is an integer number of seconds.

*tv_nsec*  
Is an integer number of nanoseconds.

This routine can only be called by DCE applications.

**Note to Users**

This API uses the system time.

**Return Values**
If the routine fails, *errno* may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <em>delta</em> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

Routines
pthread_cond_timedwait
pthread_getspecific

Purpose
Obtains the thread-specific data associated with the specified key.

Format
#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.

Usage

The pthread_getspecific API obtains the thread-specific data associated with the specified key for the current thread.

Return Values

If the routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The key value is not valid.</td>
</tr>
</tbody>
</table>

Related Information

Routines

pthread_keycreate          pthread_setspecific
pthread_join

Purpose
Causes the calling thread to wait for the termination of a specified thread.

Format
#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 returns or calls pthread_exit.

Usage
The pthread_join API 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.

OSF DCE Threads allows multiple callers of the pthread_join() targeting a single thread. z/OS DCE Threads and z/OS UNIX Threads allow only one thread to join another thread.

You cannot join with a thread after the thread has been detached.

If the current thread calls this routine, a deadlock results.

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 routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by thread is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>ESRCH</td>
<td>The value specified by thread does not refer to a currently existing thread.</td>
</tr>
<tr>
<td>-1</td>
<td>EDEADLK</td>
<td>A deadlock is detected.</td>
</tr>
</tbody>
</table>

Related Information
Routines
pthread_create    pthread_detach    pthread_exit
**pthread_keycreate**

**Purpose**
Generates a unique thread-specific data key value.

**Format**

```c
#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.

**Usage**

The **pthread_keycreate** API generates a unique thread-specific data key value. This key value identifies a thread-specific data value, which is an address of storage generated by the client containing arbitrary data of any size.

Thread-specific data allow 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.

This routine generates and returns a new key value. Each call to this API within a process returns a key value that is unique within an application call. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (Refer to "pthread_once" on page 3-61 for more information.)

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 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>key</code> is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EAGAIN</td>
<td>An attempt was made to allocate a key when the key name space is exhausted. This is not a temporary condition.</td>
</tr>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to create the key.</td>
</tr>
</tbody>
</table>

**Related Information**
pthread_keycreate

Routines

pthread_getspecific   pthread_setspecific
pthread_lock_global_np

Purpose
Locks the global mutex.

Format
#include <pthread.h>

void pthread_lock_global_np( void );

Parameters
None.

Usage
The pthread_lock_global_np API 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. Assume that it is always incompatible or nonreentrant, unless the documentation for a library routine specifically states that it is compatible with multithreading.

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 can only be called by DCE applications.

Return Values
None.

Related Information

Routines
pthread_mutexattr_setkind_np        pthread_mutex_unlock        pthread_unlock_global_np
pthread_mutex_lock
Purpose
Deletes a mutex.

Format

```c
#include <pthread.h>

int pthread_mutex_destroy(
    pthread_mutex_t *mutex);
```

Parameters

- `mutex`  
  Mutex to be deleted.

Usage

The `pthread_mutex_destroy` API destroys a mutex and must be called when a mutex object is no longer referred to. The effect of calling this routine is to reclaim system storage for the mutex object. Note that if you created the mutex in dynamically allocated memory, you must reclaim that storage to avoid memory leaks.

You cannot destroy 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EBUSY</td>
<td>An attempt was made to destroy a mutex that is locked.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>mutex</code> is not valid.</td>
</tr>
</tbody>
</table>

Related Information

Routines

- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`

3-50  Application Development Reference Volumes 1 and 2
**pthread_mutex_init**

**Purpose**
Creates a mutex.

**Format**

```c
#include <pthread.h>

int pthread_mutex_init(
   pthread_mutex_t *mutex ,
   pthread_mutexattr_t attr);
```

**Parameters**

- `mutex` : Mutex to be created.
- `attr` : Attributes object that defines the characteristics of the created mutex. If you specify `pthread_mutexattr_default`, default attributes are used.

**Usage**

The `pthread_mutex_init` API initializes a mutex object and sets it to the unlocked state. If the thread that called this routine terminates, the initialized mutex is not destroyed, 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:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EAGAIN</td>
<td>The system lacks the necessary resources to initialize another mutex.</td>
</tr>
<tr>
<td>-1</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to initialize the mutex.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_mutexattr_create`
- `pthread_mutexattr_getkind_np`
- `pthread_mutexattr_setkind_np`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`
**pthread_mutex_lock**

**Purpose**
Locks an unlocked mutex.

**Format**
```c
#include <pthread.h>

int pthread_mutex_lock(
    pthread_mutex_t *mutex);
```

**Parameters**
- **mutex**: Mutex to be locked.

**Usage**
The `pthread_mutex_lock` API 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.

---

**Important Note to Users**
The following paragraph on thread priority does not apply in z/OS.

The preemption of a lower-priority thread that locks a mutex may result in the indefinite blocking of higher-priority threads waiting for the same mutex. The running of the waiting, higher-priority threads are blocked for as long as there is a sufficient number of runnable threads of any priority between the lower- and higher-priority values. Priority inversion occurs when any resource is shared between threads with different priorities.

**Return Values**
If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by mutex is invalid.</td>
</tr>
<tr>
<td>-1</td>
<td>EDEADLK</td>
<td>A deadlock condition is detected.</td>
</tr>
</tbody>
</table>

**Related Information**
Routines

- pthread_mutexattr_setkind_np
- pthread_mutex_destroy
- pthread_mutex_init
- pthread_mutex_trylock
- pthread_mutex_unlock
**pthread_mutex_trylock**

**Purpose**
Locks a mutex.

**Format**

```c
#include <pthread.h>

int pthread_mutex_trylock(
    pthread_mutex_t *mutex);
```

**Parameters**

- `mutex`  
  Mutex to be locked.

**Usage**

The `pthread_mutex_trylock` API 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, zero is returned and the thread does not wait to acquire the lock. If a fast mutex is owned by the current thread, zero 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, you must match each call to `pthread_mutex_trylock` by a call to `pthread_mutex_unlock`.)

**Return Values**

If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Successful completion. Mutex is locked or relocked.</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>The mutex is locked; therefore, it was not acquired.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>mutex</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_mutexattr_setkind_np`
- `pthread_mutex_destroy`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
**pthread_mutex_unlock**

**Purpose**
Unlocks a mutex.

**Format**
```
#include <pthread.h>

int pthread_mutex_unlock(
    pthread_mutex_t *mutex);
```

**Parameters**

*mutex*               Mutex to be unlocked.

**Usage**
The `pthread_mutex_unlock` API 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 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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <em>mutex</em> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

- `pthread_mutexattr_setkind_np`
- `pthread_mutex_destroy`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_unlock_global_np`
pthread_mutexattr_create

pthread_mutexattr_create

Purpose
Creates a mutex attributes object.

Format
#include <pthread.h>

int pthread_mutexattr_create(
    pthread_mutexattr_t *attr);

Parameters
attr
Mutex attributes object to be created.

Usage
The pthread_mutexattr_create API 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 routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ENOMEM</td>
<td>Insufficient storage exists to create the mutex attributes object.</td>
</tr>
</tbody>
</table>

Related Information

Routines

pthread_create
pthread_mutexattr_getkind_np
pthread_mutex_init
pthread_mutexattr_delete
pthread_mutexattr_setkind_np
**pthread_mutexattr_delete**

**Purpose**
Deletes a mutex attributes object.

**Format**
```
#include <pthread.h>

int pthread_mutexattr_delete(
    mutexattr_t *attr);
```

**Parameters**
- **attr**
  Mutex attributes object to be deleted.

**Usage**
The *pthread_mutexattr_delete* API deletes a mutex attributes object. Call this routine when a mutex attributes object is no longer referred to by the *pthread_mutexattr_create* API.

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 routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>attr</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**
- `pthread_mutexattr_create`
**pthread_mutexattr_getkind_np**

**Purpose**
Obtains the mutex type attribute used when a mutex is created.

**Format**
```c
#include <pthread.h>

int pthread_mutexattr_getkind_np(
    pthread_mutexattr_t attr);
```

**Parameters**
- **attr**: Mutex attributes object to be changed.

**Usage**
The `pthread_mutexattr_getkind_np` API obtains the mutex type attribute that is used when a mutex is created. See "[pthread_mutexattr_setkind_np](#) on page 3-59" for information about mutex type attributes.

This routine can only be called by DCE applications.

**Return Values**
If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex type attribute</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>attr</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**
- `pthread_mutexattr_create`
- `pthread_mutexattr_setkind_np`
- `pthread_mutex_init`
pthread_mutexattr_setkind_np

Purpose
Specifies the mutex type attribute that is used when a mutex is created.

Format
#include <pthread.h>

int pthread_mutexattr_setkind_np(
    pthread_mutexattr_t *attr,
    int kind);

Parameters
attr Mutex attributes object to be changed.
kind New value for the mutex type attribute. The kind parameter specifies the type of mutex that is created. Valid values are MUTEX_FAST (default), MUTEX_RECURSIVE, and MUTEX_NONRECURSIVE.

Usage
The pthread_mutexattr_setkind_np API 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, and can only be locked (obtained) once.

Note: Although the MUTEX_FAST attribute exists, it is not supported in z/OS. If you use this value, it defaults to a nonrecursive mutex.

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. Nonrecursive mutexes are more informative than fast mutexes. A fast mutex blocks if the thread tries to lock the mutex again, and you must determine why the thread no longer runs. 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. No other thread can satisfy the condition of the predicate.

This routine can only be called by DCE applications.

Return Values
If the routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by attr or kind is not valid.</td>
</tr>
<tr>
<td>-1</td>
<td>EPERM</td>
<td>The caller does not have the appropriate privileges.</td>
</tr>
<tr>
<td>-1</td>
<td>ERANGE</td>
<td>One or more parameters supplied have an invalid value.</td>
</tr>
</tbody>
</table>

Related Information
Routines

pthread_mutexattr_create  pthread_mutexattr_getkind_np  pthread_mutex_init
pthread_once

Purpose
Calls an initialization routine run by one thread, a single time.

Format
#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.
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.

Usage
The pthread_once API calls an initialization routine run 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-specific 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 ran once. If it has not run, this routine calls the initialization routine specified in init_routine. If the client one-time initialization code ran 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 run 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 routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>Invalid parameter.</td>
</tr>
</tbody>
</table>
**Purpose**
Obtains the identifier of the current thread.

**Format**
```c
#include <pthread.h>
pthread_t pthread_self( void );
```

**Parameters**
None.

**Usage**
The `pthread_self` API allows a thread to obtain its own identifier. For example, this identifier allows a thread to cancel itself.

This value becomes meaningless when the thread object is deleted, for example, 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**
Routines
- `pthread_create`
- `pthread_setprio`
- `pthread_setscheduler`
**pthread_setasynccancel**

**Purpose**
Allows the current thread to be canceled asynchronously.

**Format**
```c
#include <pthread.h>
int pthread_setasynccancel(
    int state );
```

**Parameters**

- `state`
  State of asynchronous cancelation set for the calling thread. On return, receives the prior state of asynchronous cancelation. Valid values are as follows:
  - `CANCEL_ON`: Thread can be canceled asynchronously.
  - `CANCEL_OFF`: Thread cannot be canceled asynchronously.

**Usage**

The `pthread_setasynccancel` API enables or disables the ability of the current thread to be canceled asynchronously and returns the previous cancelation state to the `state` parameter.

When `CANCEL_OFF` is set by `pthread_setcancel` a cancelation cannot be delivered to the thread, even if a cancelable API is called or `pthread_setasynccancel` is enabled. When `CANCEL_ON` is set by `pthread_setcancel` the cancelation depends on the state of the `pthread_setasynccancel` parameter.

When `pthread_setcancel` is set to `CANCEL_ON` and `pthread_setasynccancel` is set to `CANCEL_OFF`, the thread can only be canceled at specific cancelation points (for example, condition waits, thread joins, or calls to `pthread_testcancel`). If both cancelation APIs are set to `CANCEL_ON`, the thread can be canceled at any point in its execution.

When a thread is created, the default state for asynchronous cancels is `CANCEL_OFF`.

If you call this routine to enable an asynchronous cancelation, call it in a region of code where asynchronous delivery of cancelations is disabled by a previous call to this routine. Do not call threads APIs in regions of code where asynchronous delivery of cancelations is enabled. The previous state of asynchronous delivery can be restored later by another call to this routine.

**Note:** For RPC calls, do not enable asynchronous cancels such as setting `CANCEL_ON` using this API as results may be indeterminate.

**Return Values**

On successful completion, the previous asynchronous cancelation state is returned. If the routine fails, -1 is returned, and `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL_ON</td>
<td></td>
<td>Thread can be canceled asynchronously.</td>
</tr>
<tr>
<td>CANCEL_OFF</td>
<td></td>
<td>Thread cannot be canceled asynchronously.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The specified state is not <code>CANCEL_ON</code> or <code>CANCEL_OFF</code>.</td>
</tr>
</tbody>
</table>

**Related Information**
**pthread_setasynccancel**

**Routines**

<table>
<thead>
<tr>
<th>pthread_cancel</th>
<th>pthread_setcancel</th>
<th>pthread_testcancel</th>
</tr>
</thead>
</table>

3-64  Application Development Reference Volumes 1 and 2
pthread_setcancel

Purpose
Allows the current thread to be canceled generally.

Format
#include <pthread.h>

int pthread_setcancel(
    int state);

Parameters
state State of general cancelation set for the calling thread. On return, receives the prior state of
general cancelation. Valid values are as follows:

CANCEL_ON Thread can be canceled generally.

CANCEL_OFF Thread cannot be canceled generally.

Usage
The pthread_setcancel API enables or disables the current thread’s ability to be canceled and returns the previous
cancelation state to the state parameter.

When pthread_setcancel is set to CANCEL_OFF, a cancelation cannot be delivered to the thread, even if a cancelable
routine is called or pthread_setasynccancel is enabled.

When a thread is created, the default state for asynchronous cancels is CANCEL_ON.

Possible Dangers of Not Allowing Cancelations
The most important use of cancelations 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 pthread_setcancel is disabled, no routine can be canceled. Waits must be completed normally before a
cancelation can be delivered. As a result, the program stops working and the user is unable to cancel the operation.

When using the CANCEL_OFF value make sure that no long waits can occur or that it is necessary for other reasons to defer
cancelations around that particular region of code.

Return Values
On successful completion, the previous cancelation state is returned. If the routine fails, -1 is returned, and errno may be set
to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL_ON</td>
<td></td>
<td>Thread can be canceled generally.</td>
</tr>
<tr>
<td>CANCEL_OFF</td>
<td></td>
<td>Thread cannot be canceled generally.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The specified state is not CANCEL_ON or CANCEL_OFF.</td>
</tr>
</tbody>
</table>

Related Information

Chapter 3. Threads 3-65
pthread_setcancel

Routines

pthread_cancel  pthread_setasynccancel  pthread_testcancel
**pthread_setspecific**

**Purpose**
Sets the thread-specific data associated with the specified key for the current thread.

**Format**
```c
#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.

**Usage**
The `pthread_setspecific` API 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 storage that are reserved for use by the calling thread.

**Return Values**
If the routine fails, -1 is returned, and `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The key value is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**
- `pthread_getspecific`
- `pthread_keycreate`
**pthread_signal_to_cancel_np**

**Purpose**
Cancels the specified thread.

**Format**

```c
#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**
  Thread canceled if a valid signal is received by the process.

**Usage**
The `pthread_signal_to_cancel_np` API 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` argument is not validated and if it is not valid, this routine will return successfully but neither the specified thread nor the previously specified thread will be 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 supersede the thread specified in the previous call. That thread will not be canceled if one of the signals specified for it is delivered to the process. Be careful when you call this routine; if another thread calls it after you do, the expected result of this routine will not occur.

This routine can only be called by DCE applications.

**Return Values**

If the routine fails, `errno` may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by <code>thread</code> is not valid.</td>
</tr>
</tbody>
</table>

**Related Information**

**Routines**

`pthread_cancel`
pthread_testcancel

Purpose
Requests delivery of a pending cancelation to the current thread.

Format
#include <pthread.h>

void pthread_testcancel( void );

Parameters
None.

Usage
The pthread_testcancel API requests delivery of a pending cancelation to the current thread. The cancelation is delivered only if a cancelation is pending for the current thread and general cancelation delivery is not currently disabled. (A thread disables delivery of cancelations to itself by calling pthread_setcancel.)

This routine, when called within very long loops, ensures that a pending cancelation is noticed within a reasonable amount of time.

Return Values
None.

Related Information

Routines
pthread_cancel  pthread_setasynccancel  pthread_setcancel
**pthread_unlock_global_np**

**Purpose**
Unlocks a global mutex.

**Format**
```c
#include <pthread.h>
void pthread_unlock_global_np( void );
```

**Parameters**
None.

**Usage**
The `pthread_unlock_global_np` API 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 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 can only be called by DCE applications.

**Return Values**
None.

**Related Information**

**Routines**
- `pthread_lock_global_np`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutexattr_setkind_np`
pthread_yield

Purpose
Notifies the scheduler that the current thread is willing to release its processor to other threads of the same priority.

Format
#include <pthread.h>

void pthread_yield( void );

Parameters
None.

Usage
The pthread_yield API 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.)

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.

Use pthread_yield carefully because misuse causes unnecessary context switching, which increases overhead without increasing fairness. For example, it is counter-productive for a thread to yield while it has a needed resource locked.

Return Values
None.

Related Information
Routines

pthread_attr_setsched           pthread_setscheduler
sigwait

Purpose
Causes a thread to wait for an asynchronous signal.

Format
#include <pthread.h>

int sigwait(
    sigset_t *set);

Parameters

set  Set one of asynchronous pending signals from which this routine chooses one signal on which
     the calling thread will wait.

Usage
The sigwait API causes a thread to wait for an asynchronous signal by choosing a pending signal from set, atomically clearing
it from the system’s set of pending signals and returning 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.

If more than one thread is using this routine to wait for the same signal, only one thread will return from this routine with the
signal number.

Return Values
If the routine fails, errno may be set to one of the following values:

<table>
<thead>
<tr>
<th>Return</th>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal number</td>
<td></td>
<td>Successful completion.</td>
</tr>
<tr>
<td>-1</td>
<td>EINVAL</td>
<td>The value specified by set is not valid.</td>
</tr>
</tbody>
</table>

Related Information

Routines
pthread_cancel        pthread_setasynccancel
Chapter 4. Directory Services

DCE Directory Services provide application developers with the XDS, XOM, and Lightweight Directory Access Protocol (LDAP) API for accessing the data in the CDS. This chapter introduces the XDS and XOM interfaces and discusses them in detail. Information about the LDAP programming interface resides in the z/OS SecureWay Security Server LDAP Client Programming.
XDS Routines

This section describes the X/Open Directory Services (XDS) routines. XDS provides a C language binding.

Note to Users

In z/OS DCE, the XDS interface does not support access to the Global Directory Service (GDS). However, it does support access to the Cell Directory Service (CDS). Any XDS interface routine called with a completely X.500 typed name returns with a DS_C_SERVICE_ERROR (DS_E_UNAVAILABLE) error.

This interface comprises a number of routines whose descriptions are summarized in the following table.

Note: The term Directory refers to a GDS Directory Information Tree (DIT) and to a CDS namespace.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds_abandon</td>
<td>API not supported</td>
</tr>
<tr>
<td>dsX_extract_attr_values</td>
<td>Extracts attribute values from an OM object.</td>
</tr>
<tr>
<td>ds_add_entry</td>
<td>Adds a leaf entry to the Directory.</td>
</tr>
<tr>
<td>ds_bind</td>
<td>Open a session with a directory user agent (DUA).</td>
</tr>
<tr>
<td>ds_compare</td>
<td>Compare a purported attribute value with the attribute value stored in the directory for a particular entry</td>
</tr>
<tr>
<td>ds_initialize</td>
<td>Initialize the interface.</td>
</tr>
<tr>
<td>ds_list</td>
<td>Enumerate the immediate subordinates of a particular directory entry.</td>
</tr>
<tr>
<td>ds_modify_entry</td>
<td>Perform an atomic modification of a directory entry.</td>
</tr>
<tr>
<td>ds_read</td>
<td>Query information on a directory entry by name.</td>
</tr>
<tr>
<td>ds_remove_entry</td>
<td>Remove a leaf entry from the Directory.</td>
</tr>
<tr>
<td>ds_shutdown</td>
<td>Shutdown the interface.</td>
</tr>
<tr>
<td>ds_unbind</td>
<td>Unbind from a directory session.</td>
</tr>
<tr>
<td>ds_version</td>
<td>Negotiate features of the interface and service.</td>
</tr>
</tbody>
</table>

The Distributed Computing Environment (DCE) XDS interface does not support asynchronous operations. Therefore, ds_abandon and ds_receive_result are not supported. A ds_abandon call returns with a DS_C_ABANDON_FAILED (DS_E_TOO_LATE) error. A ds_receive_result call returns with DS_Status return code set to DS_SUCCESS, and the completion_flag_return parameter set to DS_NO_OUTSTANDING_OPERATION.

The following differences exist between (GDS) and (CDS):

- The XDS interface routines do not operate on the GDS Directory Information Tree.
- The ds_modify_rdn and ds_search routines are not supported by CDS, and are, therefore, not supported in z/OS DCE. The error message DS_C_SERVICE_ERROR (DS_E_UNWILLING_TO_PERFORM) is returned if either of these two routines are attempted on CDS.
- In CDS, an application can only manipulate CDS objects and soft-links, such as aliases. All other CDS entry types such as directory and child pointers can only be manipulated via the CDS control/administration program, (cdscp) or the DCE control program (dcecp).
- Prior to manipulating CDS entries, the user or application programmer must be DCE-logged in; otherwise, an error is returned.
- In CDS, access control is on a per entry basis, not on a per attribute basis as in GDS.
- In CDS, no X.500 schema rules apply. There is:
  - No concept of an object class in the X.500 sense. CDS objects can have an object class attribute, but the value of the attribute is application defined.
  - No mandatory set of 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 namespace is different from the GDS Directory Information Tree (DIT). It 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 namespace 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, and ds_compare (with this attribute) returns with DS_C_ATTRIBUTE_ERROR (DS_E_CONSTRAINT_VIOLATION).

XDS functions check for NULL pointers at the function interface, and return an error. The check is only for NULL and not for validity. If invalid pointers are passed this may result in undetermined behavior.
ds_add_entry

Purpose
Adds a leaf entry to the Directory.

Format
#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_SIZE_LIMIT and DS_DONT_DEREFERENCE_ALIASES 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. Any aliases in the name are not dereferenced.
    entry (Object(DS_C_ATTRIBUTE_LIST)). The attribute information that constitutes the entry to be created. Note that an instance of Object Management (OM) class DS_C_ENTRY_INFORMATION can be specified as the value of this parameter, because DS_C_ENTRY_INFORMATION is a subclass of DS_C_ATTRIBUTE_LIST.

Output
    invoke_id_return (integer). Not supported.

Usage
The ds_add_entry API adds a leaf entry to the Directory. The entry can be either an object or an alias.

Although you are not aware of whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. See "XDS Routines" on page 4-2 for the general differences between operations on GDS and CDS.
For the \texttt{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.
- Only the \texttt{DS\_A\_COMMON\_NAME} and \texttt{DS\_A\_MEMBER} attributes are valid for the \texttt{DS\_O\_GROUP\_OF\_NAMES} object in CDS.
- GDS-structured \textit{attribute types} are not supported by CDS. If an attempt is made to add a GDS-structured attribute type to CDS, the interface returns with a \texttt{DS\_C\_ATTRIBUTE\_ERROR (DS\_E\_CONSTRAINT\_VIOLATION)}.

Because CDS does not implement the X.500 schema rules, some CDS objects may not contain mandatory attributes such as object class.

\section*{Return Values}

\texttt{DS\_status DS\_SUCCESS} is returned if the entry was added; otherwise, an error is returned.

\section*{Errors}

This routine can return the following errors:

- \texttt{DS\_C\_SYSTEM\_ERROR}
- The following \texttt{DS\_C\_LIBRARY\_ERRORS}:
  - \texttt{DS\_E\_BAD\_ARGUMENT}
  - \texttt{DS\_E\_BAD\_CONTEXT}
  - \texttt{DS\_E\_BAD\_NAME}
  - \texttt{DS\_E\_BAD\_SESSION}
  - \texttt{DS\_E\_MISCELLANEOUS}
  - \texttt{DS\_E\_MISSING\_TYPE}
  - \texttt{DS\_E\_TOO\_MANY\_OPERATIONS}
- The following directory errors:
  - \texttt{DS\_C\_ATTRIBUTE\_ERROR}
  - \texttt{DS\_C\_NAME\_ERROR}
  - \texttt{DS\_C\_REFERRAL}
  - \texttt{DS\_C\_SECURITY\_ERROR}
  - \texttt{DS\_C\_SERVICE\_ERROR}
  - \texttt{DS\_C\_UPDATE\_ERROR}
- \texttt{DS\_C\_COMMUNICATIONS\_ERROR}
- The following error constant:
  - \texttt{DS\_INVALID\_WORKSPACE} (if \texttt{DS\_DEFAULT\_SESSION} is used)
  - \texttt{DS\_NO\_WORKSPACE}

\section*{Related Information}

Routines

\texttt{ds\_read ds\_remove\_entry ds\_modify\_entry}

Books

- \texttt{z/OS DCE Application Development Guide: Directory Services}
ds_bind

Purpose
Opens a session with the Directory.

Format
#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 routine 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 routines (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 built from the session parameter instance, specified as a parameter. The routine specifies default values for any of the OM attributes that are not present in the session parameter instance specified as a parameter.

It also sets the value of the DS_FILE_DESCRIPTOR OM attribute to DS_NO_VALID_FILE_DESCRIPTOR because the functionality is not supported.

Usage
The ds_bind API sets up a communication link to the directory.

Although you normally are not aware of whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. See page 4-2 for the general differences between operations on GDS and CDS.

Return Values
DS_status DS_SUCCESS is returned if the routine completed successfully; otherwise, an error is returned.

Errors
This routine can return the following errors:

- DS_C_SYSTEM_ERROR
- The following DS_C_LIBRARY_ERRORS:
  - DS_E_BAD_SESSION
  - DS_E_BAD_WORKSPACE
The following directory errors:
- DS_C_SECURITY_ERROR
- DS_C_SERVICE_ERROR
- DS_C_COMMUNICATIONS_ERROR

The following error constant:
- DS_INVALID_WORKSPACE

Related Information

Routines
ds_unbind

Books
- z/OS DCE Application Development Guide: Directory Services
ds_compare

Purpose
Compares a purported attribute value with the attribute value stored in the Directory for a particular entry.

Format
```c
#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 distinguished name (DN) of the target object if an alias is dereferenced.
- **invoke-id_return** (integer). Not supported.

Usage
The **ds_compare** API compares the value specified in the given **ava** parameter with the value or values of the same attribute type in the named entry.

Although you normally are not aware of whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. See page 4-2 for the general differences between operations on GDS and CDS.

For the **ds_compare** operation:
- In CDS, the naming attribute of an object is not stored in the attribute list of an object. Therefore 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 a DS_C_ATTRIBUTE_ERROR (DS_E_CONSTRAINT_VIOLATION).
- When using CDS, if you use **ds_compare** on a CDS entry that is not an object, such as a Directory entry, a DS_C_NAME_ERROR (DS_E_NO_SUCH_OBJECT) is returned.
Return Values

**DS_status**

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

This routine can return the following errors:

- **DS_C_SYSTEM_ERROR**
- The following **DS_C_LIBRARY_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:
  - **DS_C_ATTRIBUTE_ERROR**
  - **DS_C_NAME_ERROR**
  - **DS_C_REFERRAL**
  - **DS_C_SECURITY_ERROR**
  - **DS_C_SERVICE_ERROR**
- **DS_C_COMMUNICATIONS_ERROR**
- The following error constant:
  - **DS_INVALID_WORKSPACE** (if **DS_DEFAULT_SESSION** is used)
  - **DS_NO_WORKSPACE**

Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services](#)
ds_initialize

ds_initialize

Purpose
Initializes the XDS interface.

Format
#include <xom.h>
#include <xds.h>

OM_workspace ds_initialize(
    void);

Parameters
None.

Usage
The ds_initialize API performs any necessary initialization of the XDS API including the creation of a workspace. It must be called before any other directory interface routines are called. It may be called multiple times; in which case, each call returns a workspace that is distinct from other workspaces created by ds_initialize but not yet deleted by ds_shutdown.

Return Values
OM_workspace
Upon successful completion, the return code 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 routines which use objects from the workspace.

Errors
This routine returns NULL if in error.

Related Information
Routines
ds_shutdown
ds_list

Purpose
Enumerates the immediate subordinates of a particular directory entry.

Format
```
#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 prohibited by the DS_DONT_DEREFERENCE_ALIASES service control attribute 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 distinguished name 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. The outcome qualifier also explains the reason (for example, because the time limit expired), and it contains information that can be helpful when you are attempting to complete the operation.
- **invoke_ID_return** (integer). Not supported.

Usage

The ds_list API 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**

  - **DS_SUCCESS** is returned if the named object is located (even if there are no subordinates), otherwise an error returned.

Note: In CDS, all objects, soft-links, and child-pointers will be returned in the result. Soft-links, as with aliases in GDS, are not dereferenced.
<any text="ds_list">

Errors

This routine can return the following errors:

- **DS_C_SYSTEM_ERROR**
- The following **DS_C_LIBRARY_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:
  - **DS_C_NAME_ERROR**
    - if the name specified is that of a CDS directory object (as opposed to the name of a CDS directory) a **DS_C_NAME_ERROR (DS_E_NO_SUCH_OBJECT)** is returned.
  - **DS_C_REFERRAL**
  - **DS_C_SECURITY_ERROR**
  - **DS_C_SERVICE_ERROR**
- **DS_C_COMMUNICATIONS_ERROR**
- The following error constant:
  - **DS_INVALID_WORKSPACE** (if **DS_DEFAULT_SESSION** is used)
  - **DS_NO_WORKSPACE**

Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services](#)
ds MODIFY ENTRY

Purpose
Performs an atomic modification on a directory entry.

Format
#include <xom.h>
#include <xsds.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.

Usage
The ds MODIFY ENTRY API 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 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, 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.

Although you normally are not aware of whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. See page 4-2 for the general differences between operations on GDS and CDS.

For the ds MODIFY ENTRY operation:
- Naming schema rules do not apply in CDS. Thus, the following attribute errors are never returned when accessing CDS:
  - DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE
  - DS_E_ATTRIBUTE_OR_VALUE_ALREADY_EXISTS
ds_modify_entry

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.

Return Values

**DS_status**

DS_SUCCESS is returned if all the modifications succeeded, otherwise an error is returned.

Errors

This routine can return the following errors:

- **DS_C_SYSTEM_ERROR**
- The following DS_C_LIBRARY_ERRORS:
  - DS_E_BAD_ARGUMENT
  - DS_E_BAD_CONTEXT
  - DS_E_BAD_NAME
  - DS_E_BAD_SESSION
  - DS_E_MISC
  - DS_E_MISSING_TYPE
  - DS_E_TOO_MANY_OPERATIONS
- 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
- **DS_C_COMMUNICATIONS_ERROR**
- The following error constant:
  - DS_INVALID_WORKSPACE (if DS_DEFAULT_SESSION is used)
  - DS_NO_WORKSPACE

Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services](#)
ds_read

Purpose
Queries information on an entry by name.

Format

```c
#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 [z/OS DCE Application Development Guide: Directory Services](see the Part entitled “XDS Class Definitions”).

**Output**

- `result_return` (Object(DS_C_READ_RESULT)). Upon successful completion, the result contains any requested attribute types and values, and a flag indicating whether the result came from the original entry or a copy. Attribute information is only returned if access rights are sufficient.
- `invoke_id_return` (integer). Not supported.

Usage

The ds_read API is used to extract information from an explicitly named entry.

Although you normally are not aware of whether naming operations are being handled by GDS or CDS, there are some situations where naming results can differ between the two services. See page 4-2 for the general differences between operations on GDS and CDS.

For the ds_read operation:

- Because CDS does not implement the X.500 schema rules, some CDS objects may not contain normally mandatory attributes like object class. In CDS, a read of an aliased object fails if the DS_A_ALIASED_OBJECT_NAME attribute 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.

Chapter 4. Directory Services 4-15
ds_read

- In CDS, access control is on a per object basis. Read access to the object is all that is required to read all of its attributes.

Return Values

DS_status Indicates whether or not the read operation is completed. This is indicated by a return value of DS_SUCCESS; otherwise, an error is returned.

Errors

This routine can return the following errors:

- DS_C_SYSTEM_ERROR
- The following DS_C_LIBRARY_ERRORS:
  - DS_E_BAD_ARGUMENT
  - DS_E_BAD_ATTRIBUTE
  - DS_E_BAD_CONTEXT
  - DS_E_BAD_NAME
  - DS_E_BAD_SESSION
  - DS_E_MISC
  - DS_E_MISSING_TYPE
  - DS_E_TOO_MANY_OPERATIONS
- The following directory errors:
  - DS_C_ATTRIBUTE_ERROR
  - DS_C_NAME_ERROR
  - DS_C_REFERRAL
  - DS_C_SECURITY_ERROR
    Note: A DS_C_SECURITY_ERROR (DS_E_INSUFFICIENT_ACCESS_RIGHTS) is only reported where access rights prevent the reading of all requested attribute values. For z/OS DCE this implies that access to the object is denied.
  - DS_C_SERVICE_ERROR
- DS_C_COMMUNICATIONS_ERROR
- The following error constant:
  - DS_INVALID_WORKSPACE (if DS_DEFAULT_SESSION is used)
  - DS_NO_WORKSPACE

Related Information

Books

- z/OS DCE Application Development Guide: Directory Services
Purpose
Removes a leaf entry from the Directory.

Format
```c
#include <xom.h>
#include <xdst.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.

Usage
The ds_remove_entry API 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. The status is DS_SUCCESS if entry was deleted; otherwise, an error is returned.

Errors
This routine can return the following errors:
- **DS_C_SYSTEM_ERROR**
- The following DS_C_LIBRARY_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:
  - DS_C_NAME_ERROR
  - DS_C_REFERRAL
  - DS_C_SECURITY_ERROR
ds_remove_entry

- DS_C_SERVICE_ERROR
- DS_C_UPDATE_ERROR

- DS_C_COMMUNICATIONS_ERROR
- The following error constant:
  - DS_INVALID_WORKSPACE (if DS_DEFAULT_SESSION is used)
  - DS_NO_WORKSPACE

Related Information

Books
- z/OS DCE Application Development Guide: Directory Services
ds_search

Purpose
Finds selected information from Directory entries. This API is not supported by CDS, and therefore is not supported in z/OS DCE.

Format
#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 prohibited by the DS_DONT_DEREFERENCE_ALIASES service control attribute of the DS_C_CONTEXT object.
subset (Integer). Specifies the portion of the Directory to be searched. Its value must be one of the following:
   DS_BASE_OBJECT Search just the given object entry.
   DS_ONE_LEVEL Search just the immediate subordinates of the given object entry.
   DS_WHOLE_SUBTREE Search the given object and all its subordinates.
filter (Object(DS_C_FILTER)). The filter is used to eliminate unwanted entries from the search. 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. Using this constant corresponds to using 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 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 may not be.

Output
result_return (Object(DS_C_SEARCH_RESULT)). On successful completion, 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 the reason it is not complete and how it could be completed.

invoke_id_return (integer). Not supported.
ds_search

Usage

The **ds_search** API 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 returns with **DS_C_SERVICE_ERROR (DS_E_UNWILLING_TO_PERFORM)**.

Return Values

**DS_status**  
**DS_SUCCESS** is returned if the named object is located, otherwise an error is returned.

Errors

This routine can return the following errors:

- **DS_C_SYSTEM_ERROR**
- The following **DS_C_LIBRARY_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:
  - **DS_C_ATTRIBUTE_ERROR**
  - **DS_C_NAME_ERROR**
  - **DS_C_REFERRAL**
  - **DS_C_SECURITY_ERROR**
    
    **Note:** A **DS_C_SECURITY_ERROR (DS_E_INSUFFICIENT_ACCESS_RIGHTS)** is only reported where access rights prevent the reading of all requested attribute values.

- **DS_C_SERVICE_ERROR**

- **DS_C_COMMUNICATIONS_ERROR**
- The following error constant:
  - **DS_INVALID_WORKSPACE** (if **DS_DEFAULT_SESSION** is used)
  - **DS_NO_WORKSPACE**

**Note:** An unfiltered search of just the base object succeeds even if none of the requested attributes is found, whereas the **ds_read** call fails with the same selected attributes.

Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services](#)
ds_shutdown

Purpose
Deletes a directory workspace.

Format
#include <xom.h>
#include <xds.h>

DS_status ds_shutdown(
    OM_workspace workspace );

Parameters
workspace Specifies the workspace (obtained from a call to ds_initialize) that is to be deleted.

Usage
The ds_shutdown API deletes a workspace established by ds_initialize, and enables the service to release resources associated with that workspace. All sessions associated with the workspace must be terminated by calling ds_unbind before calling ds_shutdown. No other directory routine can refer to the specified workspace after it is deleted.

Return Values
DS_status DS_SUCCESS is returned if the named object is located, otherwise an error is returned.

Errors
This routine can return the following error constant:

- DS_INVALID_WORKSPACE

This routine does not return a Communications-Error or any directory errors.

Related Information
Routines
ds_initialize
ds_unbind

Purpose
Ends a directory session started by ds_bind.

Format
#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 routine succeeds. The remaining OM attributes are unchanged.

Usage
The ds_unbind API terminates the given directory session, and makes the parameter unavailable for use with other interface routines (except ds_bind).

The unbound session can be used again as a parameter to ds_bind possibly after modification by the Object Management routines. When it is no longer required, it must be deleted using the Object Management routines.

Return Values
DS_status DS_SUCCESS is returned if the session parameter is unbound, otherwise an error is returned.

Errors
This routine can return the following errors:

- DS_C_SYSTEM_ERROR
- The following DS_C_LIBRARY_ERRORS:
  - DS_E_BAD_SESSION
  - DS_E_MISCELLANEOUS
- The following error constant:
  - DS_INVALID_WORKSPACE (if DS_DEFAULT_SESSION has been specified.)
  - DS_NO_WORKSPACE (if DS_DEFAULT_SESSION has not been specified.)

Related Information

Routines
ds_bind
Books

- z/OS DCE Application Development Guide: Directory Services
ds_version

Purpose
Negotiates features of the interface and service.

Format
#include <xom.h>
#include <xds.h>

DS_status ds_version(
    DS_feature feature_list[],
    OM_workspace workspace);

Parameters
Input
feature_list  (DS_feature). An ordered sequence of features, each represented by an object identifier.
workspace Specifies the workspace (obtained from a call to ds_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.

Usage
The ds_version API negotiates features of the directory interface (which are represented by Object Identifiers) and associates them with a workspace. The DS_BASIC_DIR_CONTENTS_PKG and the DS_STRONG_AUTHENT_PKG specified in [z/OS DCE Application Development Guide: Directory Services] are two of the negotiable features in this specification. For a list of supported packages, see the “Packages” section in the “XOM Programming” section of the [z/OS DCE Application Development Guide: Directory Services].

Return Values
DS_status DS_SUCCESS is returned if the routine completed successfully; otherwise, an error is returned.
activated (OM boolean). If the routine completed successfully, this return value contained in the feature_list parameter indicates whether or not a particular requested feature is available in the interface. 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;

Errors
This routine can return the following errors:

- **DS_C_SYSTEM_ERROR**
- The following **DS_C_LIBRARY_ERROR**:
  - DS_E_BAD_WORKSPACE
  - DS_E_MISCELLANEOUS
- The following error constant:
– DS_INVALID_WORKSPACE

Related Information

Books

• z/OS DCE Application Development Guide: Directory Services
Purpose
Extracts attribute values from an OM object.

Format
#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. For further information on local
strings please refer to the z/OS DCE Application Development Guide: Directory Services.

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.

Usage
The dsX_extract_attr_values() function 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() function call.

Return Values
OM_return_code Indicates whether the function succeeded and, if not, why not. If the function is successful, the return
code is set to OM_SUCCESS; if the function fails, it has one of the error values listed in the xom.h header
file (see xom.h on page 4-95).

Error Conditions
Refer to xom.h for a list of possible error values that can be returned in OM_return_code.
**xds.h**

**Purpose**
Contains declarations for the XDS routines and definitions for object identifiers defined by the Directory Service Package.

**Format**
```
#include <xom.h>
#include <xds.h>
```

**Usage**
The `xds.h` header file is a mandatory include file for all applications using the XDS API. It declares the interface routines, the structures passed to and from those routines, and the defined constants used by the routines and structures.

All application programs that include this header must first include the `xom.h` Object Management header.

The following shows the contents of the `xds.h` header file.

```c
#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(qzerodot) } */
#define OMP_O_DS_SERVICE_PKG "\x2B\xqzerodotC\xqzerodot2\x87\x73\x1C\xqzerodotqzerodot"

/* Typedefs */
typedef OM_private_object DS_status;

typedef struct {
   OM_object_identifier feature;
   OM_boolean activated;
} DS_feature;

/* API 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
);
```

Chapter 4. Directory Services 4-27
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
);

/* Defined constants */

/* Intermediate object identifier macro */
#define dsP_c(X) (OMP_O_DS_SERVICE_PKG #X)

/* OM class names (prefixed DS_C_) */

/* Every application program that 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_COMPARE_RESULT dsP_c(\x85\x46)
#define OMP_O_DS_C_CONTEXT dsP_c(\x85\x47)
#define OMP_O_DS_C_CONTINUATION_REF dsP_c(\x85\x48)
#define OMP_O_DS_C_CONTINUATION_LIST 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_ENTRY_MOD_SELECTION dsP_c(\x85\x50)
#define OMP_O_DS_C_ERROR dsP_c(\x85\x51)
#define OMP_O_DS_C_ERROR_LIST dsP_c(\x85\x52)
#define OMP_O_DS_C_ERROR_SELECTION dsP_c(\x85\x53)
#define OMP_O_DS_C_ERROR_PROBLEM dsP_c(\x85\x54)
#define OMP_O_DS_C_ERROR_PROBLEM_LIST dsP_c(\x85\x55)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION dsP_c(\x85\x56)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST dsP_c(\x85\x57)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST dsP_c(\x85\x58)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST dsP_c(\x85\x59)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST dsP_c(\x85\x5A)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST_LIST dsP_c(\x85\x5B)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST_LIST_LIST dsP_c(\x85\x5C)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST_LIST_LIST_LIST dsP_c(\x85\x5D)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST_LIST_LIST_LIST_LIST dsP_c(\x85\x5E)
#define OMP_O_DS_C_ERROR_PROBLEM_SELECTION_LIST_LIST_LIST_LIST_LIST_LIST_LIST_LIST_LIST dsP_c(\x85\x5F)
#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_ALIAS_DEREFERENCED ((OM_type) 704)
#define DS_ALIAS_ENTRY ((OM_type) 705)
#define DS_ALL_ATTRIBUTES ((OM_type) 706)
#define DS_ASYNCHRONOUS ((OM_type) 707)
#define DS_ATTRIBUTES ((OM_type) 708)
#define DS_ATTRIBUTES_SELECTED ((OM_type) 709)
#define DS_ATTRIBUTE_TYPE ((OM_type) 710)
#define DS_ATTRIBUTE_VALUE ((OM_type) 711)
#define DS_ATTRIBUTE_VALUES ((OM_type) 712)
#define DS_AUTOMATIC_CONTINUATION ((OM_type) 713)
#define DS_AVAS ((OM_type) 714)
#define DS_CHAINING_PROHIB ((OM_type) 715)
#define DS_CHANGES ((OM_type) 716)
#define DS_CRIT ((OM_type) 717)
#define DS_CRT ((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_FILTER_ITEMS ((OM_type) 727)
#define DS_FILTER_TYPE ((OM_type) 728)
#define DS_FINAL_SUBSTRING ((OM_type) 729)
#define DS_FROM_ENTRY ((OM_type) 730)
#define DS_IDENT ((OM_type) 731)
#define DS_INFO_TYPE ((OM_type) 732)
#define DS_INITIAL_SUBSTRING ((OM_type) 733)
#define DS_ITEM_PARAMETERS ((OM_type) 734)
#define DS_LIMIT_PROBLEM ((OM_type) 735)
#define DS_ITEM_PARAMTERS ((OM_type) 736)
#define DS_INITIAL_SUBSTRING ((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)

define 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
    };

#define DS_Filter_Type

    enum DS_Filter_Type {
        DS_ITEM = 0,
        DS_AND = 1,
        DS_OR = 2,
        DS_NOT = 3
    };

#define DS_Information_Type

    enum DS_Information_Type {
        DS_TYPES_ONLY = 0,
        DS_TYPES_AND_VALUES = 1
    };

#define 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
};

/* DSModification_Type */
enum DSModification_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_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
};

/* 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 (no asynchronous operations) */
#define DS_MAX_OUTSTANDING_OPERATIONS 0

/* asynchronous event posting */
#define DS_NO_VALID_FILE_DESCRIPTOR -1
xds.h

#include <xds.h>

#define 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)*
- [z/OS DCE Application Development Guide: Directory Services](#)
xdsbdcp.h

Purpose
Contains definitions for Basic Directory Contents Package.

Format
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>

Usage
The xdsbdcp.h header defines the object identifiers of directory attribute types and object classes supported by the DS_BASIC_DIRECTORY_CONTENTS_PKG. 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 “Basic Directory Contents Package” chapter of the z/OS DCE Application Development Guide: Directory Services.

The following shows the contents of the xdsbdcp.h header file:

#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
#define dsP_attributeType(X) ("\x55\x4" #X) /* joint-iso-ccitt 5 4 */
#endif

#ifndef dsP_objectClass
#define dsP_objectClass(X) ("\x55\x6" #X) /* joint-iso-ccitt 5 6 */
#endif

#define dsP_bdcp_c(X) (OMP_O_DS_BASIC_DIR_CONTENTS_PKG #X)

/* OM class names (prefixed DS_C_), */
/* Directory attribute types (prefixed DS_A_), */
/* and Directory object classes (prefixed DS_O_) */

/* Every application program that 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 OPM_O prefix to denote that they are OM classes. */
/* However, when using the OM_IMPORT and OM_EXPORT macros, */
/ xdsbdcp.h

/* 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_INTERNAL_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(\x08)
#define OMP_O_DS_A_OWNER dsP_attributeType(\x20)
#define OMP_O_DS_A_PHONE_NBR dsP_attributeType(\x14)
#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(\x10)
#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_ASEE_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_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(\x0A)
#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) 8qzerodot1)
#define DS_COUNTRY_CODE ((OM_type) 8qzerodot2)
#define DS_CRITERIA ((OM_type) 8qzerodot3)
#define DS_OBJECT_CLASS ((OM_type) 8qzerodot4)
#define DS_PARAMETERS ((OM_type) 8qzerodot5)
#define DS_POSTAL_ADDRESS ((OM_type) 8qzerodot6)
#define DS_PHONE_NBR ((OM_type) 8qzerodot7)
#define DS_TELETEX_TERM ((OM_type) 8qzerodot8)
#define DS_TELEX_NBR ((OM_type) 8qzerodot9)

/** 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_IAS_TERMINAL_DELIV 7
#define DS_VIDEOTEX_DELIV 8
#define DS_PHONE_DELIV 9

/** upper bounds on string lengths and 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_VL_TELEX_NBR ( (OM_value_length) 6)
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)*
- [z/OS DCE Application Development Guide: Directory Services](docs/1/4-38.Application_Development_Reference_Volumes_1_and_2)
xdscds.h

Purpose
Contains definitions for the Cell Directory Service (CDS).

Format
#include <xom.h>
#include <xds.h>
#include <xdscds.h>

Usage
The xdscds.h header declares the object identifiers of directory attribute types supported by the Cell Directory Service (CDS).

All application programs that include this header must first include the xom.h Object Management header and the xds.h header.

The following shows the contents of the xdscds.h header file:

#ifndef XDSCDS_HEADER
#define XDSCDS_HEADER

/* {iso(1) identified-org(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_Class "\x2B\x16\x01\x03\x0F"
#define OMP_O_DSX_A_CDS_ClassVersion "\x2B\x16\x01\x03\x10"
#define OMP_O_DSX_A_CDS_ObjectUUID "\x2B\x16\x01\x03\x11"
#define OMP_O_DSX_A_CDS_Adress "\x2B\x16\x01\x03\x12"
#define OMP_O_DSX_A_CDS_Replis "\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_DirectoryVersion "\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\x1F"
#define OMP_O_DSX_A_CDS_CHLastAddress "\x2B\x16\x01\x03\x20"
#define OMP_O_DSX_A_CDS_CHUpPointers "\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-org(3) osf(22) dce(1) gds(2)
   = "\x2B\x16\x01\x02" */
#define OMP_O_DSX_UUID "\x2B\x16\x01\x02\x00"
#define OMP_O_DSX_TYPELESS_RDN "\x2B\x16\x01\x02\x00"

Chapter 4. Directory Services 4-39


#define OMP_O_DSX_NORMAL_SIMPLE_NAME "\x20\x16\x01\x03\x00"
#define OMP_O_DSX_BINARY_SIMPLE_NAME "\x20\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)*
xFDSGDS.h

**Purpose**
Contains definitions for the Global Directory Service Package.

**Format**
```
#include <xom.h>
#include <xds.h>
#include <xdsgds.h>
```

**Usage**
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.

The `xdsgds.h` header file is required when using the GDS package.

The following shows the contents of the `xdsgds.h` header file:

```c
#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) object-class(6) ... */
#define dsP_GDSattributeType(X) "\x2B\x0C\x02\x88\x53\x01\x03\x04" #X
#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 DSX_C_)
   Directory attribute types (prefixed DSX_A_)
   Directory object classes (prefixed 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_CACHE_ATTR dsP_GDSattributeType(\x07)
#define OMP_O_DSX_A_DEFAULT_DSA dsP_GDSattributeType(\x08)
#define OMP_O_DSX_A_LOCDSA 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) 85\x00)
#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) 86\x00)
#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_us.Auth_Mechanism {
    DSX_DEFAULT = 1,
    DSX_SIMPLE = 2,
    DSX_SIMPLE_PROT1 = 3,
    DSX_SIMPLE_PROT2 = 4,
    DSX_DCE_AUTH = 5,
    DSX_STRONG = 6
    DSX_SIMPLE = 2,
};

enum DSX_Prot_Request {
    DSX_NONE = 0,
    DSX_SIGNED = 1
};

/* upper bound on string lengths */
#define DSX_VL_PASSWORD ((OM_value_length) 16)

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)
- z/OS DCE Application Development Guide: Directory Services
Purpose

Format
#include <xom.h>
#include <xds.h>
#include <xdsmdup.h>

Usage
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.

The following shows the contents of the xdsmdup.h header file:

#ifndef XDSMDUP_HEADER
#define XDSMDUP_HEADER

#ifndef XMHP_HEADER
#include <xmhp.h>
#endif

/* 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\xqzerodotC\xqzerodot2\x87\x73\x1C\x03"
/* Intermediate object identifier macros */
#define dsP_MHSattributeType(X) ("\x56\x5\x2" #X) /* joint-iso-ccitt 6 5 2 */
#define dsP_MHSobjectClass(X) ("\x56\x5\x1" #X) /* joint-iso-ccitt 6 5 1 */
#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 that 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 OPM_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)*
- z/OS DCE Application Development Guide: Directory Services
- X/Open CAE Specification (November 1991), *API to Electronic Mail (X.400)*
xdssap.h

Purpose
Contains definitions for the Strong Authentication Package.

Format
#include <xom.h>
#include <xds.h>
#include <xdssap.h>

Usage
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\xqzerodotC\xqzerodot2\x87\x73\x1c\xqzerodot2"

/* Intermediate object identifier macros */
#endif

#ifndef dsP_attributeType /* joint-iso-ccitt(2) */
#endif
#define dsP_attributeType (X) ("\x55\xqzerodot4" #X)
#endif

#ifndef dsP_objectClass /* joint-iso-ccitt(2) */
#endif
#define dsP_objectClass(X) ("\x55\xqzerodot6" #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_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)
xmhp.h

Purpose
Definitions for the MHS Directory objects/attributes.

Format
```c
#include <xom.h>
#include <xds.h>
#include <xdsmdup.h>
#include <xmhp.h>
```

Usage
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`.

The `xmhp.h` header contains definitions for the X.400 Message Handling Package. Some of these definitions are needed for negotiating the 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 "MHS Directory User Package" section in the [z/OS DCE Application Development Guide: Directory Services](#).

The following shows the contents of the `xmhp.h` header file:

```c
#ifndef XMHP_HEADER
#define XMHP_HEADER

/*****************************/
/* MH package object identifier */
/*****************************/
#define OMP_O_MH_PACKAGE "\x56\x06\x01\x02\x05\x0B"

/*****************************/
/* BEGIN MH PORTION OF INTERFACE */
/*****************************/

/*****************************/
/* SYMBOLIC CONSTANTS */
/*****************************/

/*****************************/
/* Class */
/*****************************/
#define OMP_O_MH_C_ALGORITHM "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_ALGORITHM_AND_RESULT "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_CASYMMETRIC_TOKEN "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_BILATERAL_INFO "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_COMMUNIQUE "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_CONTENT "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_DELIV_MESSAGE "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_DELIV_PER_RECIP_DR "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_DELIV_PER_RECIP_NDR "\x56\x06\x01\x02\x05\x0B"
#define OMP_O_MH_C_DELIV_PER_RECIP_REP "\x56\x06\x01\x02\x05\x0B"
/*****************************/

/*****************************/
/* Note: the following two names are not in alphabetical order to retain */
/* the value assignment as published in X.400 API, Version 2.0 */
```
/*
#define OMP_O_MH_C_DELIVERY_CONFIRM "\x56\x06\x01\x02\x05\x08\x0B"
#define OMP_O_MH_C_DELIVERY_ENVELOPE "\x56\x06\x01\x02\x05\x08\x0C"
#define OMP_O_MH_C_EITS "\x56\x06\x01\x02\x05\x08\x0F"
#define OMP_O_MH_C_EXPANSION_RECORD "\x56\x06\x01\x02\x05\x08\x10"
#define OMP_O_MH_C_EXTERNAL_TRACE_ENTRY "\x56\x06\x01\x02\x05\x08\x14"
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRM "\x56\x06\x01\x02\x05\x08\x11"
#define OMP_O_MH_C_LOCAL_DELIV_CONFIRMS "\x56\x06\x01\x02\x05\x08\x15"
#define OMP_O_MH_C_LOCAL_DELIVERY_REPORT "\x56\x06\x01\x02\x05\x08\x17"
#define OMP_O_MH_C_LOCAL_PER_RECIP_NDR "\x56\x06\x01\x02\x05\x08\x18"
#define OMP_O_MH_C_LOCAL_PER_ORDNDR "\x56\x06\x01\x02\x05\x08\x19"
#define OMP_O_MH_C_MESSAGE "\x56\x06\x01\x02\x05\x08\x1A"
#define OMP_O_MH_C_MESSAGE_RD "\x56\x06\x01\x02\x05\x08\x1B"
#define OMP_O_MH_C_MTS_IDENTIFIER "\x56\x06\x01\x02\x05\x08\x1C"
#define OMP_O_MH_C_OR_ADDRESS "\x56\x06\x01\x02\x05\x08\x1D"
#define OMP_O_MH_C_OR_NAME "\x56\x06\x01\x02\x05\x08\x1E"
#define OMP_O_MH_C_PER_RECIP_DR "\x56\x06\x01\x02\x05\x08\x20"
#define OMP_O_MH_C_PER_ORDNDR "\x56\x06\x01\x02\x05\x08\x21"
#define OMP_O_MH_C_PROBE "\x56\x06\x01\x02\x05\x08\x22"
#define OMP_O_MH_C_PROBE_RD "\x56\x06\x01\x02\x05\x08\x23"
#define OMP_O_MH_C_RETRANSMIT_CONFIRM "\x56\x06\x01\x02\x05\x08\x24"
#define OMP_O_MH_C_RETRANSMIT_CONFIRMS "\x56\x06\x01\x02\x05\x08\x25"
#define OMP_O_MH_C_RETRANSMIT_MESSAGE "\x56\x06\x01\x02\x05\x08\x26"
#define OMP_O_MH_C_RETRANSMIT_MESSAGE_RD "\x56\x06\x01\x02\x05\x08\x27"
#define OMP_O_MH_C_RETRANSMIT_PROBE "\x56\x06\x01\x02\x05\x08\x28"
#define OMP_O_MH_C_RETRANSMIT_PROBE_RD "\x56\x06\x01\x02\x05\x08\x29"
#define OMP_O_MH_C_SUBMISSION_RESULTS "\x56\x06\x01\x02\x05\x08\x2A"
#define OMP_O_MH_C_SUBMITTED_COMMUNIQUE "\x56\x06\x01\x02\x05\x08\x2B"
#define OMP_O_MH_C_SUBMITTED_MESSAGE "\x56\x06\x01\x02\x05\x08\x2C"
#define OMP_O_MH_C_SUBMITTED_MESSAGE_RD "\x56\x06\x01\x02\x05\x08\x2D"
#define OMP_O_MH_C_SUBMITTED_PROBE "\x56\x06\x01\x02\x05\x08\x2E"
#define OMP_O_MH_C_TOKEN_PUBLIC_DATA "\x56\x06\x01\x02\x05\x08\x2F"
#define OMP_O_MH_C_TOKEN_PUBLIC_DATA "\x56\x06\x01\x02\x05\x08\x30"
#define OMP_O_MH_C_TOKEN_PROBE "\x56\x06\x01\x02\x05\x08\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 )

/* Built In 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_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_VIOLATED ( (OM_enumeration) 13 )
#define MH_DG_NUMBER_CONSTRAINT_VIOLATED ( (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_REASSIGNMENT_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_RENDDITION_UNSUBSCRIBED ( (OM_enumeration) 31 )
#define MH_DG_MAIL_ADDRESS_INCORRECT ( (OM_enumeration) 32 )
#define MH_DG_MAIL_OFFICE_INCOR_OR_INVOL ( (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_MOVED ( (OM_enumeration) 36 )
#define MH_DG_MAIL_FORWARDING_UNWANTED ( (OM_enumeration) 37 )
#define MH_DG_MAIL_NEW_ADDRESS_UNKNOWN ( (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_MOVED ( (OM_enumeration) 42 )
#define MH_DG_MAIL_NEW_ADDRESS_UNKNOWN ( (OM_enumeration) 43 )
#define MH_DG_MAIL_FORWARDING_UNSUSTAINED ( (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_TELEX_TO_IA5_TEXT (OM_enumeration) 11
#define MH_EC_TELETEX_TO_G3_FAX (OM_enumeration) 12
#define MH_EC_TELEX_TO_G4_CLASS1 (OM_enumeration) 13
#define MH_EC_TELEX_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_NOTIFICATION_VIA_PDS (OM_enumeration) 1
#define MH_PR_NOTIFICATION_VIA_MTS (OM_enumeration) 2
#define MH_PR_NOTIFICATION_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_RENDERING_NOT_DONE (OM_enumeration) 3
#define MH_RE_PHYSICAL_DELIVERY_NOT_DONE (OM_enumeration) 4
#define MH_RE_RESTRICTED_DELIVERY (OM_enumeration) 5
#define MH_RE_DIRECTORY_OPERATIONS_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
xmhp.h

/* 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_IAS_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_IAS_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_ADMO_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_RECIPIENT_NAME (OM_type) 207
#define MH_T_ALTERNATE_RECIPIENT_NAME (OM_type) 208
#define MH_T_ARRIVAL_TIME (OM_type) 209
#define MH_T_ATTEMPTED_ADMO_NAME (OM_type) 210
#define MH_T_ATTEMPTED_COUNTRY_NAME (OM_type) 211
#define MH_T_ATTEMPTED_MTA_NAME (OM_type) 212

4-54 Application Development Reference Volumes 1 and 2
#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_INFORMATION ( (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_CONFIRMATIONS ( (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_MISCellanEOUS_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_ORIGINALLY_INTENDED_RECIPIENT (OM_type) 301
#define MH_T_ORIGIATOR_CERTIFICATE (OM_type) 302
#define MH_T_ORIGINATOR_NAME (OM_type) 303
#define MH_T_ORIGIATOR_REQUEST (OM_type) 304
#define MH_T_ORIGIATOR_RETURN_ADDRESS (OM_type) 305
#define MH_T_OTHER_RECIPIENT_NAMES (OM_type) 306
#define MH_T_PAGE_FORMATS (OM_type) 307
#define MH_T_PREFERENCE_REPORT_REQUEST (OM_type) 308
#define MH_T_PREFERENCE_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_DELIVERY_SYSTEM_NAME (OM_type) 316
#define MH_T_POSTAL_GENERAL_DELIVERY_ADDRESS (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_STEP_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_DELIVERY_REQUESTED (OM_type) 336
#define MH_T_PROOF_OF_SUBMISSION (OM_type) 337
#define MH_T_PROOF_OF_SUBMISSION_REQUEST (OM_type) 338
#define MH_T_PUBLIC_INFORMATION (OM_type) 339
#define MH_T_RANDOM_NUMBER (OM_type) 340

4-56 Application Development Reference Volumes 1 and 2
#define MH_T_REASON ( (OM_type) 341 )
#define MH_T_REASSIGNMENT_PROHIBITED ( (OM_type) 342 )
#define MH_T_RECPIENT_CERTIFICATE ( (OM_type) 343 )
#define MH_T_RECPIENT_DESCRIPTORS ( (OM_type) 344 )
#define MH_T_RECPIENT_NAME ( (OM_type) 345 )
#define MH_T_RECPIENT_NUMBER ( (OM_type) 346 )
#define MH_T_RECIP_NUMBER_FOR_ADVICE ( (OM_type) 347 )
#define MH_T_REDUCTION_HISTORY ( (OM_type) 348 )
#define MH_T_REGISTRATION ( (OM_type) 349 )
#define MH_T_RENDITION_ATTRIBUTES ( (OM_type) 34qzerodot )
#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_COUNTRY_NAME ( (OM_value_length) 3 )
#define MH_VL_CUTOM_NAME ( (OM_value_length) 64 )
#define MH_VL_CUTOM_NAME ( (OM_value_length) 64 )
#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_INPUTS ( (OM_value_length) 5 )
#define MH_VL_ISDN_NUMBER ( (OM_value_length) 15 )
#define MH_VL_LOCAL_SUBADDRESS ( (OM_value_length) 40 )
#define MH_VL_LAST_DELIVERY_TIME ( (OM_value_length) 7 )
#define MH_VL_LOCAL_ID ( (OM_value_length) 32 )
#define MH_VL_MTA_NAME ( (OM_value_length) 32 )
#define MH_VL_NUMERIC_USER_ID ( (OM_value_length) 32 )
#define MH_VL_ORGANIZATION_NAME ( (OM_value_length) 64 )
#define MH_VL_OU_NAME ( (OM_value_length) 32 )
#define MH_VL_POSTAL_ADDRESS_DETAILS ( (OM_value_length) 30 )
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)
- X/Open CAE Specification (November 1991), API to Electronic Mail (X.400)
- z/OS DCE Application Development Guide: Directory Services
Purpose
Definitions for the Message Store General attributes.

Format
#include <xom.h>
#include <xds.h>
#include <xdsmdup.h>
#include <xmhp.h>
#include <xmsga.h>

Usage
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, and the xdsmdup.h and xmhp.h headers.

The following shows the contents of the xmsga.h header file:

#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_CONFIDENTIAL_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_NAME     "\x56\x04\x03\x1A"
#define OMP_O_MS_A_ORIGINATOR_CERTIFICATE "\x56\x04\x03\x1B"
#define OMP_O_MS_A_ORIGINATOR_NAME     "\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_REREDIRECTION_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)
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)*
- X/Open CAE Specification (November 1991), *API to Electronic Mail (X.400)*
- *z/OS DCE Application Development Guide: Directory Services*
XOM Routines

This section describes the XOM routines, and defines the functions of the C interface. This interface comprises a number of routines 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. It is similar to the `om_get()` function but includes an additional parameter `navigation_path` which 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 built from the `string` and `class` parameters.

- **om_copy**: Creates an independent copy of an existing private object and all its subobjects. The copy can be placed in any workspace that supports the class of the object.

- **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. Does not permit the client to explicitly create instances of all classes, but only those indicated by a package’s definition as having this property.

- **om_decode**: Not supported by the DCE XOM interface, and returns with an `OM_FUNCTION_DECLINED` error.

- **om_delete**: Deletes a service-generated public object, or makes a private object inaccessible.

- **om_encode**: Not supported by the DCE XOM interface, and returns with an `OM_FUNCTION_DECLINED` error.

- **om_get**: Creates a new public object which 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 routine is useful because 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.

- **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 because any elements beyond it are discarded.

In the C interface, the routines (excluding the omX_* routines) are realized by macros.
om_copy

Purpose
Creates a new private object that is an exact but independent copy of an existing private object.

Format
#include <xom.h>

OM_return_code om_copy(
    OM_private_object original,
    OM_workspace workspace,
    OM_private_object *copy);

Parameters
Input
original The private object that is to be copied. It remains accessible.
workspace The workspace in which the copy is to be created. The original's class must be in a package that is also 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.

Usage
The om_copy routine creates a new private object, the copy, that is an exact but independent copy of an existing private object, the original. The routine is recursive: copying the original also copies its subobjects.

Return Values
OM_return_code Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine 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.

Errors
- 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

Related Information
Books

- *z/OS DCE Application Development Guide: Directory Services*
om_copy_value

Purpose
Places or replaces a string in one private object with a copy of a string in another private object.

Format
#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 object. It remains accessible.
source_type Identifies the type of the attribute that is to be copied.
source_value_position The position within the attribute of the value to be copied.
destination The destination object. It remains accessible.
destination_type Identifies the type of the attribute whose value is to be placed or replaced.
destination_value_position The position within the above attribute of the value to be 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.

Usage
The om_copy_value routine 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 must be a string. The syntax of the copy is that of the original.

Return Values
OM_return_code Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine 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.

Errors
- 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
Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services](#)
om_create

Purpose
Creates a new private object that is an instance of a particular class.

Format
```
#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 class must be concrete and among those that are permitted to be created for the workspace.

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 must be in a package associated with this workspace.

Output

object
The created object. This result is valid only if the return value for OM_return_code is OM_SUCCESS.

Usage
The om_create routine creates a new private object that is an instance of a particular class. Attributes of the class that have initial values will be set in the new object, if desired.

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

OM_return_code
Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine 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.
Errors

- 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

Related Information

Books

- z/OS DCE Application Development Guide: Directory Services
om_delete

Purpose
Deletes a service-generated object.

Format
#include <xom.h>

OM_return_code om_delete(
    OM_object subject);

Parameters
Input
subject The object that is to be deleted. It must be service generated.

Usage
The om_delete API 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 routine deletes the object and releases any resources associated with it, including the space occupied by descriptors and attribute values. The routine is applied recursively to any public subobjects. It does not affect any private subobjects.

If applied to a private object, the routine makes the object inaccessible. Any existing object handles for the object are invalidated. The routine is applied recursively to any private subobjects.

Return Values
OM_return_code Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine fails, it has one of the values listed under Errors.

Errors
- 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

Related Information
Books

- z/OS DCE Application Development Guide: Directory Services
**Purpose**

Creates a public copy of all or particular parts of a private object.

**Format**

```c
#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 private source object. It remains **accessible**.

- `exclusions` Explicit requests for 0 (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. Lower numbered exclusions have 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.

1. **OM_EXCLUDE_ALL_BUT_THOSE_TYPES**
   
The copy includes descriptors comprising only attributes of specified types.
   
   **Note**: This exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

2. **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. The resulting descriptor does not contain 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**: This exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

3. **OM_EXCLUDE_ALL_BUT_THOSE_VALUES**
   
The copy includes descriptors comprising only values at specified positions within an attribute.

   **Note**: When used in conjunction with the **OM_EXCLUDE_ALL_BUT_THOSE_TYPES** exclusion, this exclusion provides a means for determining the values of a specified attribute, as well as the syntaxes of only a portion of those values at a time.

4. **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: This exclusion provides a means for determining an object’s composition, that is, the type and syntax of each of its attribute values.

5. **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 routine calls.

   **Note:** This exclusion provides a means for examining an object one level at a time.

6. **OM_EXCLUDE_DESCRIPTORS**

   When this exclusion is specified, no descriptors are returned and copy 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:** 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**, as well as **OM_EXCLUDE_ALL_BUT_THESE_TYPES**.

   **Note:** 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 and it identifies the types of the attributes to be included in the copy (provided that they appear 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, refer to the **z/OS 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 public copy of source. 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. The **total_number** result reflects the actual number of attribute descriptors that would be returned, based on the remaining inclusion and exclusion values.

**Note:** The total includes only the attribute descriptors in the copy parameter. It excludes the special descriptor signaling the end of a public object.
**om_get**

**Usage**

The om_get API 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.

**Note:** The client can access long values by means of `om_read`.

**Return Values**

**OM_return_code**

Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of `OM_return_code` is set to `OM_SUCCESS`. If the routine 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.

**Errors**

- `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`

**Related Information**

**Books**

- [z/OS DCE Application Development Guide: Directory Services](#)
Purpose
Determines whether an object is an instance of a particular class or any of its subclasses.

Format
```
#include <xom.h>

OM_return_code om_instance(
    OM_object subject ,
    OM_object_identifier class ,
    OM_boolean *instance );
```

Parameters

Input
- `subject` The purported object. It 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 the value of the `OM_return_code` is set to `OM_SUCCESS`.

Usage
The `om_instance` API 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 specific class by simply inspecting the object (using programming language constructs if the object is public). The `om_instance` API is useful in that it reveals that an object is an instance of the specified class, even if the `subject`'s class is a subclass of that class.

Return Values

- **OM_return_code** Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of `OM_return_code` is set to `OM_SUCCESS`. If the routine fails, it has one of the values listed under Errors.

Errors

- `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_instance

Related Information

Books

- [z/OS DCE Application Development Guide: Directory Services]
om_put

Purpose
Places or replaces in one private object copies of the attribute values from another public or private object.

Format
#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 object. It remains accessible, and its 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 existing destination values, if any, are retained.)

- OM_INSERT_AT_CERTAIN_POINT
  The source values are inserted before the value at a specified position in the destination attribute. (The existing destination values, if any, are retained.)

- OM_INSERT_AT_END
  The source values are inserted after any existing destination values. (The existing destination values, if any, 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 existing destination values, if any, are discarded.)

source The source object. It 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 (not NULL), this parameter identifies the types of the attributes to be included in the destination (provided that they are present 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.
om_put

limiting_value

Present if the modification parameter is **OM_REPLACECERTAINVALUES**. 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.

**Usage**

The om_put API places or replaces in one private object (that is, the destination) copies of the attribute values that pertain to the destination from 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**

**OM_return_code**

Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of **OM_return_code** is set to **OM_SUCCESS**. If the routine 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. For more information, refer to “xom.h” on page 4-95.

**Errors**

- **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_NOTPRIVATE**
- **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**

**Related Information**
Books

- \textit{z/OS DCE Application Development Guide: Directory Services}
om_read

Purpose
Reads a segment of a string in a private object.

Format
#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 private object. It remains accessible.
type Identifies the type of the attribute one of whose values is read.
value_position The position within the above 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, refer to the z/OS DCE Application Development Guide: Directory Services.
string_offset 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 taken to be equal to the string length.
elements 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 routine 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. This can be less than the initial length if the segment is the last one in the string.

If local_string is true, the segments that will be returned will be those of the translated string. Depending on the characteristics of the implementation-defined local character set, these may not correspond directly to the segments that would be obtained if local_string were false.

Output
string_offset 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 preset 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 routine. This enables sequential reading of a value of a long string.

Usage
The om_read API 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.

Return Values
OM_return_code Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine 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.
Errors

- 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

Related Information

Books

- z/OS DCE Application Development Guide: Directory Services
om_remove

Purpose
Removes and discards values of an attribute of a private object.

Format
#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 private object. It remains accessible, and its class is unaffected.

type
Identifies the type of the attribute whose values are to be removed. The type cannot be OM_CLASS.

initial_value
The position within the attribute of the first value to be 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.

Usage
The om_remove API 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 subobject.

Return Values

OM_return_code
Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine fails, it has one of the values listed under Errors.

Errors

- 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
Related Information

Books

- z/OS DCE Application Development Guide: Directory Services
om_write

Purpose
Writes a segment of a string into a private object.

Format

```c
#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 private object. It 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 cannot 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 parameter identifies the syntax that the value is to have. The syntax must be permissible for the attribute. If the value being written is already present in the subject, then the syntax of that value is preserved and this parameter is ignored.

- **string_offset**
  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.

- **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.

**Output**

- **string_offset**
  The offset, in octets, after the last string segment written. This result is preset 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 routine is called. This enables sequential writing of the value of a long string.

Usage

The om_write API writes a segment of an attribute value in a private object, the subject parameter.

The segment specified 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 routine 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).

Return Values

- **OM_return_code**
  Indicates whether the routine succeeded and, if not, the reason. If the routine is successful, the value of OM_return_code is set to OM_SUCCESS. If the routine 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.

**Errors**

- `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`

**Related Information**

**Books**

- [z/OS DCE Application Development Guide: Directory Services](#)
omX_extract

Purpose
Extracts the first occurrence of the requested OM type from an object and creates from it a new public object.

Format
#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.

1. **OM_EXCLUDE_ALL_BUT THESE_TYPES**
   The copy includes descriptors comprising only attributes of specified types.

   Note: This exclusion provides a means for determining the values of specified attributes, as well as the syntaxes of those values.

2. **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: This exclusion provides a means for discerning the presence of multivalued attributes without simultaneously obtaining their values.

3. **OM_EXCLUDE_ALL_BUT THESE VALUES**
   The copy includes descriptors comprising only values at specified positions within an attribute.

   Note: When used in conjunction with the **OM_EXCLUDE_ALL_BUT THESE_TYPES** exclusion, this exclusion 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.
4. **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:** This exclusion provides a means for determining an object's composition; that is, the type and syntax of each of its attribute values.

5. **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:** This exclusion provides a means for examining an object one “level” at a time.

6. **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:** 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.
omX_extract

Usage

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 similar 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

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 under "xom.h" on page 4-95.

Error Conditions

Refer to xom.h for a list of the possible error values that can be returned in OM_return_code.
omX_fill

Purpose
Initializes an OM_descriptor structure.

Format
#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 or pointer to the subobject.

Output
    destination     Contains the filled descriptor.

Usage
The omX_fill() function is used to initialize an OM descriptor structure with user supplied values for its type, syntax, and value.

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 under "xom.h" on page 4-95.

Error Conditions
Refer to xom.h 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 (OID) value.

**Format**
```
#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.

**Usage**
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**

- **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 under "**xom.h**" on page 4-95.

**Error Conditions**
Refer to **xom.h** for a list of the possible error values that can be returned in **OM_return_code**.
omX_object_to_string

Purpose
Converts an OM object from descriptor to string format.

Format
```
#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 refer to the [z/OS DCE Application Development Guide: Directory Services](https://www.ibm.com/downloads/collateral/ce/dce-guide-directory-services-en.pdf).

**Output**
- **string**: Contains the converted object in string format. 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.

Usage

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` (found in `/opt/dcelocal/etc`). 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.

Possible error strings that can be returned are:

<table>
<thead>
<tr>
<th>Error Message Text</th>
<th>DS_PROBLEM Value from DS_C_ERROR Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alias found where an alias is not permitted.</td>
<td>DS_E_ALIAS_DEREFERENCING_PROBLEM</td>
</tr>
<tr>
<td>Alias that names an object that does not exist was de-referenced.</td>
<td>DS_E_ALIAS_PROBLEM</td>
</tr>
<tr>
<td>Attempt to end an operation failed.</td>
<td>DS_E_CANNOT_ABANDON</td>
</tr>
<tr>
<td>Attempted to add an attribute or value that already exists.</td>
<td>DS_E_ATTRIBUTE_OR_VALUE_EXISTS</td>
</tr>
<tr>
<td>Attempted to start a synchronous operation with outstanding asynchronous operations.</td>
<td>DS_E_MIXED_SYNCHRONOUS</td>
</tr>
<tr>
<td>Attempted to use optional function that is not available in this implementation.</td>
<td>DS_E_NOT_SUPPORTED</td>
</tr>
<tr>
<td>Attempted to use undefined matching rule for the attribute type.</td>
<td>DS_E_INAPPROPRIATE_MATCHING</td>
</tr>
<tr>
<td>Attribute or attribute value does not conform to restrictions that apply.</td>
<td>DS_E_CONSTRAINT_VIOLATION</td>
</tr>
<tr>
<td>Attribute type not included in the AVA.</td>
<td>DS_E_MISSING_TYPE</td>
</tr>
<tr>
<td>Attribute type undefined.</td>
<td>DS_E_UNDEFINED_ATTRIBUTE_TYPE</td>
</tr>
<tr>
<td>Attribute value does not conform to the attribute syntax of the attribute type.</td>
<td>DS_E_INVALID_ATTRIBUTE_SYNTAX</td>
</tr>
<tr>
<td>Attribute value in the AVA does not conform to the attribute syntax.</td>
<td>DS_E_INVALID_ATTRIBUTE_VALUE</td>
</tr>
<tr>
<td>Chaining required to perform the operation, service control.</td>
<td>DS_E_CHAINING_REQUIRED</td>
</tr>
<tr>
<td>Communication error occurred.</td>
<td>DS_E_COMMUNICATIONS_PROBLEM</td>
</tr>
</tbody>
</table>
### Return Values

**OM_return_code** Indicates whether the function succeeded and, if not, why not. If the function is successful, the value **OM_return_code** is set to **OM_SUCCESS**; if the function fails, it has one of the error values listed under “xom.h” on page 4-95.

### Error Conditions

Refer to “xom.h” on page 4-95 and “xomext.h” on page 4-101 for a list of the possible error values that can be returned in **OM_return_code**.

---

#### Table 4-2 (Page 2 of 2). DS_C_ERROR Object Error Messages

<table>
<thead>
<tr>
<th>Error Message Text</th>
<th>DS_PROBLEM Value from DS_C_ERROR Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credentials of requester are incorrect.</td>
<td>DS_E_INVALID_CREDENTIALS</td>
</tr>
<tr>
<td>DSA detected a loop within the directory.</td>
<td>DS_E_LOOP_DETECTED</td>
</tr>
<tr>
<td>DSA has no administrative authority over the specified naming context.</td>
<td>DS_E_UNABLE_TO_PROCEED</td>
</tr>
<tr>
<td>DSA unable to perform the request as specified.</td>
<td>DS_E_INVALID_REF</td>
</tr>
<tr>
<td>Directory Service has no knowledge of the operation being ended.</td>
<td>DS_E_NO_SUCH_OPERATION</td>
</tr>
<tr>
<td>Directory Service not willing to perform operation because it is unsigned.</td>
<td>DS_E_PROTECTION_REQUIRED</td>
</tr>
<tr>
<td>Directory Service too busy to perform the operation.</td>
<td>DS_E_BUSY</td>
</tr>
<tr>
<td>Directory cannot provide a referral or partial outcome qualifier within the required scope.</td>
<td>DS_E_OUT_OF_SCOPE</td>
</tr>
<tr>
<td>Inconsistency detected in the directory information tree.</td>
<td>DS_E_DIT_ERROR</td>
</tr>
<tr>
<td>Incorrect argument specified.</td>
<td>DS_E_BAD_ARGUMENT</td>
</tr>
<tr>
<td>Incorrect context argument specified.</td>
<td>DS_E_BAD_CONTEXT</td>
</tr>
<tr>
<td>Incorrect name argument specified.</td>
<td>DS_E_BAD_NAME</td>
</tr>
<tr>
<td>Incorrect session argument specified.</td>
<td>DS_E_BAD_SESSION</td>
</tr>
<tr>
<td>Incorrect work space argument specified.</td>
<td>DS_E_BAD_WORKSPACE</td>
</tr>
<tr>
<td>Miscellaneous error occurred when interacting with the Directory Service.</td>
<td>DS_E_MISCELLANEOUS</td>
</tr>
<tr>
<td>Modification affects several DAS and is prohibited.</td>
<td>DS_E_AFFECTS_MULTIPLE_DSAS</td>
</tr>
<tr>
<td>Modification leaves the directory information tree with an incorrect structure.</td>
<td>DS_E_NAMING_VIOLATION</td>
</tr>
<tr>
<td>Modification to an interior node of the directory information tree is prohibited.</td>
<td>DS_E_NOT_ALLOWED_ON_NON_LEAF</td>
</tr>
<tr>
<td>Modification will change RDN of an object.</td>
<td>DS_E_NOT_ALLOWED_ON_RDN</td>
</tr>
<tr>
<td>Modification will change object class attribute of entry.</td>
<td>DS_E_OBJECT_CLASS_MOD_PROHIB</td>
</tr>
<tr>
<td>Modification will leave directory entry inconsistent with its object class definition.</td>
<td>DS_E_OBJECT_CLASS_VIOLATION</td>
</tr>
<tr>
<td>Name passed to add entry operation already exists.</td>
<td>DS_E_ENTRY_EXISTS</td>
</tr>
<tr>
<td>No more Directory Service operations can be performed.</td>
<td>DS_E_TOO_MANY_OPERATIONS</td>
</tr>
<tr>
<td>No more Directory Service sessions can be started.</td>
<td>DS_E_TOO_MANY_SESSIONS</td>
</tr>
<tr>
<td>OM Class of an argument not supported for this operation.</td>
<td>DS_E_BAD_CLASS</td>
</tr>
<tr>
<td>Operation already completed.</td>
<td>DS_E_TOO_LATE</td>
</tr>
<tr>
<td>Operation cannot be performed within the administrative restrictions.</td>
<td>DS_E_ADMIN_LIMIT_EXCEEDED</td>
</tr>
<tr>
<td>Operation cannot be performed within the specified time limit.</td>
<td>DS_E_TIME_LIMIT_EXCEEDED</td>
</tr>
<tr>
<td>Part of directory not available.</td>
<td>DS_E_UNAVAILABLE</td>
</tr>
<tr>
<td>Part of the Directory Service not willing to perform the operation.</td>
<td>DS_E_UNWILLING_TO_PERFORM</td>
</tr>
<tr>
<td>Request produced security error for which no other information is available.</td>
<td>DS_E_NO_INFO</td>
</tr>
<tr>
<td>Requested critical extensions not available.</td>
<td>DS_E_UNAVAILABLE_CRIT_EXT</td>
</tr>
<tr>
<td>Requestor does not have permission to perform the operation.</td>
<td>DS_E_INSUFFICIENT_ACCESS_RIGHTS</td>
</tr>
<tr>
<td>Security level attached to the requestor credentials is inconsistent.</td>
<td>DS_E_INAPPROPRIATE_AUTHENTICATION</td>
</tr>
<tr>
<td>Signature attached to request is incorrect.</td>
<td>DS_E_INVALID_SIGNATURE</td>
</tr>
<tr>
<td>Specified attribute or value not found in the directory entry.</td>
<td>DS_E_NO_SUCH_ATTRIBUTE_OR_VALUE</td>
</tr>
<tr>
<td>Specified name does not match name of any object in the directory.</td>
<td>DS_E_NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>
omX_string_to_object

Purpose
Creates a new private object, which is built from the string and class input parameters.

Format
```c
#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 "Convenience Routines" chapter of the [z/OS 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. (for example, Attribute Abbreviation expected, '/' expected, and so on...). Refer to "xomext.h" on page 4-101 for explanations of the error types.

Usage
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` (found in `/opt/dcelocal/etc`).

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
- **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 under "xom.h" on page 4-95.

  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`.

Chapter 4. Directory Services 4-93
omX_string_to_object

Error Conditions

Refer to xom.h and xomext.h for a list of the possible error values that can be returned in OM_return_code and error_type.
xom.h

Purpose
Header file for XOM.

Format
#include <xom.h>

Usage
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.

The following shows the contents of the xom.h header file:
 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 / qc /qc */

/* 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 */
/* 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 */
#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 */
#define OMP_O_OM_OM         "\56\x06\x01\x02\x04"
/* Constant for the Encoding class */
#define OMP_O_OM_C_ENCODING  "\56\x06\x01\x02\x04\x01"
/* Constant for the External class */
#define OMP_O_OM_C_EXTERNAL  "\56\x06\x01\x02\x04\x02"
/* Constant for the Object class */
#define OMP_O_OM_C_OBJECT    "\56\x06\x01\x02\x04\x03"
/* Constant for the BER Object Identifier */
#define OMP_O_OM_BER         "\51\x01"
/* Constant for the Canonical-BER Object Identifier */
#define OMP_O_OM_CANONICAL_BER "\56\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) 0x8000
#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
xom.h

#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)
- X/Open CAE Specification (November 1991), API to Electronic Mail (X.400)
- z/OS DCE Application Development Guide: Directory Services
xomext.h

Purpose
Header file for XOM convenience functions.

Format
#include <xomext.h>

Usage
This file contains all the defines and function prototypes for the XOM public convenience routines. These routines, used in XDS application development, target filling, comparing, extracting, and string conversion of objects. Detailed information about them can be found in the "XDS/XOM Convenience Routines" section of the z/OS DCE Application Development Guide: Directory Services.

The following shows the contents of the xomext.h header file:

```c
#ifndef _XOMEXT_H
#define _XOMEXT_H

#if !defined(lint) && defined(GDS_RCS_ID)
static char xomext_rcsid[] = "@(#)$RCSfile: xomext.h,v $ $Revision: 1.1.2.3 $ $Date: 1994/06/10 21:21:44 $";
#endif

/**- Defines for error returns related to XOI -----------------------------*/
#define OMX_SUCCESS ((OM_integer) 0)
#define OMX_CANNOT_READ_SCHEMA ((OM_integer) -1)
#define OMX_SCHEMA_NOT_READ ((OM_integer) -2)
#define OMX_NO_START_OBJ_BLOCK ((OM_integer) -3)
#define OMX_NO_END_OBJ_BLOCK ((OM_integer) -4)
#define OMX_EMPTY_OBJ_BLOCK ((OM_integer) -5)
#define OMX_OBJ_FORMAT_ERROR ((OM_integer) -6)
#define OMX_DUPLICATE_OBJ_ABBRV ((OM_integer) -7)
#define OMX_DUPLICATE_OBJ_OBJ_ID ((OM_integer) -8)
#define OMX_NO_START_ATTR_BLOCK ((OM_integer) -9)
#define OMX_NO_END_ATTR_BLOCK ((OM_integer) -10)
#define OMX_EMPTY_ATTR_BLOCK ((OM_integer) -11)
#define OMX_ATTR_FORMAT_ERROR ((OM_integer) -12)
#define OMX_DUPLICATE_ATTR_ABBRV ((OM_integer) -13)
#define OMX_DUPLICATE_ATTR_OBJ_ID ((OM_integer) -14)
#define OMX_NO_START_CLASS_BLOCK ((OM_integer) -15)
#define OMX_NO_END_CLASS_BLOCK ((OM_integer) -16)
#define OMX_EMPTY_CLASS_BLOCK ((OM_integer) -17)
#define OMX_CLASS_FORMAT_ERROR ((OM_integer) -18)
#define OMX_NO_CLASS_NAME ((OM_integer) -19)
#define OMX_DUPLICATE_CLASS_BLOCK ((OM_integer) -20)
#define OMX_CLASS_BLOCK_UNDEFINED ((OM_integer) -21)
#define OMX_INVALID_ABBRV ((OM_integer) -22)
#define OMX_INVALID_OBJ_ID ((OM_integer) -23)
#define OMX_INVALID_CLASS_NAME ((OM_integer) -24)
#define OMX_INVALID_SYNTAX ((OM_integer) -25)
#define OMX_MEMORY_INSUFFICIENT ((OM_integer) -26)
#define OMX_INVALID_PARAMETER ((OM_integer) -27)
#define OMX_UNKNOWN_ABBRV ((OM_integer) -28)
#define OMX_UNKNOWN_OBJ_ID ((OM_integer) -29)
#define OMX_UNKNOWN_OMTYPE ((OM_integer) -30)
```
xomext.h

/**- Defines for error returns related to convenience library */
#define OMX_MISSING_AVA ((OM_integer) -101)
#define OMX_MISSING_ABBRV ((OM_integer) -102)
#define OMX_FORMAT_ERROR ((OM_integer) -103)
#define OMX_UNKNOWN_ERROR ((OM_integer) -104)
#define OMX_MISSING_RDN_DELIMITER ((OM_integer) -105)
#define OMX_MISMATCHED_QUOTES ((OM_integer) -106)
#define OMX_MISSING_ATTR_INFO ((OM_integer) -107)
#define OMX_MISSING_CLASS_START_OP ((OM_integer) -108)
#define OMX_MISSING_CLASS_END_OP ((OM_integer) -109)
#define OMX_MISMATCHED_BRACKETS ((OM_integer) -110)
#define OMX_UNEXPECTED_OPERATOR ((OM_integer) -111)
#define OMX_WRONG_VALUE ((OM_integer) -112)
#define OMX_UNKNOWN_KEYWORD ((OM_integer) -113)
#define OMX_MISSING_OPERATOR ((OM_integer) -114)
#define OMX_MISSING_COMPOUND_OP ((OM_integer) -115)

/**- Additional Errors returned by the omX_object_to_string function */
#define OMX_CLASS_NOT_FOUND_IN_SCHEMA_FILE ((OM_return_code) 31)

/**- Function Prototypes */
OM_return_code omX_fill( 
    OM_type type, /* IN - Type of Object */ 
    OM_syntax syntax, /* IN - Syntax of the object */ 
    OM_uint32 length, /* IN - Data length */ 
    void *elements, /* IN - Data Value */ 
    OM_descriptor *destination); /* OUT - The filled up descriptor */

OM_return_code omX_fill_oid( 
    OM_type type, /* IN - Type of Object */ 
    OM_object_identifier object_id, /* IN - Value of the object */ 
    OM_descriptor *destination); /* OUT - The filled up descriptor */

OM_return_code omX_extract( 
    OM_private_object object, /* IN - Extract from this object */ 
    OM_type_list navigation_path, /* IN - Leads to the target object */ 
    OM_exclusions exclusions, /* IN - Scope of extraction */ 
    OM_type_list included_types, /* IN - Objects to be extracted */ 
    OM_boolean local_strings, /* IN - Local strings required */ 
    OM_value_position initial_value, /* IN - First value to be extracted */ 
    OM_value_position limiting_value, /* IN - Last value to be extracted */ 
    OM_public_object *values, /* OUT - Array of extracted objects */ 
    OM_value_position *total_number); /* OUT - Count of extracted objects */
OM_return_code omX_string_to_object(
    OM_workspace workspace, /* IN - The workspace */
    OM_string *string, /* IN - The string to be converted */
    OM_object_identifier class, /* IN - The OM Class to be created */
    OM_boolean local_strings, /* IN - Local strings specified */
    OM_private_object *object, /* OUT - The converted Object */
    OM_integer *error_position, /* OUT - Error Position in I/P string */
    OM_integer *error_type); /* OUT - Type of error */

OM_return_code omX_object_to_string(
    OM_object object, /* The Object to be converted */
    OM_boolean local_strings, /* To indicate local string conversion */
    OM_string *string); /* The converted DN string */

#endif /* ifndef _XOMEXT_H */

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)*
- X/Open CAE Specification (November 1991), *API to Electronic Mail (X.400)*
- *z/OS DCE Application Development Guide: Directory Services*
Chapter 5. Distributed Time Services

This chapter describes the Distributed Time Service (DTS) APIs. DTS provides the following services:

- Obtains timestamps that are based on Universal Time Coordinated (UTC)
- Translates between different timestamp formats
- Performs calculations on timestamps.

Applications can call the DTS routines from server or clerk systems and can use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling.

The DTS routines 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 text strings
- Compare two binary time values
- Calculate binary time values
- Obtain time zone information.

These terms are used throughout this chapter:

Absolute time An interval on a time scale; absolute time measurements are derived from system clocks or external time-providers. For DTS, absolute times refer to the UTC standard and include inaccuracy and other information. When you display an absolute time, DTS converts the time to a text string, as shown in the following display:

1990-11-21-13:30:25.785-04:00I000.082

Relative time A discrete time interval that is often added to or subtracted from an absolute time. A Time Differential Factor (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, because they concern absolute time.

Coordinated Universal Time (UTC) The international time standard that DTS uses. The zero hour of UTC is based on the zero hour of Greenwich Mean Time (GMT). This book also refers to the time zone of the Greenwich Meridian as GMT. However, this time zone is more often referred to as UTC.

The Time Differential Factor (TDF) The difference between UTC and the time in a particular time zone.

The user’s environment determines the time zone rule. If the user’s environment does not specify a time zone rule, the system’s rule, which is in the localtime file, is used.

The host system must be set to the correct time zone, and DTS determines the time zone to use for the host system as follows:

- If the TZ environment variable has been set, DTS uses this value.
- If the TZ environment variable has not been set, DTS uses the time zone file that has been copied to the localtime file.

For more information on setting the time zone, refer to the z/OS DCE Administration Guide.

The z/OS DCE Application Development Guide: Core Components provides additional information about UTC and GMT, TDF and time zones, and relative and absolute times.

The following illustration categorizes the DTS application program interface routines by function.
The following table summarizes the DTS application program interface routines.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utc_abstime</td>
<td>Computes the absolute value of a relative binary timestamp.</td>
</tr>
<tr>
<td>utc_addtime</td>
<td>Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time.</td>
</tr>
<tr>
<td>utc_anytime</td>
<td>Converts a binary timestamp to a <em>tm</em> structure, using the TDF information contained in the timestamp to determine the TDF returned with the <em>tm</em> structure.</td>
</tr>
<tr>
<td>utc_anyzone</td>
<td>Gets the time zone label and offset from GMT, using the TDF contained in the input <em>utc</em>.</td>
</tr>
<tr>
<td>utc_ascanytime</td>
<td>Converts a binary timestamp to a text string that represents an arbitrary time zone.</td>
</tr>
<tr>
<td>utc_ascgmttime</td>
<td>Converts a binary timestamp to a text string that expresses a GMT time.</td>
</tr>
<tr>
<td>utc_asclocaltime</td>
<td>Converts a binary timestamp to a text string that represents a local time.</td>
</tr>
<tr>
<td>utc_ascreltime</td>
<td>Converts a relative binary timestamp to a text string that represents the time.</td>
</tr>
</tbody>
</table>
Table 5-1 (Page 2 of 2). DTS Application Program Interface Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utc_binreltime</td>
<td>Converts a relative binary timestamp to two timespec structures that express relative time and inaccuracy.</td>
</tr>
<tr>
<td>utc_bintime</td>
<td>Converts a binary timestamp to a timespec structure.</td>
</tr>
<tr>
<td>utc_boundtime</td>
<td>Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.</td>
</tr>
<tr>
<td>utc_cmpintervaltime</td>
<td>Compares two binary timestamps or two relative binary timestamps.</td>
</tr>
<tr>
<td>utc_cmpmidtime</td>
<td>Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.</td>
</tr>
<tr>
<td>utc_gettime</td>
<td>Returns the current DCE system time and inaccuracy as a binary timestamp.</td>
</tr>
<tr>
<td>utc_getusertime</td>
<td>Returns the DCE system time and process-specific TDF, rather than the system-specific TDF.</td>
</tr>
<tr>
<td>utc_gmtime</td>
<td>Converts a binary timestamp to a tm structure that expresses GMT or the equivalent UTC.</td>
</tr>
<tr>
<td>utc_gmtzone</td>
<td>Gets the time zone label for GMT.</td>
</tr>
<tr>
<td>utc_localtime</td>
<td>Converts a binary timestamp to a tm structure that expresses local time.</td>
</tr>
<tr>
<td>utc_localzone</td>
<td>Gets the time zone label and offset from GMT, given utc.</td>
</tr>
<tr>
<td>utc_mkanytime</td>
<td>Converts a tm structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkascreltime</td>
<td>Converts a NULL-terminated text string that represents a relative timestamp to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkasctime</td>
<td>Converts a NULL-terminated text string that represents an absolute timestamp to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkbinreltime</td>
<td>Converts a timespec structure expressing a relative time to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkbintime</td>
<td>Converts a timespec structure to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkgmtime</td>
<td>Converts a tm structure that expresses GMT or UTC to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mklcaltime</td>
<td>Converts a tm structure that expresses local time to a binary timestamp.</td>
</tr>
<tr>
<td>utc_mkreltime</td>
<td>Converts a tm structure that expresses relative time to a relative binary timestamp.</td>
</tr>
<tr>
<td>utc_mulftime</td>
<td>Multiplies a relative binary timestamp by a floating-point value.</td>
</tr>
<tr>
<td>utc_multime</td>
<td>Multiplies a relative binary timestamp by an integer factor.</td>
</tr>
<tr>
<td>utc_pointtime</td>
<td>Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.</td>
</tr>
<tr>
<td>utc_reltime</td>
<td>Converts a relative binary timestamp to a tm structure.</td>
</tr>
<tr>
<td>utc_spantime</td>
<td>Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps.</td>
</tr>
<tr>
<td>utc_subtime</td>
<td>Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times.</td>
</tr>
</tbody>
</table>

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. However, if all output parameters are NULL then the API returns unsuccessfully, with a return code of -1.

**Note:** When an API returns -1, the errno value is set to EINVAL for the invalid input and output values.

**Distributed Time Services Routines**

This section describes in detail each DTS API in alphabetical order.
utc_abstime

Purpose
Computes the absolute value of a relative binary timestamp.

Format
#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.

Usage
The utc_abstime API computes the absolute value of a relative binary timestamp. The input timestamp represents a relative (delta) time.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates a time parameter or the results that are not valid.

Examples
The following example scales a relative time, computes its absolute value, and prints the result.
utc_abstime

utc_t relutc, scaledutc;
char timstr[UTC_MAX_STR_LEN];

/*
 */

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_ascrreltime(timstr, /* Out: Text relative time */
     UTC_MAX_STR_LEN, /* In: Length of input string */
     &scaledutc); /* In: Relative time to convert */

printf("%s
",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_ascrreltime(timstr, /* Out: Text relative time */
     UTC_MAX_STR_LEN, /* In: Length of input string */
     &scaledutc); /* In: Relative time to convert */

printf("%s
",timstr);
utc_addtime

Purpose
Computes the sum of two binary timestamps. The timestamps can be two relative times or a relative time and an absolute time.

Format
#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 on 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 following Usage)

Usage
The utc_addtime API adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies. At least one of the input timestamps typically represents a relative (delta) time. The TDF in the first input timestamp is copied to the output.

Note: Although no error is returned, the combination absolute time + absolute time should not be used.

Return Values
0    Indicates that the routine ran successfully.
-1   Indicates that a time parameter or the results are not valid.

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 = qzerodot;
tzero.tv_sec = 0;
tzero.tv_nsec = 0;
utc_mkbinreltime(&fivesec, /* Out: 5 seconds in binary timestamp */
                &tfivesec, /* In: 5 seconds in timespec */
                &tzero); /* In: 0 seconds inaccuracy in timespec */

/* Get the maximum possible current time... */
/* (The NULL input parameter is used to specify the current time.) */
utc_pointtime((utc_t)qzerodot, /* Out: Earliest possible current time */
               (utc_t)qzerodot, /* 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 seconds */

Related Information

Routines

utc_subtime
Purpose
Converts a binary timestamp to a tm structure, using the TDF information contained in the timestamp to determine the TDF returned with the tm structure.

Format
#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.
tnss
Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, tm_mday returns a value of -1 and tm_mon and tm_year return values of 0. The field tm_yday contains the inaccuracy in days. If the inaccuracy is unspecified, all tm structure fields return values of -1.
inacctm
Nanoseconds of inaccuracy component of the binary timestamp.
inacctm
Nanoseconds since Time component of the binary timestamp.
tdf
TDF component of the binary timestamp in units of seconds east of GMT.

Usage
The utc_anytime API converts a binary timestamp to a 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 ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
The following example converts a timestamp, 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...  
* */
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:  
* */
printf("%d-%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%03d", ievnt.tv_sec, (ievnt.tv_nsec/1000000));

printf(" (%s)", tznam);

Related Information

Routines

utc_mkanytime
utc_gettime
utc_gmtime
utc_anyzone
utc_getusertime
utc_localtime

Chapter 5. Distributed Time Services  5-9
utc_anyzone

utc_anyzone

Purpose
Gets the time zone label and offset from GMT, using the TDF contained in the input utc.

Format
#include <dce/utc.h>

int utc_anyzone(
    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 that is long enough to hold the time zone label.
tdf    Longword 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 Savings Time. A value of -1 is always returned.

Usage
The utc_anyzone API gets the time zone label and offset from GMT, using the TDF contained in the input utc. 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.)

Note: 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 ran successfully.
-1    Indicates that a time argument is not valid or an insufficient buffer.

Examples
See the sample program in "utc_anytime" on page 5-8

Related Information
### Routines

<table>
<thead>
<tr>
<th>utc_anytime</th>
<th>utc_gmtzone</th>
<th>utc_localzone</th>
</tr>
</thead>
</table>


Purpose
Converst a binary timestamp to a text string that represents an arbitrary time zone.

Format

```c
#include <dce/utc.h>

int utc_ascanytime(
    char *cp,
    size_t strlen,
    utc_t utc);
```

Parameters

**Input**

- `strlen` 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` Text string that represents the time.

Usage

The `utc_ascanytime` API converts a binary timestamp to a text 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 ran successfully.
- `-1` Indicates that a time parameter or the results are not valid.

Examples

The following example converts a time to a text string that expresses the time in the time zone where the timestamp was generated:

```c
utc_t evnt;
char localTime[UTC_MAX_STR_LEN];

/*
 * Assuming that evnt contains the timestamp to convert, convert
 * the time to a text string in the following format:
 * 1991-04-01-12:27:38.37-8:0012.00
 */
utc_ascanytime(localtime, /* Out: Converted time */
    UTC_MAX_STR_LEN, /* In: Length of string */
    &evnt); /* In: Time to convert */
```

Related Information
Chapter 5. Distributed Time Services

**Routines**

| utc_ascgmtime | utc_asclocaltime |
utc_ascgmtime

Purpose
Converts a binary timestamp to a text string that expresses a GMT time.

Format
#include <dce/utc.h>

int utc_ascgmtime(
    char *cp,
    size_t strlen,
    utc_t *utc);

Parameters
Input
strlen 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 Text string that represents the time.

Usage
The utc_ascgmtime API converts a binary timestamp to a text string that expresses a time in GMT.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that a time parameter or the results are not valid.

Examples
The following example converts the current time to GMT format.

char gmTime[UTC_MAX_STR_LEN];

/* Convert the current time to a text string in the following format:
 * 1991-04-01-12:27:38.37I2.qzerodotqzerodot
 */

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
Routines

utc_ascanytime          utc_asclocaltime
utc_asclocaltime

Purpose
Converts a binary timestamp to a text string that represents a local time.

Format
#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 Text string that represents the time.

Note: Output parameter is optional.

Usage
The utc_asclocaltime API converts a binary timestamp to a text string that expresses local time.

The user selects a time zone by specifying the TZ environment variable. For more information on setting the time zone, refer to z/OS DCE Administration Guide. If the user’s environment does not specify a time zone rule, the rule in the localtime file is used.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that a time parameter or the results are not valid.

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
Routines

utc_ascanytime     utc_ascgmtime
**utc_ascrreltime**

**Purpose**
Converts a relative binary timestamp to a text string that represents the time.

**Format**
```
#include <dce/utc.h>

int utc_ascrreltime(
    char *cp ,
    size_t stringlen ,
    utc_t *utc );
```

**Parameters**

**Input**
- *utc* Relative binary timestamp. Use **NULL** if you want this routine to use the current time for this parameter.
- *stringlen* Length of the *cp* buffer.

**Output**
- *cp* Text string that represents the time.

**Usage**
The **utc_ascrreltime** API converts a relative binary timestamp to a text string that represents the time.

**Return Values**
- **0** Indicates that the routine ran successfully.
- **-1** Indicates that a time parameter or the results are not valid.

**Examples**
See the sample program in "utc_abstime" on page 5-4.

**Related Information**

**Routines**
- **utc_mkascrtimere**
utc_binreltime

Purpose
Converts a relative binary timestamp to two timespec structures that express relative time and inaccuracy.

Format
#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+00:00).
inaccsp Inaccuracy component of the relative binary timestamp, in the form of seconds and nanoseconds.

Usage
The utc_binreltime API 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 ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
The following example measures the duration of a process, and then prints the resulting relative time and inaccuracy:
utc_binreltime

```c
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 time to timespec structures... */

utc_binreltime(&tduration, // Out: Duration time timespec */
    &duration, // 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");
else
    printf("I%d.%04d", iduration.tv_sec, (iduration.tv_nsec/1000000));
```

**Related Information**

**Routines**

- utc_mkbinreltime
utc_bintime

Purpose
Converts a binary timestamp to a timespec structure.

Format
#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.

Usage
The utc_bintime API converts a binary timestamp to a timespec structure. The TDF information contained in the timestamp is returned.

Return Values
0     Indicates that the routine ran successfully.
-1    Indicates that a time argument or the results are not valid.

Examples
See the sample program in "utc_anytime" on page 5-8

Related Information
Routines
utc_binreltime    utc_mkbintime
utc_boundtime

Purpose
Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event.

Format
#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.

Usage
Given two UTC times, the utc_boundtime API 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, and then calls utc_boundtime to build a timestamp that includes the event.

Note: The TDF in the output UTC value is copied from the utc2 input. If one or both input values have infinite 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 ran successfully.
-1              Indicates a time parameter or parameter order that is not valid.

Examples
The following example records the time of an event and constructs a single timestamp that includes the time of the event. Note that the utc_getusertime API is called so the time zone information that is included in the timestamp refers to the user’s environment rather than the system’s default time zone.

The user’s environment determines the time zone rule. For example, the user selects a time zone by specifying the TZ environment variable. For more information on setting the time zone, refer to z/OS DCE Administration Guide.

If the user’s environment does not specify a time zone rule, the rule in the localtime file is used.
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

Routines
utc_gettime  utc_pointtime  utc_spantime
utc_cmpintervaltime

Purpose
Compares two binary timestamps or two relative binary timestamps.

Format
#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

Usage
The utc_cmpintervaltime API 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. If the intervals (time - inaccuracy, time + inaccuracy) of the two times intersect, the two times overlap.

The input binary timestamps express two absolute or two relative times. Do not compare relative binary timestamps and 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
utc_cmpintervaltime

0 Indicates that the routine ran successfully.
-1 Indicates a time argument that is not valid.

Examples

The following example checks to see if the current time is definitely after 13:00 local time.

```c
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);

/* 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 p.m. 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, noting the use of the NULL argument... */
utc_cmpintervaltime(&relation,
    /* Out: Comparison relation */
    (utc_t *)0, /* In: Current timestamp */
    &testtime); /* In: 13:00 timestamp */
```
utc_cmpintervaltime

/* If it is not later - wait, print a message, and so on */

if (relation != utc_greaterThan) {

/*
 * Note: It could be earlier than 13:00 or it could be
 * indeterminate. If indeterminate, for some applications
 * it might be worth waiting.
 */
}

Related Information

Routines

utc_cmpmidtime
utc_cmpmidtime

Purpose
Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies.

Format

```c
#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

Usage

The **utc_cmpmidtime** API 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 and 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.

```c
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 ran successfully.
- **-1** Indicates a time argument that is not valid.
utc_cmpmidtime

Examples

The following example checks if the current time (ignoring inaccuracies) is after 13:00 local time.

```c
struct tm tmtime, tmzero;
enum utc_cmp_type 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, NULL);

/* 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, &tmtime, 0, &tmzero, 0);

/* Compare to the current time. Note the use of the NULL argument... */
utc_cmpmidtime(&relation, (utc_t *)&testtime, &tmtime);

/* If the time is not later - wait, print a message, and so on. */
if (relation != utc_greaterThan) {
    /* It is not later than 13:00 local time. Note that the result depends on the setting of the user's environment. */
}
```

Related Information

5-28 Application Development Reference Volumes 1 and 2
Routines

utc_cmpintervaltime
utc_gettime

Purpose
Returns the current DCE system time and inaccuracy as a binary timestamp.

Format
#include <dce/utc.h>

int utc_gettime(
    utc_t *utc);

Parameters
Input
None.
Output
utc DCE system time as a binary timestamp.

Usage
The utc_gettime API returns the current DCE system time and inaccuracy in a binary timestamp. The routine takes the TDF from the operating system-specific value; the TDF is specified in a system dependent manner.

Return Values
0 Indicates that the routine ran successfully.
-1 Generic error that indicates the time service cannot be accessed.

Examples
See the sample program in "utc_binreltime on page 5-19"
utc_getusertime

Purpose
Returns the time and process-specific TDF, rather than the system-specific TDF.

Format
#include <dce/utc.h>

int utc_getusertime(
    utc_t *utc);

Parameters
Output
 utc DCE system time as a binary timestamp.

Usage
The utc_getusertime API returns the DCE system time and inaccuracy in a binary timestamp. The routine takes the TDF from the user’s environment, which determines the time zone rule. The user selects a time zone by specifying the TZ environment variable. For more information on setting the time zone, refer to z/OS DCE Administration Guide. If the user’s environment does not specify a time zone rule, the rule in the localtime file is used.

Return Values
0 Indicates that the routine ran successfully.
-1 Generic error that indicates the time service cannot be accessed.

Examples
See the sample program in utc_boundtime on page 5-22

Related Information
Routines
utc_gettime
utc_gmtime

Purpose
Converts a binary timestamp to a tm structure that expresses GMT or the equivalent UTC.

Format
#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 Time component of the binary timestamp.
inacctm Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, tm_mday returns a value of -1 and tm_mon and tm_year return values of 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 inaccuracy component of the binary timestamp. If the inaccuracy is unspecified, ins returns a value of -1.

Usage
The utc_gmtime API converts a binary timestamp to a tm structure that expresses GMT (or the equivalent UTC). Additional returns include nanoseconds since Time component of the binary timestamp and nanoseconds of inaccuracy.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
See the sample program in "utc_cmpintervaltime" on page 5-24
Related Information

Routines

utc_anytime  utc_localtime  utc_mkgmttime
utc_gmtzone
Purpose
Gets the time zone label for GMT.

Format

```c
#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 zero is always returned.
- `isdst`: Integer with a value of zero, indicating that Daylight Savings Time is not in effect. A value of zero is always returned.

Usage

The `utc_gmtzone` API 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` APIs.

**Note:** 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 ran successfully (always returned).

Examples

The following example prints out the current time in both local time and GMT time:

```c
utc_t now;
struct tm tmlocal, tmgmt;
long tzoffset;
int tzdaylight;
char tizzly[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 binary timestamp */

/*  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 seconds */
    &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 seconds */
    (int *)0, /* Out: GMT time zone daylight flag */
    &now); /* In: Current binary timestamp */

/*  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)",
    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", tzlocal, tzoffset/60);
if (tzdaylight != 0)
    printf("Daylight savings time is in effect");

Related Information
utc_gmtzone

Routines
 utc_anyzone           utc_gmtime           utc_localzone
utc_localtime

Purpose
Converts a binary timestamp to a tm structure that expresses local time.

Format
#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 Time component of the binary timestamp.

  inacctm
  Seconds of inaccuracy component of the binary timestamp. If the inaccuracy is finite, tm_mday
  returns a value of -1 and tm_mon and tm_year return values of 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 inaccuracy component of the binary timestamp. If the inaccuracy is unspecified,
  ins returns a value of -1.

Note:  All output parameters are optional.

Usage
The utc_localtime API converts a binary timestamp to a tm structure that expresses local time. The user selects a time zone
by specifying the TZ environment variable. If the user’s environment does not specify a time zone rule, the rule in the
localtime file is used. Additional returns include nanoseconds since Time component of the binary timestamp, and
nanoseconds of inaccuracy.

Note:  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 ran successfully.
  -1   Indicates that a time argument or the results are not valid.

Examples
See the sample program in utc_gmtzone on page 5-34

Related Information
<table>
<thead>
<tr>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>utc_anytime</td>
</tr>
<tr>
<td>utc_gmtime</td>
</tr>
<tr>
<td>utc_localzone</td>
</tr>
<tr>
<td>utc_mklocaltime</td>
</tr>
</tbody>
</table>
utc_localzone

Purpose
Gets the local time zone label and offset from GMT, given utc.

Format

```c
#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 zero if Standard Time is in effect or a value of 1 if Daylight Savings Time is in effect.

Usage

The `utc_localzone` API gets the local time zone label and offset from GMT, given `utc`. The user selects a time zone by specifying the TZ environment variable. For more information on setting the time zone, refer to [z/OS DCE Administration Guide](#). If the user’s environment does not specify a time zone rule, the rule in the `localtime` file is used.

Notes:
1. All of the output parameters are optional. No value is returned, and no error occurs if the pointer is NULL.
2. Standard time is returned when the input time is before 1902, or after 2037.

Return Values

- 0 Indicates that the routine ran successfully.
- -1 Indicates that a time argument is not valid or an insufficient buffer.

Examples

See the sample program in "utc_gmtime" on page 5-34.

Related Information
utc_localzone

Routines

utc_anyzone  utc_gmtzone  utc localtime
utc_mkanytime

Purpose
Converts a tm structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp.

Format

```
#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 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. Otherwise, tm_yday expresses the inaccuracy in days. On input, tm_mday, tm_mon, tm_wday, tm_isdst, tm_gmtoff, and tm_zone are ignored.

- **ins**
  Nanoseconds of inaccuracy component.

- **tdf**
  Time Differential Factor to use in conversion.

**Output**

- **utc**
  Resulting binary timestamp.

Usage

The utc_mkanytime API converts a tm structure and TDF (expressing the time in an arbitrary time zone) to a binary timestamp. Required input includes nanoseconds since Time components, and nanoseconds of inaccuracy.

Return Values

- **0** Indicates that the routine ran successfully.
- **-1** Indicates that a time argument or the results are not valid.

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_mkanytime

utc_t utc;
struct tm tmtime, tminacc;
float tsec, isec;
double tmp;
long tnsec, insec;
int i, offset, tzhour, 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
Routines

 utc_anytime  utc_anyzone
utc_mkascreltime

Purpose
Converts a NULL-terminated text string that represents a relative timestamp to a binary timestamp.

Format
#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 the ISO format. For more information, refer to the section on Relative Time Representation in z/OS DCE Application Development Guide: Core Components.

Output
utc     Resulting binary timestamp.

Usage
The utc_mkascreltime API converts a NULL-terminated text string, which represents a relative timestamp, to a binary timestamp.

Note: The text string must be NULL-terminated.

Return Values
0     Indicates that the routine ran successfully.
-1    Indicates that a time parameter or the results are not valid.

Examples
The following example converts a relative-time text 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
Routines

utc_ascreltime
utc_mkasctime

Converts a NULL-terminated text string that represents an absolute timestamp to a binary timestamp.

Format

#include <dce/utc.h>

int utc_mkasctime(
    utc_t *utc,
    char *string);

Parameters

Input

string A NULL-terminated string that expresses an absolute time in its ISO format.

Output

utc Resulting binary timestamp.

Usage

The utc_mkasctime API converts a NULL-terminated text string that represents an absolute time to a binary timestamp.

Note: The text string must be NULL-terminated.

Return Values

0 Indicates that the routine ran successfully.
-1 Indicates that a time parameter or the results are not valid.

Examples

The following example converts a time text 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

Routines

utc_ascanytime         utc_ascgmtime         utc_asclocaltime
utc_mkbinreltime

Purpose
Converts a timespec structure expressing a relative time to a binary timestamp.

Format
#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.

Usage
The utc_mkbinreltime API converts a timespec structure that expresses relative time to a binary timestamp.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
See the sample program in the Examples section of "utc_addtime" on page 5-6.

Related Information
Routines
utc_binreltime utc_mkbinreltime utc_mkbintime
**utc_mkbintime**

**Purpose**
Converts a `timespec` structure to a binary timestamp.

**Format**

```c
#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+0:00I0.
- `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.

**Usage**
The `utc_mkbintime` API 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 ran successfully.
- -1 Indicates that a time argument or the results are not valid.

**Examples**
The following example obtains the current time from `time()`, converts it to a binary timestamp with an inaccuracy of 5.2 seconds, and specifies GMT:
utc_mkbintime

timespec_t ttime, tinacc;
tc_t utc;

/* Obtain the current time (without the inaccuracy)...
 */

time.tv_sec = time((time_t)0);
time.tv_nsec = 0;

/* Specify the inaccuracy... 
 */

tinacc.tv_sec = 5;
tinacc.tv_nsec = 200000000;

/* Convert to a binary timestamp... 
 */

utc_mkbintime(UTC, 
    &ttime, /* Out: Binary timestamp */
    &tinacc, /* In: Current time in timespec */
    0); /* In: 5.2 seconds in timespec */

/* TDF of GMT */

Related Information

Routines

utc_bintime
utc_mkbinreltime

5-50  Application Development Reference Volumes 1 and 2
**utc_mkgmtime**

**Purpose**
Converts a tm structure that expresses GMT or UTC to a binary timestamp.

**Format**
```
#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 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. Otherwise, tm_yday expresses the inaccuracy in days. On input, tm_mday, tm_mon, tm_wday, tm_isdst, tm_gmtoff, and tm_zone are ignored.

- **ins**
  Nanoseconds of inaccuracy component.

**Output**

- **utc**
  Resulting binary timestamp.

**Usage**
The utc_mkgmtime API converts a tm structure that expresses GMT or UTC to a binary timestamp. Additional input includes nanoseconds since the last second of Time and nanoseconds of inaccuracy.

**Return Values**

- **0** Indicates that the routine ran successfully.
- **-1** Indicates that a time argument or the results are not valid.

**Examples**
See the sample program in the Examples section of "utc_cmpintervaltime on page 5-24".

**Related Information**
utc_mkgmtime

Routines

utc_gmtime
utc_mklocaltime

Purpose
Converts a tm structure that expresses local time to a binary timestamp.

Format
```c
#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 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. Otherwise, `tm_yday` expresses the inaccuracy in days. On input, `tm_mday, tm_mon, tm_wday, tm_isdst, tm_gmtoff`, and `tm_zone` are ignored.
- `ins`: Nanoseconds of inaccuracy component.

Output
- `utc`: Resulting binary timestamp.

Note: Output parameter is optional.

Usage
The `utc_mklocaltime` API converts a tm structure that expresses local time to a binary timestamp. The user selects a time zone by specifying the TZ environment variable. For more information on setting the time zone, refer to z/OS DCE Administration Guide. If the user's environment does not specify a time zone rule, the rule in the `localtime` file is used. Additional input includes nanoseconds since the last second of Time and nanoseconds of inaccuracy.

Note: 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 ran successfully.
- -1: Indicates that a time argument or the results are not valid.

Examples
See the sample program in the Examples section of "utc_cmpmidtime" on page 5-27.

Related Information
utc_mklocaltime

Routines

utc_localtime
utc_mkreltime

Purpose
Converts a tm structure that expresses relative time to a relative binary timestamp.

Format
#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 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. Otherwise, tm_yday expresses the inaccuracy in days. On input, tm_mday, tm_mon, tm_year, tm_wday, tm_isdst, and tm_zone are ignored.

ins Nanoseconds of inaccuracy component.

Output
utc Resulting relative binary timestamp.

Usage
The utc_mkreltime API converts a tm structure that expresses relative time to a relative binary timestamp. Additional input includes nanoseconds since the last second of Time and nanoseconds of inaccuracy.

Return Values
0 Indicates that the routine ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
The following example converts the relative time: 125-03:12:30.1I120.25 to a relative binary timestamp. This may be part of an input relative timestamp routine, although a real implementation will include range checking.
utc_mkreltime

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 = 1000000000; /* .1 * 1.0E9 */
tminacc.tm_sec = 120;
insec = 250000000; /* .25 * 1.0E9 */
insec = (modf(isec, &tmp)*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 */

Related Information

Routines

utc_reltimel
utc_mulftime

Purpose
Multiplies a relative binary timestamp by a floating-point value.

Format
#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.

Usage
The utc_mulftime API 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 ran successfully.
-1 Indicates that a time argument or the results are not valid.

Examples
The following example scales a relative time by a floating-point factor and prints the result.
utc_multime

utc_t relutc, scaledutc;
struct tm scaledreltm;
char timstr[UTC_MAX_STR_LEN];

/* Assume relutc contains the time to scale. */

utc_multime(&scaledutc, /* Out: Scaled rel time */
   &relutc, /* In: Rel time to scale */
   17.65); /* In: Scale factor */

utc_ascreltime(timstr, /* Out: Rel time text string */
   UTC_MAX_STR_LEN, /* In: Input buffer length */
   &scaledutc); /* In: Rel time to convert */

printf("%s", timstr);

/* Convert it to a tm structure and print it. */

utc_reltm(&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",
   scaledreltm.tm_yday, scaledreltm.tm_hour, scaledreltm.tm_min);

Related Information

Routines

utc_multime
utc_multime

Purpose
Multiplies a relative binary timestamp by an integer factor.

Format
#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.

Usage
The utc_multime API 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 ran successfully.
-1 Indicates that a time argument or the results are not valid.

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: Rel time text string */
    UTC_MAX_STR_LEN, /* In: Input buffer length */
    &scaledutc); /* In: Rel time to convert */

printf("Scaled result is %s", timstr);
Related Information

Routines

utc_mulftime
utc_pointtime

Purpose
Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time.

Format

```c
#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 binary timestamp.
- `utchp`: Highest (latest) possible absolute time or longest possible relative time that the input timestamp can represent.

Usage

The `utc_pointtime` API 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 output represents relative binary times.

Note: All the output has zero inaccuracy. An error is returned if the input binary timestamp has an unspecified inaccuracy.

Return Values

- `0`: Indicates that the routine ran successfully.
- `-1`: Indicates that a time argument is not valid.

Examples

See the sample program in the Examples section of "utc_addtime" on page 5-6.

Related Information

- `utc_boundtime`
- `utc_spantime`
utc_reltime

**Purpose**
Converts a relative binary timestamp to a tm structure.

**Format**
```
#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 zero. The field tm_yday contains the number of days of relative time.

tns
Nanoseconds since Time component of the relative binary timestamp.

inacctm
Seconds of inaccuracy component of the relative binary timestamp. If the inaccuracy is finite, tm_mday returns a value of -1 and tm_mon and tm_year return values of 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 inaccuracy component of the relative binary timestamp.

**Usage**
The utc_reltime API 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 ran successfully.
-1 Indicates that a time argument or the results are not valid.

**Examples**
See the sample program in the Examples section of "utc_mulmtime" on page 5-57.

**Related Information**

5-62 Application Development Reference Volumes 1 and 2
Routines

utc_mkreltime
utc_spantime

__Purpose__
Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input binary timestamps.

__Format__
```c
#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.

__Usage__
Given two binary timestamps, the `utc_spantime` API 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).

**Note:** The `tdf` 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 ran successfully.
- **-1** Indicates that a time argument is not valid.

__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 *)&testtime, /* Out: Midpoint */
               &latest, /* Out: Latest poss time in array */
               &testtime); /* In: Spanning interval */

Related Information

Routines
utc_boundtime     utc_gettime     utc_pointtime
utc_subtime

Purpose
Computes the difference between two binary timestamps that express an absolute time and a relative time, two relative times, or two absolute times.

Format

```c
#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 on “Usage”.

Usage

The `utc_subtime` API subtracts one binary timestamp from another. 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.

**Note**: Although no error is returned, do not use the combination `relative time – absolute time`.

Return Values

- 0: Indicates that the routine ran successfully.
- -1: Indicates that a time argument or the results are not valid.

Examples

See the sample program in the Examples section of "utc_binreltime" on page 5-19.

Related Information
Routines

utc_addtime
Chapter 6. Security and Related Services .......................................................... 6-1
Chapter 7. Kerberos ....................................................................................... 7-1
Chapter 8. DCE Utility Routines ................................................................. 8-1
Chapter 6. Security and Related Services

This chapter contains information about the following sets of APIs:

- **dce_acl_***: Server ACL Manager APIs
- **dce_aud_***: DCE Audit Service APIs
- **gss_***: Generic Security Service API (standard) APIs
- **gssdce_***: Generic Security Service API (DCE extensions) APIs
- **rdacl_***: ACL Network APIs
- **sec_***: DCE Security Service APIs
General ACL Manager Interface APIs

This section describes in detail the following server ACL manager library APIs:

- `dce_acl_copy_acl`
- `dce_acl_inq_acl_from_header`
- `dce_acl_inq_client_creds`
- `dce_acl_inq_client_permset`
- `dce_acl_inq_permset_for_creds`
- `dce_acl_is_client_authorized`
- `dce_acl_obj_add_any_other_entry`
- `dce_acl_obj_add_foreign_entry`
- `dce_acl_obj_add_group_entry`
- `dce_acl_obj_add_id_entry`
- `dce_acl_obj_add_obj_entry`
- `dce_acl_obj_add_unauth_entry`
- `dce_acl_obj_add_user_entry`
- `dce_acl_obj_free_entries`
- `dce_acl_obj_init`
- `dce_acl_register_object_type`
- `dce_acl_resolve_by_name`
- `dce_acl_resolve_by_uuid`
dce_acl_copy_acl

**Purpose**
Copies an ACL.

**Format**
```c
#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. Possible status codes and their meanings are:
  - `rpc_s_no_memory` The `rpc_sm_allocate()` routine could not obtain memory.

**Usage**
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.

**Related Information**

**Routines**
- `dce_acl_obj_free_entries`
Purpose
Retrieves the UUID of an ACL from an item's header in a backing store.

Format
```c
#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`, or `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. Possible status codes and their meanings are:
  - `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_acl_type`: The `sec_acl_type` parameter does not contain a valid type.

Usage
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.

Related Information
Routines
- `dce_acl_resolve_by_uuid`
dce_acl_inq_client_creds

Purpose
Returns the client’s credentials.

Format
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_client_creds(
    handle_t handle,
    rpc_authz_cred_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. Possible status codes and their meanings are:
    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().

Usage
The dce_acl_inq_client_creds() routine returns the client’s security credentials found through the RPC binding handle.

Related Information
Routines
dce_acl_inq_permset_for_creds dce_acl_register_object_type
dce_acl_inq_client_permset

Purpose
Returns the client's permissions corresponding to an ACL.

Format
#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_objACLE (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
sec_acl_e_type_group_objACLE 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. Possible status codes and their meanings are:

acl_s_bad_manager_type
The mgr_type parameter does not match the manager
type in the ACL itself.

Usage
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.

Related Information
Routines

dce_acl_inq_permset_for_creds  dce_acl_register_object_type
Purpose
Determines the complete extent of access to an object for the credentials presented.

Format
```c
#include <dce/dce.h>
#include <dce/aclif.h>

void dce_acl_inq_permset_for_creds(
    rpc_authz_cred_handle_t creds,
    sec_acl_t acl,
    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`: A credential handle of type `rpc_authz_cred_handle_t`. This handle is supplied as output of the `dce_acl_inq_client_creds` call.
- `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` ACL entry exists, then the `owner_id` (uuid_t pointer) cannot 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 `sec_acl_e_type_group_obj` ACL entry exists, the `group_id` (uuid_t pointer) cannot be NULL. If it is, the error `sec_acl_expected_group_obj` is returned.
- `posix_semantics`: This parameter is currently unused.

**Output**
- `perms`: A bit mask containing a 1 bit for each permission granted by the ACL and 0 bits elsewhere.
- `status`: A pointer to the completion status. On successful completion, the routine returns `error_status_ok`.

Usage
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 following ACL entry types (of type `sec_acl_entry_type_t`) are supported by this routine. The categories are checked in the order shown.
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. The final permission is the intersection of the permission of the initiator and of each delegate.

### Table 6-1. Supported ACL Entry Types

<table>
<thead>
<tr>
<th>ACL Entry Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_acl_e_type_user_obj</td>
<td>At most, one can match.</td>
</tr>
<tr>
<td>sec_acl_e_type_user_obj_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_user</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_user_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_foreign_user</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_for_user_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_group_obj</td>
<td>The union of all permissions for each matching group.</td>
</tr>
<tr>
<td>sec_acl_e_type_group_obj_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_group</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_group_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_foreign_group</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_for_group_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_other_obj</td>
<td>Matching local realm accesses.</td>
</tr>
<tr>
<td>sec_acl_e_type_other_obj_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_foreign_other</td>
<td>At most, one can match.</td>
</tr>
<tr>
<td>sec_acl_e_type_for_other_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_any_other</td>
<td>Anything not in the other types.</td>
</tr>
<tr>
<td>sec_acl_e_type_any_other_deleg</td>
<td></td>
</tr>
<tr>
<td>sec_acl_e_type_unauthenticated</td>
<td>The <code>sec_acl_e_type_unauthenticated</code> type is a mask that is used for all matches on unauthenticated credentials. It is also intersected with <code>sec_acl_e_type_any_other</code> for NULL credentials.</td>
</tr>
</tbody>
</table>

### 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. The final permission is the intersection of the permission of the initiator and of each delegate.

### Related Information

#### Routines

- `dce_acl_inq_client_creds`
- `dce_acl_inq_client_permset`
- `dce_acl_register_object_type`
Purpose
Inquires about the principal and group of an RPC caller.

Format
```c
#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.

Usage
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
```c
dce_db_std_header_init(db, &data, ..., &st);
dce_acl_inq_prin_and_group(h, 73548dhh.owner_id,
    73548dhh.group_id, &st);
```

Error Conditions
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
Routines

dce_acl_register_object_type
**dce_acl_is_client_authorized**

**Purpose**
Checks whether a client's credentials are authenticated.

**Format**
```c
#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,
    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.
- `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. Possible status codes and their meanings are:
  - `acl_s_bad_manager_type` The `mgr_type` does not match the manager type in the ACL itself.

**Usage**
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.
dce_acl_obj_add_any_other_entry

Purpose
Adds permissions for any_other ACL entry to a given ACL.

Format
#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.

Usage
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.

Related Information
Routines
dce_acl_obj_add_obj_entry
dce_acl_obj_add_foreign_entry

Purpose
Adds permissions for an ACL entry for a foreign user or group to the given ACL.

Format
#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. Possible status codes and their meanings are:
- sec_acl_invalid_entry_type The type specified in entry_type is not one of the four specified types.

Usage
The dce_acl_obj_add_foreign_entry() routine adds an ACL entry for sec_acl_e_type_foreign_xxx access to the specified ACL.

Related Information
Routines
dce_acl_obj_add_id_entry sec_id_parse_name
**dce_acl_obj_add_group_entry**

**Purpose**
Adds permissions for a group ACL entry to the given ACL.

**Format**
```c
#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.

**Usage**
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.

**Related Information**

**Routines**
- `dce_acl_obj_add_id_entry`
Purpose
Adds permissions for an ACL entry to the given ACL.

Format
#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. Possible status codes and their meanings are:
    sec_acl_invalid_entry_type The type specified in entry_type is not one of the six
    specified types.

Usage
The dce_acl_obj_add_id_entry() routine adds an ACL entry (user or group, domestic or foreign) to the given ACL.

Related Information
Routines
dce_acl_obj_add_group_entry dce_acl_obj_add_user_entry
**dce_acl_obj_add_obj_entry**

**Purpose**
Adds permissions for an object (obj) ACL entry to the given ACL.

**Format**
```c
#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_user_obj_deleg`
    - `sec_acl_e_type_group_obj_deleg`
    - `sec_acl_e_type_other_obj_deleg`
    - `sec_acl_e_type_any_other_deleg`

**Output**
- **status**
  - A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:
    - `sec_acl_invalid_entry_type` The type specified in `entry_type` is not one of the six specified types.

**Usage**
The `dce_acl_obj_add_obj_entry()` routine adds an obj ACL entry to the given ACL.

**Related Information**

**Routines**
- `dce_acl_obj_add_any_other_entry`
- `dce_acl_obj_add_unauth_entry`
dce_acl_obj_add_unauth_entry

**Purpose**

Adds permissions for **unauthenticated** ACL entry to the given ACL.

**Format**

```
#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.

**Usage**

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.

**Related Information**

**Routines**

- `dce_acl_obj_add_unauth_entry`

- `dce_acl_obj_add_obj_entry`
dce_acl_obj_add_user_entry

Purpose
Adds permissions for a user ACL entry to the given ACL.

Format
#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.

Usage
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.

Related Information

Routines
dce_acl_obj_add_id_entry
dce_acl_obj_free_entries

Purpose
Frees space used by an ACL's entries.

Format
#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.

Usage
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).
dce_acl_obj_init

**Purpose**
Initializes an ACL.

**Format**
```
#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.

**Usage**
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);
```
**dce_acl_register_object_type**

**Purpose**
Registers an ACL manager's object type.

**Format**
```c
#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.
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.

A similar bit for group owners of objects.

A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns an error. One possible status code and its meaning is:

The mgr_type parameter does not point to a valid manager type.

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.

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.


dce_acl_register_object_type

#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_00000000 }, { "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, &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);

Related Information

Routines
dce_acl_resolve_by_name dce_acl_resolve_by_uuid

Files
/usr/include/dce/aclif.h Definition of dce_acl_resolve_func_t.
dce_acl_resolve_by_name

Purpose
Finds an ACL's UUID, given an object's name.

Format
```c
#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`, or `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.
- `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.

Usage

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.

Related Information
dce_acl_resolve_by_name

Routines

dce_db_open               dce_acl_register_object_type               dce_acl_resolve_by_uuid
**dce_acl_resolve_by_uuid**

**Purpose**
Finds an ACL's UUID, given an object's UUID.

**Format**
```c
#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`, or `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.

**Usage**
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.

**Related Information**
<table>
<thead>
<tr>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dce_db_open</code></td>
</tr>
<tr>
<td><code>dce_acl_register_object_type</code></td>
</tr>
<tr>
<td><code>dce_acl_resolve_by_name</code></td>
</tr>
</tbody>
</table>
Introduction to the DCE Audit API Runtime

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:

- Audit API routines
- Audit data types
- Environment variables
- Permissions required

Audit API Routines

The following is an alphabetical list of the Audit API routines. With each routine name is its description. The types of application programs that are most likely to 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()**
  Writes the audit record to the audit trail file (client/server applications).

- **dce_aud_discard()**
  Frees the memory allocated to an audit record (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()` routine (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).
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 API routines allow applications to include event-specific information in audit records. Event-specific information must be represented as information items using the following data type.

```c
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_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.
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.

```c
typedef struct {
    unsigned16 format;
    uuid_t server;
    unsigned32 event;
    unsigned16 outcome;
    unsigned16 authz_st;
    unsigned16 num_client_ids;
    uuid_t client;
    uuid_t cell;
    unsigned16 num_groups;
    uuid_t *groups;
    utc_t time;
    idl_char *addr;
} dce_aud_hdr_t;
```

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Data Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>aud_c_evt_info_small_int</td>
<td>idl_small_int</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_short_int</td>
<td>idl_short_int</td>
<td>2 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_long_int</td>
<td>idl_long_int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_hyper_int</td>
<td>idl_hyper_int</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_usmall_int</td>
<td>idl_usmall_int</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_ushort_int</td>
<td>idl_ushort_int</td>
<td>2 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_ulong_int</td>
<td>idl_ulong_int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_uhyper_int</td>
<td>idl_uhyper_int</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_short_float</td>
<td>idl_short_float</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_long_float</td>
<td>idl_long_float</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_boolean</td>
<td>idl_boolean</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_uuid</td>
<td>uuid_t</td>
<td>16 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_utc</td>
<td>utc_t</td>
<td>16 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_acl</td>
<td>sec_acl_t *</td>
<td>variable size</td>
</tr>
<tr>
<td>aud_c_evt_info_byte_string</td>
<td>idl_byte *</td>
<td>variable size</td>
</tr>
<tr>
<td>aud_c_evt_info_char_string</td>
<td>idl_char *</td>
<td>variable size</td>
</tr>
</tbody>
</table>

Table 6-2. Event Data Format Specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Data Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>aud_c_evt_info_small_int</td>
<td>idl_small_int</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_short_int</td>
<td>idl_short_int</td>
<td>2 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_long_int</td>
<td>idl_long_int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_hyper_int</td>
<td>idl_hyper_int</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_usmall_int</td>
<td>idl_usmall_int</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_ushort_int</td>
<td>idl_ushort_int</td>
<td>2 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_ulong_int</td>
<td>idl_ulong_int</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_uhyper_int</td>
<td>idl_uhyper_int</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_short_float</td>
<td>idl_short_float</td>
<td>4 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_long_float</td>
<td>idl_long_float</td>
<td>8 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_boolean</td>
<td>idl_boolean</td>
<td>1 byte</td>
</tr>
<tr>
<td>aud_c_evt_info_uuid</td>
<td>uuid_t</td>
<td>16 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_utc</td>
<td>utc_t</td>
<td>16 bytes</td>
</tr>
<tr>
<td>aud_c_evt_info_acl</td>
<td>sec_acl_t *</td>
<td>variable size</td>
</tr>
<tr>
<td>aud_c_evt_info_byte_string</td>
<td>idl_byte *</td>
<td>variable size</td>
</tr>
<tr>
<td>aud_c_evt_info_char_string</td>
<td>idl_char *</td>
<td>variable size</td>
</tr>
</tbody>
</table>

Chapter 6. Security and Related Services
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 \texttt{utc_t} type that records the time when the server committed the audit record (that is, after providing the event information through audit API routine 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 \texttt{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 \texttt{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, \texttt{dce_aud_rec_t}, is used to represent an audit record descriptor. An audit record descriptor can be manipulated by the following routines:

These routines return a record descriptor:

- \texttt{dce_aud_start()}
- \texttt{dce_aud_start_with_pac()}
- \texttt{dce_aud_start_with_name()}
- \texttt{dce_aud_start_with_server_binding()}
- \texttt{dce_aud_next()}

- \texttt{dce_aud_put_ev_info()} adds event information to an audit record through a record descriptor.

- These routines get the event and record information through a record descriptor:
  - \texttt{dce_aud_get_header()}
  - \texttt{dce_aud_get_ev_info()}
  - \texttt{dce_aud_length()}

- \texttt{dce_aud_commit()} commits an audit record through its record descriptor.

- \texttt{dce_aud_discard()} disposes of a record descriptor. This routine is used to free a record descriptor obtained by a call to the \texttt{dce_aud_next()} routine.

### Audit Trail Descriptor

An opaque data type, \texttt{dce_aud_trail_t}, is used to represent an audit trail descriptor. The \texttt{dce_aud_open} routine opens an audit trail and returns a trail descriptor; \texttt{dce_aud_next()} obtains an audit record from this descriptor; and \texttt{dce_aud_commit()} commits an audit record from and to an opened audit trail through this descriptor. The \texttt{dce_aud_close()} routine 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 \texttt{dce_aud_set_trail_size_limit()} routine.
Permissions Required

To use the Audit daemon's audit record logging service to write records to the central audit file, you need the log (l) permission to the Audit daemon's ACL (Access Control List).
dce_aud_close

Purpose
Closes an audit trail file.

Used by client/server applications and audit trail analysis and examination tools.

Format
```c
#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
A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

- `aud_s_invalid_trail_descriptor` The audit trail descriptor is not valid.

Usage

The `dce_aud_close` routine closes the audit trail file and releases any memory associated with the audit trail file that is specified by the audit trail descriptor.

Return Values

None.

Related Information

Routines: `dce_aud_open`
dce_aud_commit

Purpose
Writes the audit record in the audit trail file.
Used by client/server applications.

Format
#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 routine.

ard
Designates an audit record descriptor that was returned by a previously successful call to one of the dce_aud_start_* routines. 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 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  Flushed the audit
    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.
    aud_c_evt_commit_sync_no_wait  Flush the audit record to stable storage. Returns an error status if the storage or audit logging service is not available.

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 values are:

    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. In this case, the outcome is one in a series of connected events, and the application waits to record the real outcome only after the last event.

Output
status
A pointer to the completion status code. On successful completion, the routine returns aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:
dce_aud_commit

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aud_s_cannot_getgmtime</td>
<td>The audit library cannot backup a trail file due to failure of the utc_gmtime() call.</td>
</tr>
<tr>
<td>aud_s_cannot_gettime</td>
<td>The audit library cannot backup a trail file due to failure of the utc_gettime() call.</td>
</tr>
<tr>
<td>aud_s_cannot_reopen_trail_file_rc</td>
<td>Internally, the audit trail file was being reopened and the reopening of the file failed.</td>
</tr>
<tr>
<td>aud_s_dmn_disabled</td>
<td>The daemon is disabled for logging.</td>
</tr>
<tr>
<td>aud_s_invalid_outcome</td>
<td>The event outcome parameter that was provided is invalid.</td>
</tr>
<tr>
<td>aud_s_invalid_record_descriptor</td>
<td>The audit record descriptor is not valid.</td>
</tr>
<tr>
<td>aud_s_log_access_denied</td>
<td>The client's access to the Audit log was denied.</td>
</tr>
<tr>
<td>aud_s_outcomes_inconsistent</td>
<td>The event outcome parameter is inconsistent with the outcome parameter provided in the dce_aud_start() call.</td>
</tr>
<tr>
<td>aud_s_rename_trail_file_rc</td>
<td>Cannot rename the audit trail file.</td>
</tr>
<tr>
<td>aud_s_rename_trail_index_file_rc</td>
<td>Internally, the audit trail index file was being renamed and the renaming of the file failed.</td>
</tr>
<tr>
<td>aud_s_trl_write_failure</td>
<td>The audit record cannot be written to stable storage.</td>
</tr>
<tr>
<td>aud_s_wrong_protection_level</td>
<td>Client used the wrong protection level.</td>
</tr>
</tbody>
</table>

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()

Usage

The dce_aud_commit() routine determines whether the event should be audited given the event outcome. If it should be audited, the routine 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 specified, the event is always audited (logged or an alarm message is sent to the standard output and to the operator console).

If the aud_c_evt_commit_sync option is selected, the routine attempts to flush the audit record to stable storage. If the stable storage write cannot be performed, the routine 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_failure.

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 routine. If the final outcome can be aud_c_esl_cond_success, then it should be specified in this routine.

Return Values

None.
### Related Information

**Routines**

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_aud_put_ev_info</td>
<td>dce_aud_start_with_pac</td>
<td>dce_aud_start_with_server_binding</td>
</tr>
<tr>
<td>dce_aud_start</td>
<td>dce_aud_start_with_name</td>
<td>dce_aud_open</td>
</tr>
</tbody>
</table>
dce_aud_discard

Purpose
Discards an audit record (releases the memory).

Used by client/server applications, as well as trail analysis and examination tools.

Format
#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_* routines or the dce_aud_next() routine.

Output

status
A pointer to the completion status code. On successful completion, the routine returns
aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:
Status codes passed from dce_aud_free_header().

Usage

The dce_aud_discard() routine releases the memory used by the audit record descriptor and the associated audit record that
is to be discarded.

Return Values

None.

Related Information

Routines

dce_aud_next

dce_aud_start

dce_aud_free_header

dce_aud_start_with_pac

dce_aud_start_with_name

dce_aud_start_with_server_binding

dce_aud_open
**dce_aud_free_ev_info**

**Purpose**
Frees the memory allocated for an event information structure returned from calling `dce_aud_get_ev_info()`.

Used by the audit trail analysis and examination tools.

**Format**
```
#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 previously successful call to the `dce_aud_get_ev_info()` routine.

**Output**
- `status` A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`.

**Usage**
The `dce_aud_free_ev_info()` routine frees the memory allocated for an event information structure returned by a previously successful call to the `dce_aud_get_ev_info()` routine.

**Return Values**
None.

**Related Information**

**Routines**
- `dce_aud_get_ev_info`
- `dce_aud_next`
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.

Format
#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 previously successful call to the dce_aud_get_header() routine.

Output
    status          A pointer to the completion status code. On successful completion, the routine returns aud_s_ok.

Usage
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
None.

Related Information
Routines
dce_aud_next        dce_aud_open        dce_aud_get_header
dce_aud_get_ev_info

Purpose
Returns a pointer to an event information structure (dce_aud_ev_info_t).
Used by the audit trail analysis and examination tools.

Format
#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() routine.

Output
event_info
Returns an event-specific information item of the designated audit record. Returns NULL if there
are no more information items.

status
A pointer to the completion status code. On successful completion, the routine returns
aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

    aud_s_invalid_record_descriptor The audit record descriptor is not valid.

Usage
The dce_aud_get_ev_info() routine 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 routine allocates the memory to hold the human-readable representation of the audit record and returns the address
of this memory.

Memory allocated by dce_aud_get_ev_info() is freed by a call to dce_aud_free_ev_info().

Return Values

None.

Related Information

Routines

dce_aud_next          dce_aud_free_ev_info          dce_aud_put_ev_info
dce_aud_open
dce_aud_get_header

Purpose
Gets the header of a specified audit record.
Used by the audit trail analysis and examination tools.

Format
#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() routine.

Output
default
header
Returns the header information of the designated audit record.

status
A pointer to the completion status code. On successful completion, the routine returns
aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

   aud_s_invalid_record_descriptor The audit record descriptor is not valid.

Usage
The dce_aud_get_header() routine 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().

This routine allocates the memory to hold the header information. This memory is freed by a call to dce_aud_free_header().

Return Values
None.

Related Information

Routines

dce_aud_free_header dce_aud_next dce_aud_open

6-42   Application Development Reference Volumes 1 and 2
dce_aud_length

Purpose
Gets the length of a specified audit record.

Used by client/server applications, as well as trail analysis and examination tools.

Format
```
#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_*() routines.

Output
status
A pointer to the completion status code. On successful completion, the routine returns
aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

- aud_s_invalid_record_descriptor
  The audit record descriptor is not valid.

Status codes passed from idl-es_encode_dyn_buffer().

Status codes passed from idl_es_handle_free().

Status codes passed from rpc_sm_client_free().

Usage
The dce_aud_length() routine 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() routines.

Applications can use this routine to know how much space an audit record will use before it is committed. This routine 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.

Related Information

Chapter 6. Security and Related Services  6-43
dce_aud_length

Routines

<table>
<thead>
<tr>
<th>dce_aud_next</th>
<th>dce_aud_start_with_name</th>
<th>dce_aud_start_with_server_binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_aud_open</td>
<td>dce_aud_start_with_pac</td>
<td>dce_aud_put_ev_info</td>
</tr>
<tr>
<td>dce_aud_start</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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.

**Format**
```c
#include <dce/audit.h>

void dce_aud_next(  
    dce_aud_trail_t at,  
    char *predicate,  
    unsigned16 format,  
    dce_aud_rec_t *ord,  
    unsigned32 *status);
```

**Parameters**

**Input**
- **at**
  A pointer to the descriptor of an audit trail file previously opened for reading by the `dce_aud_open()` routine.
- **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), and > (greater than):
  - `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, of the 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:

**SERVER**
The UUID of the server principal that generated the record. The attribute value must be a UUID string. Operator allowed: `=`.

**EVENT**
The audit event number. The attribute value must be an integer. Operator allowed: `=`.

**OUTCOME**
The event outcome of the record. The possible attribute values are `SUCCESS`, `FAILURE`, or `PENDING`, or `DENIAL`. Operator allowed: `=`.

**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: `=`.

**CLIENT**
The UUID of the client principal. The attribute value must be a UUID string. Operator allowed: `=`.

**TIME**
The time the record was generated. The attribute value must be a null-terminated string that expresses an absolute time. Operators allowed: `= <`, and `=`. 
The `dce_aud_next()` routine attempts to read the next record from the audit trail that is specified by the audit trail descriptor, `at`. This routine 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()` routine 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 routine returns the next record read from the audit trail.

If no record satisfies the predicate specified for the call, a value of `NULL` is returned in `ard`.

The value returned in `ard` can be supplied as an input parameter to the routines `dce_aud_get_header()`, `dce_aud_length()`, `dce_aud_get_ev_info()`, and `dce_aud_discard()`.

Storage allocated by this routine must be explicitly freed by a call to `dce_aud_discard()` with `ard` as the input parameter.

If the routine 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.

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

None.

### Related Information
Routines

<table>
<thead>
<tr>
<th>dce_aud_get_ev_info</th>
<th>dce_aud_length</th>
<th>dce_aud_open</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_aud_get_header</td>
<td>dce_aud_discard</td>
<td></td>
</tr>
</tbody>
</table>
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.

Format
#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_s_trl_open_write_no_filter
    • aud_c_trl_ss_save (default)
    • 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() routine.
status    A pointer to the completion status code. On successful completion, the routine returns aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:
    aud_s_cannot_allocate_memory    Memory allocation failed.
    aud_s_cannot_init_trail_mutex    Audit trail mutex initialization failed.
    aud_s_cannot_mkdir    Cannot create a directory for storing the bindings file for the filter update notification interface.
    aud_s_cannot_open_dmn_binding_file    The local audit daemon trail file is designated, but the daemon's binding file cannot be opened.
    aud_s_cannot_open_dmn_identity_file    The local audit daemon trail file is designated, but the daemon's identity file cannot be opened.
    aud_s_cannot_open_trail_file_rc    Cannot open a local trail file.
    aud_s_trl_invalid_open_flags    The flags argument must include either aud_c_trl_open_read or aud_c_trl_open_write mode flag, but not both.
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.

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()`.

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()`.

Usage
The `dce_aud_open()` routine opens the audit trail file specified by the `description` parameter. If `description` is NULL, the routine uses the default audit trail which is an RPC interface to the local Audit daemon.

This routine 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()` routine. 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()` routine. 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
None.

Related Information
Routines

- `dce_aud_start_with_server_binding`
- `dce_aud_start_with_name`
- `dce_aud_open`
- `dce_aud_commit`
- `dce_aud_start_with_pac`
- `dce_aud_next`
Environment Variables

DCEAUDITFILTERON indicates filters should be applied to the audit record to determine if it is an auditable event. A process can only open one audit trail file for write with filters.

DCEAUDITTRAILSIZE sets maximum audit file size

DCEAUDITOFF turns auditing off
**dce_aud_print**

**Purpose**
Formats an audit record into human-readable form.

Used by audit trail examination and analysis tools.

**Format**

```c
#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_` routines.

- `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_delegates_info`: Includes delegation information.
  - `aud_c_evt_groups_info`: Includes the groups' information.
  - `aud_c_evt_address_info`: Includes the address information.
  - `aud_c_evt_raw_info`: Includes raw 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`:
  A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:
  - `aud_s_cannot_allocate_memory`: The `malloc()` call failed.
  - `aud_s_invalid_record_descriptor`: Record descriptor is invalid.

Status codes passed from `sec_rgy_site_open()`.

Status codes passed from `sec_login_get_current_context()`.

Status codes passed from `sec_login_inquire_net_info()`.
dce_aud_print

Usage

The dce_aud_print() routine 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 routine 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 for options will result in default operation, that is, no group, address, and event-specific information is included in the output string.

Return Values

None.

Related Information

Routines

dce_aud_next        dce_aud_start_with_pac
dce_aud_open        dce_aud_start_with_name
dce_aud_start       dce_aud_start_with_server_binding
dce_aud_put_ev_info
dce_aud_put_ev_info

Purpose
Adds event-specific information to a specified audit record buffer.

Used by client/server applications.

Format
#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_* routines.
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 "Audit Data Types" on page 6-30.

Output
status A pointer to the completion status code. On successful completion, the routine returns aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

aud_s_invalid_record_descriptor The input audit record descriptor is not valid.
aud_s_evt_tail_info_exceeds_limit The tail portion of the audit trail record has exceeded its limit of 4K.

Usage
The dce_aud_put_ev_info() routine 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() routine. This order is also observed by the dce_aud_print() routine. The info parameter is a pointer to an instance of the self-descriptive dce_aud_ev_info_t structure.

Return Values
None.

Related Information

Routines
dce_aud_start dce_aud_start_with_pac dce_aud_commit
dce_aud_start_with_name dce_aud_start_with_server_binding dce_aud_open
dce_aud_set_trail_size_limit

Purpose
Sets a limit to the audit trail size.
Used by client/server applications.

Format
```
#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 routine `dce_aud_open()`.

`file_size_limit_value` The desired maximum size of the audit trail file, in bytes.

Output
`status` A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

- `aud_s_invalid_trail_descriptor` The audit trail descriptor `at` is null.

Usage
The `dce_aud_set_trail_size_limit()` routine can be used by an application to set the maximum size of the audit trail. This routine 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 routine. Any of the values set by `DCEAUDITTRAILSIZE` or this routine overrides the hardcoded default.

When the size limit is reached and the audit trail was open with the save storage flag, the current trail file is copied to another file. The name of this new file is the original file name 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:

`central_trail.1994-09-26-16-38-15`

and


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 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`).

If the audit trail file was open with the wrap storage flag and the size limit is reached, the audit trail contents are truncated and auditing continues with an empty file.
Related Information

Routines

dce_aud_open

Environment Variable

DCEAUDITTRAILSIZE sets maximum audit file size
dce_aud_start

Purpose
Determines whether a specified event should be audited given the client binding information and the event outcome. If the event must be audited, or if the outcome of the event is not yet known, the memory of the audit record descriptor is allocated and the address to this structure is returned to the caller.

Used by client/server applications.

Format
```c
#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 wanted (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 system console (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
  - **aud_c_evt_delegates_info** Includes delegation information.
  - **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.
dce_aud_start

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. In this case, the outcome is one in a series of connected events, and the application waits 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 routine is called with outcome set to aud_c_esl_cond_unknown, it is possible that the routine 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() routine.

Storage for ard will be freed by dce_aud_commit(). If dce_aud_commit() can not be invoked due to error before the start of the commit, issue dce_aud_discard() to free the storage.

status
A pointer to the completion status code. On successful completion, the routine returns aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

Status codes passed from rpc_binding_inq_auth_caller().

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_client_princ_name().

Status codes passed from sec_cred_get_initiator().

Status codes passed from sec_cred_get_v1_pac().

Status codes passed from sec_cred_init_cursor().

Status codes passed from dce_aud_start_with_pac().

Status codes passed from sec_cred_get_delegate().

Usage
The dce_aud_start() routine 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 routine 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 has occurred, a NULL pointer is returned in ard. If the aud_c_evt_always_log or aud_c_evt_always_alarm option is specified, an audit record descriptor will always be created and returned.
The `dce_aud_start()` routine 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 routine 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()` routine. This routine 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 routine 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 routine and the outcome does not require any action according to filters, then this routine 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 routine. If the final outcome can be `aud_c_esl_cond_success`, then it should be specified in this routine, or use `aud_c_esl_cond_unknown`.

This routine 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()` routine 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**

None.

**Related Information**

**Routines**

- `dce_aud_open`
- `dce_aud_start_with_server_binding`
- `dce_aud_start_with_pac`
- `dce_aud_start_with_name`
- `dce_aud_put_ev_info`
- `dce_aud_commit`
dce_aud_start_with_name

Purpose
Determines whether a specified event should be audited given the client/server name 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.

Used by non-RPC based client/server applications that do not use the DCE authorization model.

Format
#include <dce/audit.h>

void dce_aud_start_with_name(
    unsigned32 event,
    unsigned_char_t *client,
    unsigned_char_t *address,
    unsigned32 options,
    unsigned32 outcome,
    dce_aud_rec_t *ord,
    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 wanted (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 system console (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.
**dce_aud_start_with_name**

**aud_c_esl_cond_pending**  
The event is in an intermediate state, and the outcome is pending. In this case, the outcome is one in a series of connected events, and the application waits 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 routine is called with `outcome` set to `aud_c_esl_cond_unknown`, the routine 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()` routine.

Storage for `ard` will be freed by `dce_aud_commit()`. If `dce_aud_commit()` can not be invoked due to error before the start of the commit, issue `dce_aud_discard()` to free the storage.

**status**  
A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

- 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()`.

**Usage**

The `dce_aud_start_with_name()` routine 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 routine 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 specified, an audit record descriptor will always be created and returned.

The `dce_aud_start_with_name()` routine 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 routine 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()` routine, 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 routine 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 routine and the outcome does not require any action according to filters, then this routine 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.

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`.

6-60  Application Development Reference Volumes 1 and 2
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 routine. If the final outcome can be `aud_c_esl_cond_success`, then it should be specified in this routine, or use `aud_c_esl_cond_unknown`.

This routine 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()` routine 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**

None.

**Related Information**

**Routines**

- `dce_aud_open`
- `dce_aud_start`
- `dce_aud_start_with_server_binding`
- `dce_aud_start_with_pac`
- `dce_aud_put_ev_info`
- `dce_aud_commit`
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. 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.

Used by non-RPC based client/server applications that use the DCE authorization model.

Format
```
#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_groups_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 system console (`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.
The event is in an intermediate state, and the outcome is pending. In this case, the outcome is one in a series of connected events, and the application waits to record the real outcome only after the last event.

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 routine is called with outcome set to `aud_c esl cond unknown`, it is possible that the routine 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()` routine.

Storage for `ard` will be freed by `dce_aud_commit()`. If `dce_aud_commit()` can not be invoked due to error before the start of the commit, issue `dce_aud_discard()` to free the storage.

*status* A pointer to the completion status code. On successful completion, the routine returns `aud_s ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

Status codes passed from `sec_rgy_site_open()`.

Status codes passed from `sec_rgy_properties_get_info()`.

Status codes passed from `uuid_create_nil()`.

**Usage**

The `dce_aud_start_with_pac()` routine 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 routine 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 specified, then an audit record descriptor will always be created and returned.

The `dce_aud_start_with_pac()` routine 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 routine 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()` routine. This routine can be called multiple times after calling any of the `dce_aud_start_*` routines 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()` routine.

This routine 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 routine and the outcome does not require any action according to filters, then this routine 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 routine. If the final outcome can be `aud_c_esl_cond_success`, then it should be specified in this routine, or use `aud_c_esl_cond_unknown`.

This routine 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()` routine 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**

None.

**Related Information**

**Routines**

- `dce_aud_open`
- `dce_aud_start`
- `dce_aud_start_with_server_binding`
- `dce_aud_start_with_name`
- `dce_aud_put_ev_info`
- `dce_aud_commit`
**dce_aud_start_with_server_binding**

**Purpose**
Determines whether a specified event must be audited given the server binding information 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.

Used by client/server applications.

**Format**
```c
#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 *ord,
    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 wanted (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 system console (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. In this case, the outcome is one in a series of connected events, and the application waits to record the real outcome only after the last event.
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 routine is called with `outcome` set to `aud_c_esl_cond_unknown`, it is possible that the routine 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()` routine.

Storage for `ard` will be freed by `dce_aud_commit()`. If `dce_aud_commit()` can not be invoked due to error before the start of the commit, issue `dce_aud_discard()` to free the storage.

`status`

A pointer to the completion status code. On successful completion, the routine returns `aud_s_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

- 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()`.

Usage

The `dce_aud_start_with_server_binding()` routine 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 routine 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 specified, an audit record descriptor will always be created and returned.

The `dce_aud_start_with_server_binding()` routine 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()` routine. Non-RPC applications that do not use the DCE authorization model must use the `dce_aud_start_with_name()` routine.

This routine 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()` routine. The `dce_aud_put_ev_info()` routine can be called multiple times after calling any of the `dce_aud_start_*` routines 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 routine 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 routine and the outcome does not require any action according to filters, then this routine 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 routine. If the final outcome can be `aud_c_esl_cond_success`, then it should be specified in this routine, or use `aud_c_esl_cond_unknown`.

This routine 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()` routine 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**

None.

**Related Information**

**Routines**

- `dce_aud_open`
- `dce_aud_start`
- `dce_aud_start_with_pac`
- `dce_aud_start_with_name`
- `dce_aud_put_ev_info`
- `dce_aud_commit`
**dce_aud_start_with_uuid**

**Purpose**
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.

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.

**Format**

```c
#include <dce/audit.h>

void dce_aud_start_with_uuid(
    unsigned32 event,
    uuid_t server_uuid,
    uuid_t client_uuid,
    uuid_t realm_uuid,
    unsigned_char_t * address,
    unsigned32 options,
    unsigned32 outcome,
    dce_aud_rec_t * ord,
    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 system console (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:
The event outcome (denial, failure, or success) is still unknown.

The event completed successfully.

The event failed due to access denial.

The event failed due to reasons other than access denial.

The event is in an intermediate state, and the outcome is pending. In this case, the outcome is one in a series of connected events, and the application waits 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 routine is called with outcome set to aud_c_esl_cond_unknown, it is possible that the routine 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() routine.

Storage for ard will be freed by dce_aud_commit(). If dce_aud_commit() can not be invoked due to error before the start of the commit, issue dce_aud_discard() to free the storage.

status

A pointer to the completion status code. On successful completion, the routine returns aud_s_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

Status codes passed from dce_aud_start_with_pac().

Usage

The dce_aud_start_with_uuid() routine 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 routine 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 specified, an audit record descriptor will always be created and returned.

The dce_aud_start_with_uuid() routine 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() routine should be used by Non-RPC applications that do not use the DCE authorization model.

This routine 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() routine, 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 routine 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 routine and the outcome does not require any action according to filters, then this routine 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.

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.
**dce_aud_start_with_uuid**

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 routine. If the final outcome can be **aud_c_esl_cond_success**, then it should be specified in this routine, or use **aud_c_esl_cond_unknown**.

This routine 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()** routine 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**

None.

**Related Information**

**Routines**

- dce_aud_open
- dce_aud_start
- dce_aud_start_with_server_binding
- dce_aud_start_with_pac
- dce_aud_start_with_name
- dce_aud_put_ev_info
- dce_aud_commit
DCE Generic Security Service API

This introduction includes general information about the Generic Security Service Application Programming Interface (GSSAPI) defined in Internet RFC 2078, Generic Security Service Application Program Interface, Version 2 and Internet RFC 1509, Generic Security Service API: C-bindings. It also includes an overview of error handling, data types, and calling conventions.

General Information

The Generic Security Service Application Programming Interface (GSSAPI) provides security services to applications using peer-to-peer communications (instead of DCE secure RPC). Using GSSAPI routines, applications can perform the following operations:

- Enable an application to determine another application's user identification
- Enable an application to delegate access rights to another application
- Apply security services, such as confidentiality and integrity, on a per-message basis

A secure connection between two communicating applications is represented by a data structure called a security context. The application that establishes the secure connection is called the context initiator. The context initiator is similar to a DCE RPC client. The application that accepts the secure connection is the context acceptor. The context acceptor is similar to a DCE RPC server. The GSSAPI routines use tokens as input and output values. The communicating applications are responsible for exchanging these tokens using whatever communication channels are appropriate.

There are four stages involved in using the GSSAPI:

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 acquisition and use their default credential. 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 be obtained from an existing login context or can be created using a principal name and key obtained from a key table.

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 establishing 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 by requesting mutual authentication.

   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 call to 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 returned credential to initiate additional security contexts with other applications.

3. The applications exchange protected messages and data.

   The applications can call GSSAPI routines to protect data exchanged in messages. GSSAPI treats application data as arbitrary octet strings. The GSSAPI message security services can provide either integrity and authentication of data origin or confidentiality, integrity, and authentication of data origin. The capability to provide data confidentiality is dependent upon the capabilities of the underlying data encryption support.

4. When the applications have finished communicating, either one may instruct GSSAPI to delete the security context.

There are three types of GSSAPI routines:

1. Standard GSSAPI routines, which are defined in Internet RFC 2078, Generic Security Service Application Program Interface, Version 2 and Internet RFC 1509, Generic Security Service API: C-bindings. These routines have the prefix gss_.

Chapter 6. Security and Related Services  6-71
2. DCE extensions to the GSSAPI. These are additional routines that enable an application to use DCE security services. These routines have the prefix `gssdce_`. DCE extensions are available only in the DCE runtime environment.

3. Kerberos extensions to the GSSAPI. These are additional routines that enable an application to use Kerberos security services. These routines have the prefix `gsskrb_`. Kerberos extensions are available in both the DCE and the Kerberos runtime environments.

The GSSAPI which is provided with DCE supports multiple security mechanisms. A security mechanism defines the contents of tokens which are passed between initiator and acceptor applications. Both the initiator and the acceptor must have at least one security mechanism in common in order to be able to establish a GSSAPI security context.

### GSSAPI Services

GSSAPI provides the following services:

- Message integrity and confidentiality
- Message replay and sequencing
- Quality of protection
- Anonymity

**Message integrity and confidentiality:** GSSAPI provides message security services. Depending upon the underlying security mechanism capabilities, message integrity and message confidentiality services are available. When a security context is established, the GSSAPI routines return two flags to indicate the set of message protection security services which will be available for the context:

- The `GSS_C_INTEG_FLAG` indicates whether message integrity and origin authenticity services are available.
- The `GSS_C_CONF_FLAG` indicates whether message confidentiality services are available. This flag will never be `TRUE` unless the `GSS_C_INTEG_FLAG` is also `TRUE`.

GSSAPI callers desiring message security services should check the values of these flags at context establishment time and must be aware that a returned `FALSE` value means that the invocation of the `gss_get_mic()` and `gss_wrap()` routines will apply no cryptographic protection to user data messages.

The GSSAPI message integrity and data origin authentication services provide assurance to a receiving caller that protection was applied to a message by the caller's peer on the security context, corresponding to the entities named during context establishment. The GSSAPI message confidentiality service provides assurance to a sending caller that the message's content is protected from access by entities other than the context's named peer.

**Message replay and sequencing:** GSSAPI also provides message sequencing and replay detection services. These selectable protection features are distinct from the replay detection and sequencing features supplied by the context establishment operation. The presence or absence of context-level replay or sequencing is a function of the underlying security mechanism layer capabilities and is not selected or omitted as a caller option.

The caller initiating a context provides two flags to specify whether the use of message replay detection and sequencing features is desired on the context being established:

- `GSS_C_REPLAY_FLAG` indicates whether message replay detection services are to be used.
- `GSS_C_SEQUENCE_FLAG` indicates whether message sequencing services are to be used.

The GSSAPI implementation at the initiator system can determine whether these services are supported as a function of the mechanism type. When enabled, these services provide recipients with indicators as a result of GSSAPI processing on incoming messages, identifying whether those messages were detected as duplicate or out-of-sequence. Detection of such events does not prevent a suspect message from being provided to a recipient; the appropriate course of action on a suspect message is a matter of caller policy.

When replay detection is enabled, the possible `major_status` returns for well-formed and correctly signed messages are as follows:

- `GSS_S_COMPLETE` indicates that the message was within the window (of time or sequence space) allowing replay events to be detected, and that the message was not a replay of a previously processed message within that window.
- `GSS_S_DUPLICATE_TOKEN` indicates that the cryptographic check value on the received message was correct, but that the message was recognized as a duplicate of a previously processed message.
• **GSS_S_OLD_TOKEN** indicates that the cryptographic check value on the received message was correct, but that the message is too old to be checked for duplication.

When message sequencing is enabled, the possible **major_status** returns for well-formed and correctly signed messages are as follows:

• **GSS_S_COMPLETE** indicates that the message was within the window (of time or sequence space) allowing replay events to be detected, that the message was not a replay of a previously processed message within that window, and that no predecessor sequenced messages are missing relative to the last received message processed on the context with a correct cryptographic check value.

• **GSS_S_DUPLICATE_TOKEN** indicates that the integrity check value on the received message was correct, but that the message was recognized as a duplicate of a previously processed message.

• **GSS_S_OLD_TOKEN** indicates that the integrity check value on the received message was correct, but that the token is too old to be checked for duplication.

• **GSS_S_UNSEQ_TOKEN** indicates that the cryptographic check value on the received message was correct, but that it is earlier in a sequence stream than a message already processed on the context.

• **GSS_S_GAP_TOKEN** indicates that the cryptographic check value on the received message was correct, but that one or more predecessor sequenced messages have not been successfully processed relative to the last received message on the context with a correct cryptographic check value.

**Quality of protection:** Some mechanisms provide their users with fine granularity control over the means used to provide message protection, allowing callers to trade off security processing overhead dynamically against the protection requirements of particular messages. A message quality-of-protection parameter selects among different QOP options supported by that mechanism. On context establishment for a multi-QOP mechanism, context-level data provides the prerequisite data for a range of protection qualities.

**Anonymity:** In certain situations or environments, an application may wish to authenticate a peer, protect communications, or both using GSSAPI message services without revealing its own identity. In ordinary GSSAPI usage, a context initiator’s identity is made available to the context acceptor as part of the context establishment process. To provide for anonymity support, a **GSS_C_ANON_FLAG** is provided through which context initiators may request that their identity not be provided to the context acceptor. Mechanisms are not required to honor this request, but a caller will be informed through the return flags whether or not the request was honored. Note that authentication as the anonymous principal does not necessarily imply that credentials are not required in order to establish a context.

**Error Handling**

Each GSSAPI routine returns two status values:

• **Major status**
  
  Major status values are generic API errors defined in RFC 2078. They are the same for all implementations of GSSAPI and are not dependent upon the underlying mechanism.

• **Minor status**
  
  Minor status values are mechanism-specific errors which further define the error reported. Minor status values are not portable between implementations of GSSAPI and will vary across mechanisms.

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. The **gss_display_status()** routine may be used to obtain printable text strings for major and minor status values.

**Major Status Values:** GSSAPI routines return GSS status codes as their **OM_uint32** function value. These codes indicate generic API errors and are common across GSSAPI implementations. A GSS status code can indicate a single API error from the routine and a single calling error. Additional status information can also be contained in the GSS status code as supplementary information. The errors are encoded into a 32-bit GSS status code as follows:

```
+-------------------+-------------------+-------------------+
| Calling Error | Routine Error | Supplementary Information |
+-------------------+-------------------+-------------------+
Bit 31  24  23  16  15  0
```
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 application's call of the routine was in error. In addition, the routine can indicate additional information by setting one or more bits in the supplementary information field of the status code.

Following is a list of the GSSAPI calling errors and their meanings:

- **GSS_S_CALL_INACCESSIBLE_READ**: Unable to read an input parameter
- **GSS_S_CALL_INACCESSIBLE_WRITE**: Unable to write an output parameter
- **GSS_S_CALL_BAD_STRUCTURE**: Incorrect parameter structure

Following is a list of the GSSAPI routine errors and their meanings:

- **GSS_S_BAD_MECH**: Mechanism is not supported
- **GSS_S_NAME**: Name is not valid
- **GSS_S_NAMETYPE**: Name type is not valid
- **GSS_S_BAD_BINDINGS**: Channel bindings are not correct
- **GSS_S_BAD_STATUS**: Status value is not valid
- **GSS_S_BAD_SIG**: Token signature is not correct
- **GSS_S_NO_CRED**: No credentials supplied
- **GSS_S_NO_CONTEXT**: No context established
- **GSS_S_DEFECTIVE_TOKEN**: Token is not valid
- **GSS_S_DEFECTIVE_CREDENTIAL**: Credential is not valid
- **GSS_S_CREDENTIALS_EXPIRED**: Credentials have expired
- **GSS_S_CONTEXT_EXPIRED**: Context has expired
- **GSS_S_FAILURE**: Routine failed (check minor status)
- **GSS_S_BAD_QOP**: Bad quality-of-protection value
- **GSS_S_UNAUTHORIZED**: Operation not authorized by local security policy
- **GSS_S_UNAVAILABLE**: Operation or option not available
- **GSS_S_DUPLICATE_ELEMENT**: Credential element already exists
- **GSS_S_NAME_NOT_MN**: Not a mechanism name

Following is a list of the GSSAPI supplementary status bits and their meanings:

- **GSS_S_CONTINUE_NEEDED**: Call routine again to complete request
- **GSS_S_DUPLICATE_TOKEN**: Token is duplicate of earlier token
- **GSS_S_OLD_TOKEN**: Token validity period has expired
- **GSS_S_UNSEQ_TOKEN**: Later token has already been processed
- **GSS_S_GAP_TOKEN**: Skipped predecessor token detected

All **GSS_S_** symbols equate to complete OM_uint32 status codes rather than to bitfield values.

The major status code **GSS_S_FAILURE** indicates that an error was detected for which no major status code is defined. Check the minor status code for details about the error.

The GSSAPI provides three macros for manipulating major status values:

- **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 supplementary information fields.
An additional macro, `GSS_ERROR()`, lets you determine whether the status code indicates a calling or routine error. If the status code indicates a calling or routine error, the macro returns a nonzero value. If no calling or routine error is indicated, the macro returns zero.

Note that an inaccessible read or write error may not be returned. Instead, a signal may be generated as a result of the attempt to access the storage location.

**Minor Status Values:** The GSSAPI routines return a `minor_status` parameter to indicate errors from either the GSSAPI interface layer or the underlying security mechanism layer. The parameter can contain a single error, indicated by an `OM_uint32` value. For the DCE runtime environment, this value is equivalent to the DCE `error_status_t` data type and will contain a DCE status code. For the Kerberos runtime environment, this value is equivalent to the `krb5_error_code` data type and will contain a Kerberos error code. The `gss_display_status()` routine can be used to generate a displayable message describing the minor status code.

**Data Types**

This section provides an overview of the GSSAPI data types and their definitions.

**Integer:** The GSSAPI defines the following integer data type:

```c
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:** 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 `gss_buffer_desc` buffer descriptor, to pass the data between the GSSAPI routines and the application.

The `gss_buffer_t` data type has the following definition:

```c
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. 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 is responsible for allocating the `gss_buffer_desc` object. It can initialize `gss_buffer_desc` objects with the value `GSS_C_EMPTY_BUFFER`. To free the storage allocated by a GSSAPI routine, the application calls the `gss_release_buffer()` routine.

**Object Identifier:** Applications use the `gss_OID` data type to specify a security mechanism and to specify name types.

Select a security mechanism by using the following OIDs:

- For the DCE security mechanism, specify `GSSDCE_C_OID_DCE_KRBV5_DES`. This corresponds to object identifier `{1 3 24 9 8}`. The DCE security mechanism is the default mechanism and will be used if `GSS_C_NO_OID` is specified for the security mechanism type.

- For the Kerberos security mechanism, specify `GSSDCE_C_OID_KRBV5_DES` or `GSSDCE_C_OID_KRBV5_DES_RFC`. These correspond to object identifiers `{1 3 5 1 5 2}` and `{1 2 840 113554 1 2 2}`. You should use `GSSDCE_C_OID_KRBV5_DES_RFC` instead of `GSSDCE_C_OID_KRBV5_DES` unless you need to interoperate with an older level of the Kerberos support.

Select a name type by using the following OIDs:

- For a DCE name, specify `GSSDCE_C_OID_DCENAME`. This corresponds to object identifier `{1 3 24 9 9}`. A DCE name is a character string which is fully-qualified (`/.../cell/principal`), cell-relative (`/.../principal`), or unqualified (`principal`). A DCE name may be used with the DCE security mechanism.

- For a Kerberos name, specify `GSSDCE_C_OID_KRBV5_NAME` or `GSSDCE_C_OID_KRBV5_USER`. These correspond to object identifiers `{1 2 840 113554 1 2 2 1}` and `{1 2 840 113554 1 2 1 1}`. A Kerberos name is a character string which is fully-qualified (`principal@realm`) or unqualified (`principal`). A Kerberos name may be used with the Kerberos mechanism.
For a Kerberos principal, specify GSSDCE_C_OID_KRBV5_PRINCIPAL. This corresponds to object identifier {1 2 840 113554 1 2 2 2}. A Kerberos principal is a krb5_principal structure created by the krb5_parse_name() function. A Kerberos principal may be used with the Kerberos security mechanism.

For a Kerberos service, specify GSSDCE_C_OID_KRBV5_SERVICE. This corresponds to object identifier {1 2 840 113554 1 2 1 4}. A Kerberos service is a character string which is fully-qualified (service@host) or unqualified (service). A Kerberos service may be used with the Kerberos security mechanism.

For a Kerberos UID, specify GSSDCE_C_OID_KRBV5_MACHINE_UID or GSSDCE_C_OID_KRBV5_STRING_UID. These correspond to {1 2 840 113554 1 2 1 2} and {1 2 840 113554 1 2 1 3}. A machine UID is a uid_t as returned by the getuid() function. A string UID is a decimal text string obtained by converting a uid_t to a character string. A Kerberos UID may be used with the Kerberos security mechanism.

You can use the GSSDCE_C_OID_* definitions in both the DCE runtime environment and the Kerberos runtime environment. In addition, the Kerberos runtime environment provides the following equivalent definitions:

- gss_mech_krb5 is equivalent to GSSDCE_C_OID_KRBV5_DES_RFC
- gss_mech_krb5_old is equivalent to GSSDCE_C_OID_KRBV5_DES
- gss_nt_krb5_name is equivalent to GSSDCE_C_OID_KRBV5_NAME
- gss_nt_krb5_principal is equivalent to GSSDCE_C_OID_KRBV5_PRINCIPAL
- gss_nt_service_name is equivalent to GSSDCE_C_OID_KRBV5_SERVICE
- gss_nt_user_name is equivalent to GSSDCE_C_OID_KRBV5_USER
- gss_nt_machine_uid_name is equivalent to GSSDCE_C_OID_KRBV5_MACHINE_UID
- gss_nt_string_uid_name is equivalent to GSSDCE_C_OID_KRBV5_STRING_UID

The gss_OID data type contains tree-structured values defined by ISO and has the following definition:

```c
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 (Basic Encoding Rules) 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 by GSSAPI routines are read-only values. The application should not attempt to release them by calling the gss_release_oid() function.

**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:

- Report the available mechanisms supported by GSSAPI.
- Request specific mechanisms.
- Indicate the mechanisms supported by a GSSAPI credential.
- Report the available name types supported by GSSAPI.

The gss_OID_set data type has the following definition:

```c
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 release the storage associated with gss_OID_set values that are returned by GSSAPI routines.

**Credentials:** Credentials establish, or prove, the identity of an application or other principal. The gss_cred_id_t is an atomic data type that identifies a GSSAPI credential data structure. The data type is opaque to the caller.

**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.
**Tokens:** GSSAPI uses tokens to maintain the synchronization between the communicating applications sharing a security context. The token is a cryptographically protected octet string generated by the underlying security mechanism 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 caller uses the `gss_buffer_t` data type as tokens to GSSAPI routines.

GSSAPI uses two types of tokens. Context-level tokens are used to establish the security context between the communicating applications. Per-message tokens are used to provide integrity and confidentiality services for messages exchanged by the applications.

**Names:** Names identify principals. The GSSAPI authenticates the relationship between a name and the principal claiming the name.

Names are represented in two forms:

- A printable form, for presentation to an application.
- An internal, canonical form that is used by the GSSAPI and is opaque to applications.

The `gss_import_name()` and `gss_display_name()` routines convert names between their printable and internal forms. Each security mechanism has its own name format. The `gss_import_name()` routine will create internal representations of the supplied name for use by each of the supported security mechanisms. Internal names which are created by a specific security mechanism will contain internal representations for just that security mechanism. The `gss_compare_name()` routine can be used to compare two names in their internal format.

**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:

```c
typedef struct gss_channel_binding_struct {
    OM_uint32 initiator_addrtype;
    gss_buffer_desc initiator_address;
    OM_uint32 acceptor_addrtype;
    gss_buffer_desc acceptor_address;
    gss_buffer_desc application_data;
} gss_channel_bindings_desc, *gss_channel_bindings_t;
```

Use the `initiator_addrtype` and `acceptor_addrtype` fields to indicate the type of addresses contained in the `initiator_address` and `acceptor_address` buffers. Following is a list of the address types and their `addrtype` values:

- **GSS_C_AF_UNSPEC** Unspecified
- **GSS_C_AF_LOCAL** Host local address
- **GSS_C_AF_INET** DARPA internet address
- **GSS_C_AF_IMPLINK** ARPAnet IMP
- **GSS_C_AF_PUP** pup protocols (for example, BSP)
- **GSS_C_AF_CHAOS** MIT CHAOS protocol
- **GSS_C_AF_NS** XEROX NS
- **GSS_C_AF_NBS** nbs
- **GSS_C_AF_ECMA** ECMA
- **GSS_C_AF_DATAKIT** datakit protocols
- **GSS_C_AF_CCITT** CCITT protocols (for example, X.25)
- **GSS_C_AF_SNA** IBM SNA
- **GSS_C_AF_DECnet** Digital DECnet
- **GSS_C_AF_DLI** Direct data link interface
- **GSS_C_AF_LAT** LAT
- **GSS_C_AF_HYLINK** NSC Hyperchannel
- **GSS_C_AF_APPLETALK** AppleTalk
- **GSS_C_AF_BSC** BISYNC 2780/3780
GSS_C_AF_DSS          Distributed system services
GSS_C_AF_OSI           OSI TP4
GSS_C_AF_X25           X25
GSS_C_AF_NULLADDR      No address specified

The tags specify address families rather than addressing formats. For address families that contain several alternative address forms, the initiator_address and acceptor_address fields should contain sufficient information to determine which address form is being 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 of the fields in the gss_channel_bindings_desc data structure. 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 generates its own 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 local system. Therefore, portable applications should use either the correct address type and value or specify GSS_C_AF_NULLADDR for the initiator_addrtype 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 DCE GSSAPI does not verify the address or include the plain text binding information in the token.

Optional Parameters: In some of the routine descriptions, optional parameters allow the application to request default behavior by passing a default value for a parameter. The following conventions are used for optional parameters:

- **gss_buffer_t** data types: GSS_C_NO_BUFFER
- Output integer data types: NULL
- OID data types: GSS_C_NO_OID
- OID set data types: GSS_C_NO_OID_SET
- Credential data types: GSS_C_NO_CREDENTIAL
- Context data types: GSS_C_NO_CONTEXT
- Channel binding data types: GSS_C_NO_CHANNEL_BINDINGS

Using GSSAPI Services

Some of the type definitions used by GSSAPI function prototypes have changed between Version 1 and Version 2 of the GSSAPI specifications (Internet RFC 1509). The default definitions will be those defined by Version 2 of the specifications. You can use the Version 1 definitions by defining the GSSAPI_V1_COMPAT compiler variable when compiling your source code.

The following function names have changed between GSSAPI Version 1 and GSSAPI Version 2. The original function names are still supported for compatibility with applications which have been written to the GSSAPI Version 1 specifications.

- The gssapi_sign() routine is now the gssapi_get_mic() routine.
- The gssapi_verify() routine is now the gssapi_verify_mic() routine.
- The gssapi_seal() routine is now the gssapi_wrap() routine.
- The gssapi_unseal() routine is now the gssapi_unwrap() routine.

The z/OS implementation of DCE provides two methods for using GSSAPI services:

- Use the DCE runtime environment
  An application which uses the DCE runtime environment can use the DCE security mechanism as well as the Kerberos security mechanism. All minor status codes returned by GSSAPI routines will be DCE status codes as defined in the various <dce/dcexxmsg.h> include files. In particular, the GSSAPI status codes are defined in <dce/dcegssmsg.h>, the DCE security status codes are defined in <dce/dcesecmsg.h>, and the Kerberos status codes are defined in <dce/dcekrbmsg.h>.
In order to use the DCE runtime environment, the application must include the `<dce/gssapi.h>` include file and must link with the `libdce.a` archive library. Both `/usr/lib/EUVPDLL.x` and `/usr/lib/EUVFDLL.x` must be specified as inputs to the link step. The `-Wl,DLL` option must be specified since the GSSAPI support is packaged as a DLL.

- Use the Kerberos runtime environment

An application which uses the Kerberos runtime environment can use just the Kerberos security mechanism. No DCE services are available in this environment. All minor status codes returned by GSSAPI routines will be Kerberos status codes as defined in the `<krb5/krb5.h>` include file.

In order to use Kerberos services, the application must include the `<krb5/gssapi.h>` include file instead of the `<dce/gssapi.h>` include file and must link with the `libkrb5.a` archive library instead of the `libdce.a` archive library. Only `/usr/lib/EUVFDLL.x` needs to be specified as input to the link step. The `-Wl,DLL` option must be specified since the GSSAPI support is packaged as a DLL.
gss_accept_sec_context

Purpose
Accepts a security context created by the context initiator.

Format
#include <dce/gssapi.h>

OM_uint32 gss_accept_sec_context (  
    OM_uint32 *minor_status,  
    gss_ctx_id_t *context_handle,  
    gss_cred_id_t acceptor_cred_handle,  
    gss_buffer_t input_token,  
    gss_channel_bindings_t input_chan_bindings,  
    gss_name_t *src_name,  
    gss_OID *mech_type,  
    gss_buffer_t output_token,  
    gss_flags_t *ret_flags,  
    OM_uint32 *time_rec,  
    gss_cred_id_t *delegated_cred_handle)

Parameters

Input
acceptor_cred_handle
Specifies the GSSAPI credential for the identity claimed by the context acceptor. The credential
must be either an ACCEPT type credential or a BOTH type credential. If you are using the DCE
security mechanism, you may specify GSS_C_NO_CREDENTIAL for this parameter. In this
case, 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
Specifies the token received from the context initiator.

input_chan_bindings
Specifies the bindings describing the communications channel used between the communicating
applications. The channel bindings specified by the context acceptor must match the bindings
which were specified by the context initiator when the input token was created.

Input/Output
context_handle
Specifies a context handle for the context. The first time that the context acceptor calls the
gss_accept_sec_context() routine, the context handle value must be set to
GSS_C_NO_CONTEXT. For subsequent calls to continue setting up the context, the context
handle must be the value returned by the previous call to the gss_accept_sec_context()
routine.

Output
src_name
Returns the authenticated name of the context initiator. If the authenticated name is not
required, specify NULL for this parameter. The returned name will be an anonymous internal
name if 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.

mech_type
Returns the security mechanism with which the context was established. If the security
mechanism type is not required, specify NULL for this parameter. The gss_OID value returned
for this parameter points to a read-only structure and must not be released by the application.
The returned security mechanism will be one of the following:

- GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism
- GSSDCE_C_OID_KRBV5_DES - Beta Kerberos V5 mechanism
- GSSDCE_C_OID_KRBV5_DES_RFC - Kerberos V5 mechanism

output_token
Returns a token to be returned to the context initiator. If no token is to be passed to the context
initiator, the gss_accept_sec_context() routine will set the output_token length field to zero.
Otherwise, the output_token length and value fields will be set to nonzero values. The
application should release the output token when it is no longer needed by calling the
`gss_release_buffer()` routine.

`ret_flags`  
Returns a bitmask containing independent flags representing services which have been
requested by the initiating application. Specify NULL for this parameter if the flag values are not
required. The following symbolic definitions are provided to test the individual flags and should
be logically ANDed with the value of `ret_flags` to test whether the context supports the service
option.

- **GSS_C_DELEG_FLAG** - Delegated credentials are available if this flag is TRUE.
- **GSS_C_MUTUAL_FLAG** - Mutual authentication is required if this flag is TRUE.
- **GSS_C_REPLAY_FLAG** - Replayed signed or sealed messages will be detected if this flag is
  TRUE.
- **GSS_C_SEQUENCE_FLAG** - Out-of-sequence signed or sealed messages will be detected
  if this flag is TRUE.
- **GSS_C_CONF_FLAG** - Confidentiality services are available if this flag is TRUE.
- **GSS_C_INTEG_FLAG** - Integrity services are available if this flag is TRUE.
- **GSS_C_ANON_FLAG** - Anonymous services are available if this flag is TRUE. The
  `src_name` parameter will return an anonymous internal name.
- **GSS_C_PROT_READY_FLAG** - Protection services, as specified by the
  `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG`, are available if the accompanying major
  status is `GSS_S_COMPLETE` or `GSS_S_CONTINUE_NEEDED`. Otherwise, protection
  services are available only if the accompanying major status is `GSS_S_COMPLETE`.

`time_rec`  
Returns the number of seconds remaining before the context is no longer valid. If the
mechanism does not support credential expiration, the return value will be `GSS_C_INDEFINITE`.
Specify NULL for this parameter if the remaining time is not required.

`delegated_cred_handle`  
Returns the credential handle for delegated credentials received from the context initiator.
Specify NULL for this parameter if the delegated credentials are not required. A credential
handle will be returned only if the `GSS_C_DELEG_FLAG` flag is set in the return flags. The
returned credential can then be used to initiate a new security context by calling the
`gss_init_sec_context()` routine. The returned credential should be released when it is no longer
needed by calling the `gss_release_cred()` routine.

`minor_status`  
Returns a status code from the security mechanism.

**Usage**

The `gss_accept_sec_context()` routine is the second step in establishing a security context between the context initiator and
the context acceptor. In the first step, the context initiator calls the `gss_init_sec_context()` routine, which returns a token for
the security context. The context initiator then passes this security token to the context acceptor. In the second step, the
context acceptor takes the token supplied by the context initiator and calls the `gss_accept_sec_context()` routine to accept the
context.

If the length value in the `output_token` is not zero, the context acceptor must pass the returned token to the context initiator.
The context initiator must then call `gss_init_sec_context()` and specify the context identifier returned by the original call to
`gss_init_sec_context()` as well as the output token which was returned by the context acceptor.

To complete the context establishment, one or more reply tokens may be required from the peer application. If so,
`gss_accept_sec_context()` will return a status flag of `GSS_S_CONTINUE_NEEDED`, in which case it should be called again
when the reply token is received from the peer application, passing the token to `gss_accept_sec_context()` through the
`input_token` parameter.

The availability of confidentiality services is dependent upon the underlying security mechanism and the features which have
been installed on the system. The `GSS_C_CONF_FLAG` will be returned only if confidentiality services are available on both
the local and remote systems. If confidentiality services are available on the remote system but not on the local system, an
error will be returned by the `gss_unwrap()` routine if an encrypted message is received (that is, confidentiality was requested
on the call to the `gss_wrap()` routine on the remote system).
Whenever the **GSS_S_CONTINUE_NEEDED** status flag is set, the context is not fully established and the following restrictions apply to the output parameters:

- The value returned by the *time_rec* parameter is undefined. Unless the accompanying *ret_flags* parameter contains the bit **GSS_C_PROT_READY_FLAG**, indicating that per-message services may be applied in advance of a successful completion status, the value returned by the *mech_type* parameter may be undefined until the routine returns a major status of **GSS_S_COMPLETE**.
- The values of the **GSS_C_DELEG_FLAG**, **GSS_C_MUTUAL_FLAG**, **GSS_C_REPLAY_FLAG**, **GSS_C_SEQUENCE_FLAG**, **GSS_C_CONF_FLAG**, **GSS_C_INTEG_FLAG**, and **GSS_C_ANON_FLAG** bits returned through the *ret_flags* parameter will contain the values that the implementation expects would be valid if context establishment were to succeed.
- The value of the **GSS_C_PROT_READY_FLAG** bit returned through the *ret_flag* parameter will indicate the actual state at the time **gss_accept_sec_context**() returns, whether or not the context is fully established.

**DCE mechanism:** The **gss_accept_sec_context**() routine needs a key to decrypt the token provided by the context initiator. The token contains the unencrypted principal name of the context acceptor. This name identifies the key that the context initiator used to encrypt the token. The **gss_accept_sec_context**() routine matches the principal name with the key in the following way:

- If a credential is specified on the call to the **gss_accept_sec_context**() routine, the principal name in the credential must match the principal name in the token. GSSAPI looks in the key table registered for the principal in order to get the key. If the principal has not been registered by calling the **gssdce_register_acceptor_identity**() routine, GSSAPI looks in the default key table in order to get the key.
- If **GSS_C_NO_CREDENTIAL** is specified on the call to the **gss_accept_sec_context**() routine and the principal name in the token has been registered, GSSAPI looks in the key table registered for the principal in order to get the key.
- If **GSS_C_NO_CREDENTIAL** is specified on the call to the **gss_accept_sec_context**() routine and the principal name in the token has not been registered, GSSAPI returns **GSS_S_FAILURE** because it does not know how to find the key.

Context expiration is not supported and the expiration time will always be returned as **GSS_C_INDEFINITE**.

When delegation is used, a new DCE login context is created and will be associated with the GSSAPI credential returned for the *delegated_cred_handle* parameter. This GSSAPI credential can then be used to initiate new security contexts on behalf of the original context initiator. In addition, the application can access the DCE login context directly by calling the **gssdce_cred_to_login_context**() routine to obtain the handle for the login context.

**Kerberos mechanism:** The **gss_accept_sec_context**() routine needs a key to decrypt the token provided by the context initiator. The token contains the unencrypted principal name of the context acceptor. This name identifies the key that the context initiator used to encrypt the token. The default key table is used to obtain the key for the indicated principal. The **KRB5_KTNAME** environment variable can be set to use a different key table.

The context expiration time will be obtained from the service ticket which was obtained by the context initiator as part of the **gss_init_sec_context**() processing.

When delegation is used, the forwarded Kerberos credentials are stored in a new Kerberos credentials cache which will be associated with the GSSAPI credential returned for the *delegated_cred_handle* parameter. This GSSAPI credential can then be used to initiate new security contexts on behalf of the original context initiator. In addition, the application can access the forwarded Kerberos credentials directly by calling the **gss_krb5_get_ccache**() routine to obtain the handle for the credentials cache.

**Status Codes**

This routine returns the following status codes:

- **GSS_S_COMPLETE**
  - The routine completed successfully.

- **GSS_S_FAILURE**
  - The routine failed for reasons which are not defined at the GSS level. The *minor_status* return parameter contains a mechanism-dependent error code describing the reason for the failure.

- **GSS_S_BAD_BINDINGS**
  - The *input_token* parameter contains different channel bindings from those specified with the *input_chan_bindings* parameter.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_CONTINUE_NEEDED</td>
<td>Control information in the returned output token must be sent to the initiator and a response must be received and passed as the <code>input_token</code> argument to a continuation call to the <code>gss_accept_sec_context()</code> routine.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_TOKEN</td>
<td>Consistency checks performed on the input token failed.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_CREDENTIAL</td>
<td>Consistency checks performed on the credential structure referenced by the <code>verifier_cred_handle</code> parameter failed.</td>
</tr>
<tr>
<td>GSS_S_BAD_SIG</td>
<td>The received input token contains an incorrect signature.</td>
</tr>
<tr>
<td>GSS_S_DUPLICATE_TOKEN</td>
<td>The token is a duplicate of a token which has already been processed. This is a fatal error during context establishment.</td>
</tr>
<tr>
<td>GSS_S_OLD_TOKEN</td>
<td>The token is too old to be checked for duplication against previous tokens. This is a fatal error during context establishment.</td>
</tr>
<tr>
<td>GSS_S_NO_CRED</td>
<td>No credentials are available or the credentials are valid for context initiation use only.</td>
</tr>
<tr>
<td>GSS_S_CREDENTIALS_EXPIRED</td>
<td>Credentials are no longer valid.</td>
</tr>
<tr>
<td>GSS_S_NO_CONTEXT</td>
<td>The context identifier provided by the caller does not refer to a valid security context.</td>
</tr>
<tr>
<td>GSS_S_BAD_MECH</td>
<td>The security mechanism used by the context initiator is not available on the acceptor system.</td>
</tr>
</tbody>
</table>
gss_acquire_cred

Purpose
Allows an application to acquire a GSSAPI credential.

Format
#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,
    gss_cred_usage_t cred_usage,
    gss_cred_id_t *output_cred_handle,
    gss_OID_set *actual_mechs,
    OM_uint32 *time_rec)

Parameters

Input
desired_name Specifies the principal name to be used for the credential. Specify GSS_C_NO_NAME for this parameter to use the name obtained from the default login context.

time_req Specifies the number of seconds that the credential remains valid. Specify GSS_C_INDEFINITE to request the maximum credential lifetime. Specify zero for the default lifetime of two hours.

desired_mechs Specifies the desired security mechanisms for use with the credential. Mechanisms which are not available on the local system will be ignored. The actual mechanisms which can be used with the credential will be returned in the actual_mechs parameter. Specify GSS_C_NO_OID_SET for this parameter to use the default mechanism.

GSSDCE_C_OID_DCE_KRBV5_DES is the default for the DCE runtime environment and GSSDCE_C_OID_KRBV5_DES RFC is the default for the Kerberos runtime environment.

The following security mechanisms are supported:

- GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism (available only in the DCE environment)
- GSSDCE_C_OID_KRBV5_DES - Beta Kerberos V5 mechanism (available in both the DCE and Kerberos environments)
- GSSDCE_C_OID_KRBV5_DES/rfc - Kerberos V5 mechanism (available in both the DCE and Kerberos environments)

cred_usage Specifies the desired credential usage as follows:

- GSS_C_INITIATE if the credential can be used only to initiate security contexts.
- GSS_C_ACCEPT if the credential can be used only to accept security contexts.
- GSS_C_BOTH if the credential can be used to both initiate and accept security contexts.

Output

output_cred_handle Returns the handle for the GSSAPI credential.

actual_mechs Returns the set of mechanism identifiers for which the credential is valid. If the actual mechanisms are not required, specify NULL for this parameter. The gss_OID_set returned for this parameter should be released by calling the gss_release_oid_set() routine when it is no longer needed.

time_rec Returns the number of seconds for which the credential will remain valid. If the time remaining is not required, specify NULL for this parameter.

minor_status Returns a status code from the security mechanism.
Usage
The **gss_acquire_cred()** routine allows an application to obtain a GSSAPI credential. The application can then use the credential with the **gss_init_sec_context()** and **gss_accept_sec_context()** routines.

For the DCE security mechanism, the current login context is obtained by calling the **sec_login_get_current_context()** routine. For the Kerberos security mechanism, the current login context is obtained by calling the **krb5_cc_default()** routine. The current login context must contain a valid ticket-granting ticket if a **GSS_C_INITIATE** credential is to be acquired.

If **GSS_C_INITIATE** is specified for the credential usage, the application must have a valid login context. The **gss_acquire_cred()** routine will use this login context to create the GSSAPI credential. The principal specified by the **desired_name** parameter must match the principal obtained from the login context or must be specified as **GSS_C_NO_NAME**. In the case of the DCE security mechanism, the **desired_name** parameter must be specified as **GSS_C_NO_NAME**.

If **GSS_C_ACCEPT** or **GSS_C_BOTH** is specified for the credential usage, the principal specified by the **desired_name** parameter must be defined in a key table. The **gssdce_register_acceptor_identity()** routine can be used to register a principal and key table for use with the DCE security mechanism (the key retrieval function must be specified as **NULL**, indicating that a key table will be used to obtain the principal's key). The **KRB5_KTNAME** environment variable can be used to set the key table used by the Kerberos security mechanism.

If **GSS_C_BOTH** is specified for the credential usage and DCE is one of the security mechanisms, a new DCE login context will be created for use when initiating a security context. This login context will be deleted when the GSSAPI credential is deleted unless the application calls the **gssdce_set_cred_context_ownership()** routine to change the ownership of the login context.

Status Codes
This routine returns the following status codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GSS_S_COMPLETE</strong></td>
<td>The routine completed successfully.</td>
</tr>
<tr>
<td><strong>GSS_S_FAILURE</strong></td>
<td>The routine failed for reasons which are not define at the GSS level. The <strong>minor_status</strong> return parameter contains a mechanism-dependent error code describing the reason for the failure.</td>
</tr>
<tr>
<td><strong>GSS_S_BAD_MECH</strong></td>
<td>None of the requested mechanisms are supported by the local system.</td>
</tr>
<tr>
<td><strong>GSS_S_BAD_NAMETYPE</strong></td>
<td>The name specified for the <strong>desired_name</strong> parameter is not supported by the applicable underlying GSSAPI mechanisms.</td>
</tr>
<tr>
<td><strong>GSS_S_BAD_NAME</strong></td>
<td>The name specified for the <strong>desired_name</strong> parameter is not valid.</td>
</tr>
<tr>
<td><strong>GSS_S_NO_CRED</strong></td>
<td>Default credentials are not available.</td>
</tr>
</tbody>
</table>
gss_add_cred

Purpose
Adds a credential element to an existing GSSAPI credential.

Format
#include <dce/gssapi.h>

OM_uint32 gss_add_cred (
    OM_uint32 *minor_status,
    gss_cred_id_t input_cred_handle,
    gss_name_t desired_name,
    gss_OID mech_type,
    gss_cred_usage_t cred_usage,
    OM_uint32 init_time_req,
    OM_uint32 accept_time_req,
    gss_cred_id_t *output_cred_handle,
    gss_OID_set *actual_mechs,
    OM_uint32 *init_time_rec,
    OM_uint32 *accept_time_rec)

Parameters

Input
- **input_cred_handle** Specifies the GSSAPI credential which is to be modified. Specify **GSS_C_NO_CREDENTIAL** to modify the default GSSAPI credential.
- **desired_name** Specifies the principal name to be used for the credential.
- **mech_type** Specifies the mechanism element to be added to the credential. The credential must not already contain an element for this mechanism. The supported security mechanisms are as follows:
  - **GSSDCE_C_OID_DCE_KRBV5_DCE** - DCE mechanism
  - **GSSDCE_C_OID_KRBV5_DCE** - Beta Kerberos V5 mechanism
  - **GSSDCE_C_OID_KRBV5_DCE_RFC** - Kerberos V5 mechanism
- **cred_usage** Specifies the desired credential usage as follows:
  - **GSS_C_INITIATE** - The credential can be used only to initiate security contexts
  - **GSS_C_ACCEPT** - The credential can be used only to accept security contexts
  - **GSS_C_BOTH** - The credential can be used to both initiate and accept security contexts
- **init_time_req** Specifies the number of seconds the credential will remain valid for initiating contexts. The DCE implementation of GSSAPI does not support separate initiate and accept expiration times. The actual expiration time will be the smaller of the initiate and accept times. Specify zero to request the default lifetime of two hours. Specify **GSS_C_INDEFINITE** to request the maximum lifetime.
- **accept_time_req** Specifies the number of seconds the credential will remain valid for accepting contexts. The DCE implementation of GSSAPI does not support separate initiate and accept expiration times. The actual expiration time will be the smaller of the initiate and accept times. Specify zero to request the default lifetime of two hours. Specify **GSS_C_INDEFINITE** to request the maximum lifetime.

Output
- **output_cred_handle** Returns the credential handle for the updated credential. If **NULL** is specified for this parameter, the new credential element will be added to the input credential. Otherwise, a new credential will be created from the input credential and will contain all of the credential elements of the input credential plus the new credential element. **NULL** may not be specified for this parameter if **GSS_C_NO_CREDENTIAL** is specified for the input credential.
actual_mechs: Returns the total set of mechanisms supported by the GSSAPI credential. Specify NULL for this parameter if the actual mechanisms are not required. The gss_OID_set returned for this parameter should be released by calling the gss_release_oid_set() routine when it is no longer needed.

init_time_rec: Returns the initiate expiration time in seconds. Specify NULL for this parameter if the initiate time is not required.

accept_time_rec: Returns the accept expiration time in seconds. Specify NULL for this parameter if the accept time is not required.

minor_status: Returns a status code from the security mechanism.

Usage

The gss_add_cred() routine adds a new mechanism element to a GSSAPI credential. The credential must not already contain an element for the mechanism. A GSSAPI credential must contain an element for each mechanism that will be used for contexts which are initiated or accepted using the credential.

The gss_add_cred() routine performs the same function as the gss_acquire_cred() routine for a single mechanism.

Status Codes

This routine returns the following status codes:

GSS_S_COMPLETE: The routine completed successfully.
GSS_S_FAILURE: The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_DUPLICATE_ELEMENT: The credential already contains an element for the specified mechanism.
GSS_S_BAD_MECH: The specified mechanism is not supported.
GSS_S_BAD_NAMETYPE: The supplied name does not contain an internal representation for the requested mechanism.
GSS_S_BAD_NAME: The supplied name is not valid.
GSS_S_NO_CRED: The referenced credential does not exist.
gss_add_oid_set_member

Purpose

Adds an OID to an OID set.

Note: The gss_add_oid_set_member() routine replaces the gssdce_add_oid_set_member() routine. While the gssdce_add_oid_set_member() routine is still available for compatibility with existing applications, new applications should use the gss_add_oid_set_member() routine.

Format

```c
#include <dce/gssapi.h>

OM_uint32 gss_add_oid_set_member (  
  OM_uint32 *minor_status,  
  gss_OID input_oid,  
  gss_OID_set *oid_set)
```

Parameters

Input

input_oid Specifies the OID you want to add to the OID set.

Input/Output

oid_set Specifies the OID set. The gss_OID array referenced by the elements field of the gss_OID_set will be reallocated to hold the new OID. The application should call the gss_release_oid_set() routine to release the OID set when it is no longer needed.

Output

minor_status Returns a status code from the security mechanism.

Usage

The gss_add_oid_set_member() routine adds a new OID to an existing OID set. You can create an empty OID set by calling the gss_create_empty_oid_set() routine. The gss_add_oid_set_member() routine makes a copy of the input OID, so any future changes to the input OID will have no effect on the copy in the OID set.

Status Codes

This routine returns the following status codes:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_COMPLETE</td>
<td>The routine completed successfully.</td>
</tr>
<tr>
<td>GSS_S_FAILURE</td>
<td>The routine failed for reasons which are not define at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.</td>
</tr>
</tbody>
</table>
**gss_canonicalize_name**

**Purpose**
Reduces a GSSAPI internal name to a mechanism name.

**Format**
```c
#include <dce/gssapi.h>

OM_uint32 gss_canonicalize_name (  
    OM_uint32 *minor_status,  
    gss_name_t input_name,  
    gss_OID mech_type,  
    gss_name_t *output_name)
```

**Parameters**

**Input**
- `input_name` Specifies the name to be processed. An error will be returned if GSS_C_NO_NAME is specified for this parameter.
- `mech_type` Specifies the security mechanism to be used as follows:
  - GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism
  - GSSDCE_C_OID_KRBV5_DES - Beta Kerberos V5 mechanism
  - GSSDCE_C_OID_KRBV5_DES_RFC - Kerberos V5 mechanisms

**Output**
- `output_name` Returns the mechanism name. The `gss_name_t` returned by this parameter should be released by calling the gss_release_name() function when it is no longer needed.
- `minor_status` Returns a status code from the security mechanism.

**Usage**
The gss_canonicalize_name() routine takes a GSSAPI internal name which contains multiple internal representations and returns a new GSSAPI internal name with a single name representation which corresponds to the specified security mechanism. A name which represents a single security mechanism is called a mechanism name.

**Status Codes**
This routine returns the following status codes:
- **GSS_S_COMPLETE** The routine completed successfully.
- **GSS_S_FAILURE** The routine failed for reasons which are not define at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_BAD_NAMETYPE** The input name does not contain an element for the requested mechanism.
- **GSS_S_BAD_NAME** The input name is not valid.
- **GSS_S_BAD_MECH** The requested mechanism is not supported.
gss_compare_name

Purpose
Allows an application to compare two internal names to determine if they refer to the same object.

Format
#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 1 if the names refer to the same object and 0 otherwise.
minor_status
Returns a status code from the security mechanism.

Usage
The gss_compare_name() routine lets an application compare two internal names to determine whether they refer to the same object. The two names must have an internal representation format in common in order to be comparable. The names will be considered not equal if either name denotes an anonymous principal.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE
The routine completed successfully.

GSS_S_FAILURE
The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_BAD_NAMETYPE
The two name types are incomparable. The names must have an internal representation in common in order to be comparable.

GSS_S_BAD_NAME
One of the input names is not valid.
gss_context_time

Purpose
Returns the number of seconds that the context will remain valid.

Format
```
#include <dce/gssapi.h>

OM_uint32 gss_context_time (
    OM_uint32 *minor_status,
    gss_ctx_id_t context_handle,
    OM_uint32 *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.

minor_status Returns a status code from the security mechanism.

Usage
The gss_context_time() routine checks the specified security context and returns the number of seconds that the context will remain valid. The returned value will be GSS_C_INDEFINITE if the context does not have an expiration time. The DCE security mechanism does not support context expiration and will always return GSS_C_INDEFINITE. The Kerberos security mechanism does support context expiration and will return the time remaining before the underlying service ticket expires.

Status Codes

GSS_S_COMPLETE The routine completed successfully.

GSS_S_FAILURE The routine failed for reasons which are not define at the GSS level. The 
minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_CONTEXT_EXPIRED The referenced context has expired.

GSS_S_CREDENTIALS_EXPIRED The credentials associated with the referenced context have expired.

GSS_S_NO_CONTEXT The referenced context does not exist.
gss_create_empty_oid_set

Purpose
Creates a new, empty OID set.

Note: The gss_create_empty_oid_set() routine replaces the gssdce_create_empty_oid_set() routine. While the gssdce_create_empty_oid_set() routine is still available for compatibility with existing applications, new applications should use the gss_create_empty_oid_set() routine.

Format
#include <dce/gssapi.h>

OM_uint32 gss_create_empty_oid_set (    OM_uint32 *minor_status,    gss_OID_set *oid_set)

Parameters
Output
oid_set Returns the OID set created by this routine. The application should call the gss_release_oid_set() routine when the OID set is no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gss_create_empty_oid_set() routine creates a new, empty OID set. Members can be added to the OID set by calling the gss_add_oid_set_member() routine. The OID set should be released when it is no longer needed by calling the gss_release_oid_set() routine.

Status Codes
This routine returns the following status codes:

GSS_C_COMPLETE The routine completed successfully.
GSS_C_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
gss_delete_sec_context

Purpose
Deletes a security context.

Format
#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)

Parameters

Input/Output
context_handle Specifies the context to be deleted. Upon successful completion, the context_handle value will be set to GSS_C_NO_CONTEXT.

Output
output_token Returns a token to be sent to the partner application. The partner application then passes this token to the gss_process_context_token() routine to delete the other end of the security context. The gss_delete_sec_context() routine will set the output_token length field to zero if no token needs to be sent to the partner application.

GSS_C_NO_BUFFER may be specified for the output_token parameter. In this case, no token will be returned by the gss_delete_sec_context() routine. Both of the communicating applications must call gss_delete_sec_context() in order to delete both ends of the security context.

minor_status Returns a status code from the security mechanism.

Usage
The gss_delete_sec_context() routine deletes one end of 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 must then pass this token to the partner application. The partner application calls the gss_process_context_token() routine to process the token and complete the process of deleting the security context.

This call can be made by either peer in a security context to flush context-specific information. Both communicating applications must call the gss_delete_sec_context() routine if GSS_C_NO_BUFFER is specified for the output_token parameter.

The context_handle may not be used for additional security services once the gss_delete_sec_context() routine has successfully completed.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.

GSS_S_FAILURE The routine failed for reasons which are not define at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_NO_CONTEXT The supplied context handle did not refer to a valid context.
**gss_display_name**

**Purpose**
Provides the textual representation of an opaque internal name.

**Format**

```c
#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 internal name to be converted to a text string.

**Output**
- **output_name_buffer**
  Return buffer for the character string.
- **output_name_type**
  Returns the name type corresponding to the returned character string. The `gss_OID` value returned for this parameter points to read-only storage and must not be released by the application.
- **minor_status**
  Returns a status code from the security mechanism.

**Usage**

The `gss_display_name()` routine provides an application with the text form of an opaque internal name. The syntax of the text representation is determined by the mechanism which was used to convert the name. The DCE mechanism will be used if the internal name contains a DCE representation. Otherwise, the Kerberos mechanism will be used.

DCE names are formatted as `.../cell-name/principal-name`. Kerberos names are formatted as `principal-name@realm-name`. For the DCE implementation of GSSAPI, the realm name and the cell name are the same.

**Status Codes**

This routine returns the following status codes:

- **GSS_S_COMPLETE**
  The routine completed successfully.
- **GSS_S_FAILURE**
  The routine failed for reasons which are not define at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_BAD_NAMETYPE**
  The internal name provided does not have an internal representation for any of the supported mechanisms.
- **GSS_S_BAD_NAME**
  The provided name is not valid.
gss_display_status

Purpose
Provides an application with the textual representation of a GSS or mechanism status code.

Format
```
#include <dce/gssapi.h>

OM_uint32 gss_display_status (
  OM_uint32 *minor_status,
  OM_uint32 status_value,
  int status_type,
  gss_OID mech_type,
  gss_msg_ctx_t *message_context,
  gss_buffer_t status_string)
```

Parameters

Input
- **status_value**: Specifies the status value to be converted. A status value of zero is not valid and will cause the `gss_display_status()` routine to return a major status of GSS_S_BAD_STATUS to the application.
- **status_type**: Specifies the status value type and must be one of the following:
  - GSS_C_GSS_CODE - GSS major status code
  - GSS_C_MECH_CODE - Mechanism minor status code
- **mech_type**: Specifies the security mechanism associated with a minor status code. This parameter is used only when converting a minor status code. The DCE implementation of GSSAPI validates this parameter but does not use it to convert the status code since DCE uses common status code definitions between the DCE and Kerberos mechanisms.

Input/Output
- **message_context**: Indicates whether the status code has multiples messages to be processed. The first time an application calls `gss_display_status()`, the `message_context` parameter must be initialized to zero. The `gss_display_status()` routine will return the first message and set the `message_context` parameter to a nonzero value if there are additional messages available. The application then continues to call the `gss_display_status()` routine to obtain the additional messages until the `message_context` value is zero upon return from the `gss_display_status()` routine.

Output
- **status_string**: Returns the text message for the status value.
- **minor_status**: Returns a status code from the security mechanism.

Usage
The `gss_display_status()` routine provides the application with a textual representation of a status code. The returned message can then be displayed to the user or written to a log file.

The `message_context` parameter indicates which error message should be returned when a status code has multiple messages. The first time an application calls the `gss_display_status()` routine, it must initialize the `message_context` value to zero. The `gss_display_status()` routine will then return the first message for the status code and will set `message_context` to a nonzero value if there are additional messages available. The application can then continue to call `gss_display_status()` until the `message_context` value is zero upon return.
### Status Codes

This routine returns the following status codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GSS_S_COMPLETE</strong></td>
<td>The routine completed successfully.</td>
</tr>
<tr>
<td><strong>GSS_S_FAILURE</strong></td>
<td>The routine failed for reasons which are not defined at the GSS level. The <em>minor_status</em> return parameter contains a mechanism-dependent error code describing the reason for the failure.</td>
</tr>
<tr>
<td><strong>GSS_S_BAD_MECH</strong></td>
<td>The mechanism specified by the <em>mech_type</em> parameter is not supported.</td>
</tr>
<tr>
<td><strong>GSS_S_BAD_STATUS</strong></td>
<td>The value of the <em>status_type</em> parameter is not <strong>GSS_C_GSS_CODE</strong> or <strong>GSS_C_MECH_CODE</strong>, or the value of the <em>status_value</em> parameter is not a valid status code.</td>
</tr>
</tbody>
</table>
**gss_duplicate_name**

**Purpose**
Creates a duplicate of a GSSAPI internal name.

**Format**
```c
#include <dce/gssapi.h>

OM_uint32 gss_duplicate_name (  
    OM_uint32 *minor_status,  
    gss_name_t input_name,  
    gss_name_t *output_name)
```

**Parameters**

**Input**
- `input_name`: Specifies the name to be duplicated. An error will be returned if `GSS_C_NO_NAME` is specified for this parameter.

**Output**
- `output_name`: Returns the new GSSAPI internal name. The `gss_name_t` returned for this parameter should be released by calling the `gss_release_name()` function when it is no longer needed.
- `minor_status`: Returns a status code from the security mechanism.

**Usage**
The `gss_duplicate_name()` routine makes a copy of a GSSAPI internal name.

**Status Codes**
This routine returns the following status codes:

- **GSS_S_COMPLETE**: The routine completed successfully.
- **GSS_S_FAILURE**: The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_BAD_NAME**: The input name is not valid.
- **GSS_S_BAD_NAMETYPE**: The input name type is not supported.
gss_get_mic

Purpose
Generates a cryptographic signature for a message.

Note: The gss_get_mic() routine replaces the gss_sign() routine. While the gss_sign() routine is still available for compatibility with existing applications, new applications should use the gss_get_mic() routine.

Format
#include <dce/gssapi.h>

OM_uint32 gss_get_mic (  
  OM_uint32 *minor_status,  
  gss_ctx_id_t context_handle,  
  gss_qop_t qop_req,  
  gss_buffer_t input_message,  
  gss_buffer_t output_token)

Parameters
Input
context_handle Specifies the context to be associated with the message when it is sent to the partner application.

qop_req Specifies the requested quality of protection for the message. Specify GSS_C_QOP_DEFAULT to use the default quality of protection as defined by the selected security mechanism.

The DCE security mechanism supports three quality-of-protection levels as follows (in decreasing order of speed but increasing protection):

- GSSDCE_C_QOP_MD5 - Truncated MD5 (default)
- GSSDCE_C_QOP_DES_MD5 - DES_MAC of an MD5 hash
- GSSDCE_C_QOP_DES_MAC - Normal DES_MAC algorithm

The Kerberos security mechanism supports three quality-of-protection levels as follows (in decreasing order or speed):

- GSS_KRB5_INTEG_C_QOP_MD5 - Truncated MD5
- GSS_KRB5_INTEG_C_QOP_DES_MD5 - DES_MAC of an MD5 hash (default)
- GSS_KRB5_INTEG_C_QOP_DES_MAC - Normal DES_MAC algorithm

input_message Specifies the message for which a signature is to be generated.

Output
output_token Returns a token containing the message signature. The message and this token is then sent to the partner application, which calls the gss_verify_mic() function to verify the authenticity of the message.

minor_status Returns a status code from the security mechanism.

Usage
The gss_get_mic() routine generates an encrypted signature for a message and returns this signature in a token which can be sent to a partner application. The partner application then calls the gss_verify_mic() routine to validate the signature.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE  The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_NO_CONTEXT The referenced context does not exist.

GSS_S_CONTEXT_EXPIRED The referenced context has expired.

GSS_S_CREDENTIALS_EXPIRED The credentials associated with the referenced context have expired.

GSS_S_BAD_QOP The requested quality-of-protection value is not valid.
gss_import_name

Purpose
Converts a printable name to the GSSAPI internal format.

Format
#include <dce/gssapi.h>

OM_uint32 gss_import_name (
    OM_uint32 *minor_status,
gss_buffer_t input_name_buffer,
gss_OID input_name_type,
gss_name_t *output_name)

Parameters

Input
input_name_buffer Specifies the buffer containing the name to convert.
input_name_type Specifies the object identifier for the type of printable name. The following name types are supported:

- GSS_C_NO_OID - specifies the default name type. For the DCE runtime environment, the default is GSSDCE_C_OID_DCENAME. For the Kerberos runtime environment, the default is GSSDCE_C_OID_KRBV5_NAME.
- GSSDCE_C_OID_DCENAME - specifies a DCE name
- GSSDCE_C_OID_KRBV5_NAME - specifies a Kerberos name
- GSSDCE_C_OID_KRBV5_USER - specifies a Kerberos user name
- GSSDCE_C_OID_KRBV5_SERVICE - specifies a Kerberos service. The krb5_sname_to_principal() routine will be used to convert the service to a Kerberos principal. The domain_realm section in the Kerberos configuration file must contain an entry which matches the host name in order to assign the proper Kerberos realm name.
- GSSDCE_C_OID_KRBV5_PRINCIPAL - specifies a Kerberos principal
- GSSDCE_C_OID_KRBV5_MACHINE_UID - specifies a Kerberos binary UID
- GSSDCE_C_OID_KRBV5_STRING_UID - specifies a Kerberos string UID

Output
output_name Return the name in the GSSAPI internal format. The internal format will contain an internal representation for each of the supported security mechanisms.
minor_status Returns a status code from the security mechanism.

Usage
The gss_import_name() routine converts a printable name to the internal GSSAPI format. The gss_name_t object created by this routine can then be used as input to other GSSAPI routines. The gss_name_t object created by the gss_import_name() routine will contain an internal representation for each of the supported security mechanisms.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_BAD_NAMETYPE</td>
<td>The name type specified by the <code>input_name_type</code> parameter is not valid.</td>
</tr>
<tr>
<td>GSS_S_BAD_NAME</td>
<td>The input name is not formatted properly as defined by the name type specification.</td>
</tr>
</tbody>
</table>
Purpose
Allows an application to determine which security mechanisms are available.

Format
#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 application should release the
gss_OID_set returned for this parameter by calling the gss_release_oid_set() routine.

minor_status
Returns a status code from the security mechanism.

Usage
The gss_indicate_mechs() routine enables an application to determine which security mechanisms are available on the local system.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE
The routine completed successfully.

GSS_S_FAILURE
The routine failed for reasons which are not defined at the GSS level. The
minor_status return parameter contains a mechanism-dependent error code
describing the reason for the failure.
gss_init_sec_context

Purpose
Initiates a security context for use by two communicating applications.

Format
#include <dce/gssapi.h>

OM_uint32 gss_init_sec_context (
    OM_uint32 minor_status,
    gss_cred_id_t cred_handle,
    gss_ctx_id_t *context_handle,
    gss_name_t target_name,
    gss_OID mech_type,
    gss_flags_t req_flags,
    OM_uint32 time_req,
    gss_channel_bindings_t input_chan_bindings,
    gss_buffer_t input_token,
    gss_OID *actual_mech_type,
    gss_buffer_t output_token,
    gss_flags_t *ret_flags,
    OM_uint32 *time_rec)

Parameters
Input

cred_handle Specifies the credential handle of the GSSAPI credential to be used to initiate the security context. The specified credential must be either an INITIATE or BOTH credential. Specify GSS_C_NO_CREDENTIAL to use the default credential obtained from the current login context.

target_name Specifies the name of the context acceptor. This must be a Kerberos service name if delegation is requested for the Kerberos security mechanism. Otherwise, it can be any principal defined in the security registry, subject to registry policy rules.

mech_type Specifies the desired security mechanism as follows:

- GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism
- GSSDCE_C_OID_KRBV5_DES - Beta Kerberos V5 mechanism
- GSSDCE_C_OID_KRBV5_DES_RFC - Kerberos V5 mechanism
- GSS_C_NO_OID - Default mechanism. For the DCE runtime environment, the default mechanism is the DCE mechanism. For the Kerberos runtime environment, the default mechanism is the Kerberos V5 mechanism.

req_flags Specifies a bitmask containing independent flags representing requested GSS services. GSSAPI does not guarantee that a requested service will be available on all systems. The application should check the ret_flags parameter to determine which of the requested services are actually provided for the context. The following symbolic definitions are provided to correspond to each flag. The symbolic names should be logically ORed to form the bitmask value.

- GSS_C_DELEG_FLAG - Request delegated credentials for use by the context acceptor.
- GSS_C_MUTUAL_FLAG - Request mutual authentication to validate the identity of the context acceptor.
- GSS_C_REPLAY_FLAG - Request message replay detection for signed or sealed messages. Specifying this flag implies mutual authentication when the DCE security mechanism is used.
- GSS_C_SEQUENCE_FLAG - Request message sequence checking for signed or sealed messages. Specifying this flag implies mutual authentication when the DCE security mechanism is used.
**gss_init_sec_context**

- **GSS_C_ANON_FLAG** - Request initiator anonymity. This flag is ignored in the current GSSAPI implementation since the DCE and Kerberos mechanisms do not support initiator anonymity.

**time_req**

Specifies the desired number of seconds that the security context will remain valid. Specify zero for the default lifetime of two hours. Specify **GSS_C_INDEFINITE** to request the maximum lifetime.

**input_chan_bindings**

Specifies the bindings describing the communications channel which will be used between the communicating applications. The channel bindings information will be placed into the output token which is generated by the `gss_init_sec_context()` routine and will be validated by the `gss_accept_sec_context()` routine.

**input_token**

Specifies the token received from the context acceptor. **GSS_C_NO_BUFFER** should be specified if this is the first call to the `gss_init_sec_context()` routine.

**Input/Output**

**context_handle**

Specifies the context handle for the context. The first time that the context initiator calls the `gss_init_sec_context()` routine, the context handle must be set to **GSS_C_NO_CONTEXT**. For subsequent calls to continue setting up the context, the context handle must be the value returned by the previous call to the `gss_init_sec_context()` routine.

**Output**

**actual_mech_type**

Returns the security mechanism which will be used with the context. The `gss_OID` value returned for this parameter points to read-only storage and must not be released by the application.

**output_token**

Returns a token to be sent to the context acceptor. If no token is to be sent to the context acceptor, the `gss_init_sec_context()` routine will set the `output_token` length field to zero. Otherwise, the `output_token` length and value fields will be set. The application should release the output token when it is no longer needed by calling the `gss_release_buffer()` routine.

**ret_flags**

Returns a bitmask containing independent flags indicating which GSS services will be available for the context. The following symbolic definitions are provided to test the individual flags and should be logically ANDed with the value of `ret_flags` to test whether the context supports the service options.

- **GSS_C_DELEG_FLAG** - Delegated credentials will be available to the context acceptor.
- **GSS_C_MUTUAL_FLAG** - Mutual authentication will be performed. The `gss_accept_sec_context()` routine will generate an output token which the context acceptor must return to the context initiator to complete the security context setup.
- **GSS_C_REPLAY_FLAG** - Message replay detection will be performed.
- **GSS_C_SEQUENCE_FLAG** - Message sequence checking will be performed.
- **GSS_C_CONF_FLAG** - Message confidentiality services are available.
- **GSS_C_INTEG_FLAG** - Message integrity services are available.
- **GSS_C_ANON_FLAG** - The initiator identity will not be provided to the context acceptor.
- **GSS_C_PROT_READY_FLAG** - If this flag is set, protection services, as specified by the states of the `GSS_C_CONF_FLAG` and `GSS_C_INTEG_FLAG`, are available for use if the accompanying major status return value is **GSS_S_COMPLETE** or **GSS_S_CONTINUE_NEEDED**. Otherwise, protection services are available for use only if the accompanying major status return value is **GSS_S_COMPLETE**.

**time_rec**

Return the number of seconds for which the context will remain valid. If the mechanism does not support context expiration, the return value will be **GSS_C_INDEFINITE**. Specify **NULL** for this parameter if the context expiration time is not required.

**minor_status**

Returns a status code from the security mechanism.
Usage

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 the default credential by specifying `GSS_C_NO_CREDENTIAL` for the `cred_handle` parameter.

The first time the application calls the `gss_init_sec_context()` routine, the `input_token` parameter should either be specified as `GSS_C_NO_BUFFER` or the buffer length field should be set to zero. If no token needs to be sent to the context acceptor, the `gss_init_sec_context()` routine will set the `output_token` length field to zero.

To finish establishing the context, the calling application can require one or more tokens from the context acceptor. If the application requires reply tokens, the `gss_init_sec_context()` routine returns `GSS_S_CONTINUE_NEEDED` in the supplementary information portion of the major status value. The application must call the `gss_init_sec_context()` routine again when it receives the reply token from the context acceptor and pass the token through the `input_token` parameter. When calling the `gss_init_sec_context()` routine to continue processing a context, the same request values must be used as for the initial call.

The availability of confidentiality services is dependent upon the underlying security mechanism and the features which have been installed on the system. The `GSS_C_CONF_FLAG` will be returned only if confidentiality services are available on local system. However, this does not guarantee that confidentiality services are also available on the remote system. If confidentiality services are available on the local system but not on the remote system, an error will be returned by the `gss_unwrap()` routine on the remote system if an encrypted message is received (that is, confidentiality was requested on the call to the `gss_wrap()` routine on the local system).

Whenever the routine returns a major status that includes the value `GSS_S_CONTINUE_NEEDED`, the context is not fully established and the following restrictions apply to the output parameters:

- The value returned by the `time_rec` parameter is undefined.
- Unless the accompanying `ret_flags` parameter contains the bit `GSS_C_PROT_READY_FLAG`, indicating that per-message services may be applied in advance of a successful completion status, the value returned by the `actual_mech_type` parameter is undefined until the routine returns a major status value of `GSS_S_COMPLETE`.
- The values of the `GSS_C_DELEG_FLAG`, `GSS_C_MUTUAL_FLAG`, `GSSC_REPLAY_FLAG`, `GSS_C_SEQUENCE_FLAG`, `GSS_C_CONF_FLAG`, `GSS_C_INTEG_FLAG`, and `GSS_C_ANON_FLAG` bits returned by the `ret_flags` parameter will contain the values that would be returned if the context establishment were to succeed. In particular, if the application has requested a service such as delegation or anonymous authentication through the `req_flags` parameter, and such a service is unavailable from the underlying mechanism, `gss_init_sec_context()` will generate a token that will not provide the service and will indicate through the `ret_flags` parameter that the service will not be supported. The application may choose to abort the context establishment by calling `gss_delete_sec_context()` or it may choose to transmit the token and continue context establishment.
- The value of the `GSS_C_PROT_READY_FLAG` bit returned by the `ret_flags` parameter will indicate the actual state at the time `gss_init_sec_context()` returns, whether or not the context is fully established.

**DCE mechanism:** If delegation is requested but the login context associated with the GSSAPI credential has not enabled delegation, the `gss_init_sec_context()` routine will call the `sec_login_become_initiator()` routine to create a temporary login context and will specify `sec_id_deleg_type_impersonation` for the delegation type. Similarly, if delegation is not requested but the login context associated with the GSSAPI credential has enabled delegation, the `gss_init_sec_context()` routine will call the `sec_login_disable_delegation()` routine to create a temporary login context with delegation disabled. The temporary login context will be used to establish the security context and will then be purged before control returns from the `gss_init_sec_context()` routine.

Context expiration is not supported and the context lifetime will always be returned as `GSS_C_INDEFINITE`.

**Kerberos mechanism:** In order to use delegation, the target principal name must be a service name. A service name is created by calling the `gss_import_name()` routine with the name type specified as `GSSDCE_C_OID_KRBV5_SERVICE` (object identifier 1.2.840.113554.1.2.1.4). The service name is specified as `name@host` and results in a Kerberos principal of `name@host` or `host@host-realm`. The local host name will be used if no host is specified. If a host name alias is specified, the primary host name returned by the domain name service will be used when constructing the principal name.

The requested context lifetime is used to specify the end time when obtaining a Kerberos service ticket to the target application. The actual context lifetime will then be set to the lifetime of the ticket, which may be less than the requested lifetime as determined by the registry policy.
gss_init_sec_context

Status Codes

This routine returns the following status codes:

GSS_S_COMPLETE  The routine completed successfully.
GSS_S_FAILURE   The routine failed for reasons which are not defined at the GSS level. The
                minor_status return parameter contains a mechanism-dependent error code
                describing the reason for the failure.
GSS_S_BAD_MECH  The request security mechanism is not supported.
GSS_S_BAD_NAME  The target_name parameter is not valid.
GSS_S_CONTINUE_NEEDED  To complete the context, the gss_init_sec_context() routine must be called again
                       with a token created by the gss_accept_sec_context() routine.
GSS_S_DEFECTIVE_CREDENTIAL  Consistency checks performed on the credential failed.
GSS_S_DEFECTIVE_TOKEN  Consistency checks performed on the input token failed.
GSS_S_NO_CONTEXT  The supplied context handle does not refer to a valid context.
GSS_S_NO_CRED  The supplied credential handle does not refer to a valid credential, the supplied
                credential is not valid for context initiation, or there are no default credentials
                available.
GSS_S_CREDENTIALS_EXPIRED  The supplied credentials are no longer valid.
GSS_S_BAD_BINDINGS  The channel bindings are not valid.
GSS_S_BAD_SIG  The input token contains an incorrect integrity check value.
GSS_S_DUPLICATE_TOKEN  The token is a duplicate of a token which has already been processed.
GSS_S_OLD_TOKEN  The token is too old to be checked for duplication against tokens which have
                 already been processed.
Purpose
Returns information about a security context.

Format
#include <dce/gssapi.h>

OM_uint32 gss_inquire_context (  
OM_uint32 *minor_status,  
gss_ctx_id_t context_handle,  
gss_name_t *src_name,  
gss_name_t *tgt_name,  
OM_uint32 *lifetime,  
gss_OID *mech_type,  
gss_flags_t *ret_flags,  
int *local,  
int *open)

Parameters
Input  
context_handle  Specifies the handle for the security credential.

Output  
src_name  Returns the principal name associated with the context initiator. Specify NULL for this parameter if the principal name is not required.

tgt_name  Returns the principal name associated with the context acceptor. Specify NULL for this parameter if the principal name is not required.

lifetime  Returns the number of seconds for which the context will remain valid. Specify NULL for this parameter if the context lifetime is not required. The returned value will be GSS_C_INDEFINITE if the security mechanism does not support context expiration.

mech_type  Returns the mechanism used to create the security context. The gss_OID value returned for this parameter points to read-only storage and must not be released by the application. Specify NULL for this parameter if the mechanism type is not required.

ret_flags  Returns a bitmask containing independent flags indicating which GSS services are available for the context. Specify NULL for this parameter if the available service flags are not required. The following symbolic definitions are provided to test the individual flags and should be logically ANDed with the value of ret_flags to test whether the context supports the service options.

- GSS_C_DELEG_FLAG - Delegated credentials will be available to the context acceptor.
- GSS_C_MUTUAL_FLAG - Mutual authentication will be performed. The gss_accept_sec_context() routine will generate an output token which the context acceptor must return to the context initiator to complete the security context setup.
- GSS_C_REPLAY_FLAG - Message replay detection will be performed.
- GSS_C_SEQUENCE_FLAG - Message sequence checking will be performed.
- GSS_C_CONF_FLAG - Message confidentiality services are available.
- GSS_C_INTEG_FLAG - Message integrity services are available.
- GSS_C_ANON_FLAG - The initiator identity will not be provided to the context acceptor.
- GSS_C_PROT_READY_FLAG - If set, protection services, as specified by the states of the GSS_C_CONF_FLAG and GSS_C_INTEG_FLAG bits, are available for use even if the context is not fully established. Otherwise, protection services are available only if the value returned by the open parameter is TRUE.
gss_inquire_context

local          Returns **TRUE** if the context was initiated locally and **FALSE** otherwise. Specify **NULL** for this parameter if the local indication is not required.

open          Returns **TRUE** if context establishment has been completed and **FALSE** otherwise. Specify **NULL** for this parameter if the open indication is not required.

minor_status          Returns a status code from the security mechanism.

Usage
The `gss_inquire_context()` routine provides information about a security context to the calling application.

Status Codes
This routine returns the following status codes:

- **GSS_S_COMPLETE**          The routine completed successfully.
- **GSS_S_FAILURE**          The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_NO_CONTEXT**         The referenced context does not exist.
- **GSS_S_CONTEXT_EXPIRED**     The referenced context has expired.
gss_inquire_cred

Purpose
Returns information about a GSSAPI credential.

Format
#define 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,
    gss_cred_usage_t *cred_usage,
    gss_OID_set *mechanisms)

Parameters
Input
cred_handle Specifies the handle for the GSSAPI credential. Specify GSS_C_NO_CREDENTIAL to get information about the default credential for the default security mechanism.

Output
name Returns the principal name associated with the credential. Specify NULL for this parameter if the principal name is not required.
lifetime Returns the number of seconds for which the credential will remain valid. The return value will be set to zero if the credential has expired. Specify NULL for this parameter if the credential lifetime is not required.
cred_usage Returns one of the following values describing how the application can use the credential. Specify NULL for this parameter if the credential usage is not required.
    • GSS_C_INITIATE - the application may initiate a security context.
    • GSS_C_ACCEPT - the application may accept a security context.
    • GSS_C_BOTH - the application may both initiate and accept security contexts.
mechanisms Returns the set of security mechanisms supported by the credential. Specify NULL for this parameter if the mechanism set is not required. The gss_OID_set returned for this parameter should be released when it is no longer needed by calling the gss_release_oid_set() routine.
minor_status Returns a status code from the security mechanism.

Usage
The gss_inquire_cred() routine provides information about a GSSAPI credential to the calling application. If GSS_C_NO_CREDENTIAL is specified for the cred_handle parameter, the default security mechanism will be used to process the request.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_DEFECTIVE_CREDENTIAL The credentials are not valid.
gss_inquire_cred

GSS_S_NO_CRED  The cred_handle does not refer to a valid credential or there are no default credentials available.

GSS_S_CREDENTIALS_EXPIRED  The credentials have expired. Credential information will still be returned for an expired credential but the lifetime value will be returned as zero.
**gss_inquire_cred_by_mech**

**Purpose**

Returns information about a GSSAPI credential for a single security mechanism.

**Format**

```c
#include <dce/gssapi.h>

OM_uint32 gss_inquire_cred_by_mech (  
    OM_uint32 *minor_status,  
    gss_cred_id_t cred_handle,  
    gss_OID mech_type,  
    gss_name_t *name,  
    OM_uint32 *init_lifetime,  
    OM_uint32 *accept_lifetime,  
    gss_cred_usage_t *cred_usage)
```

**Parameters**

**Input**

- **cred_handle**
  - Specifies the handle for the GSSAPI credential. Specify `GSS_C_NO_CREDENTIAL` to get information about the default credential for the specified security mechanism.

- **mech_type**
  - Specifies the mechanism to be used to obtain the return information as follows:
    - `GSSDCE_C_OID_DCE_KRBV5_DES` - DCE mechanism
    - `GSSDCE_C_OID_KRBV5_DES` - Beta Kerberos V5 mechanism
    - `GSSDCE_C_OID_KRBV5_DES_RFC` - Kerberos V5 mechanisms

**Output**

- **name**
  - Returns the principal name associated with the credential. Specify `NULL` for this parameter if the principal name is not required.

- **init_lifetime**
  - Returns the number of seconds for which the credential will remain valid for initiating contexts. Specify `NULL` for this parameter if the credential lifetime is not required.

- **accept_lifetime**
  - Returns the number of seconds for which the credential will remain valid for accepting contexts. Specify `NULL` for this parameter if the credential lifetime is not required.

- **cred_usage**
  - Returns one of the following values describing how the application can use the credential. Specify `NULL` for this parameter if the credential usage is not required.
    - `GSS_C_INITIATE` - the application may initiate a security context.
    - `GSS_C_ACCEPT` - the application may accept a security context.
    - `GSS_C_BOTH` - the application may both initiate and accept security contexts.

- **minor_status**
  - Returns a status code from the security mechanism.

**Usage**

The `gss_inquire_cred_by_mech()` routine provides information about a GSSAPI credential to the calling application. The information is obtained using the specified security mechanism.

**Status Codes**

This routine returns the following status codes:

- **GSS_S_COMPLETE**
  - The routine completed successfully.
gss_inquire_cred_by_mech

GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_BAD_MECH The requested mechanism is not supported.

GSS_S_DEFECTIVE_CREDENTIAL The credentials are not valid.

GSS_S_NO_CRED The cred_handle does not refer to a valid credential or there are no default credentials available.

GSS_S_CREDENTIALS_EXPIRED The credentials have expired.
gss_inquire_mechs_for_name

Purpose
Returns the mechanisms with which a name may be processed.

Format
#include <dce/gssapi.h>

OM_uint32 gss_inquire_mechs_for_name (  
    OM_uint32 *minor_status,  
    gss_name_t input_name,  
    gss_OID_set *mech_types)

Parameters
Input
input_name Specifies the name to be queried.

Output
mech_types Returns the mechanisms which can be used with the specified name. The gss_OID_set returned for this parameter should be released by calling the gss_release_oid_set() routine when it is no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gss_inquire_mechs_for_name() routines returns the set of mechanisms which can be used with a given name.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_BAD_NAME The supplied name is not valid.
GSS_S_BAD_NAMETYPE The name type is not supported.
gss_inquire_names_for_mech

Purpose
Returns the name types supported by a security mechanism.

Format
#include <dce/gssapi.h>

OM_uint32 gss_inquire_names_for_mech (  
    OM_uint32 *minor_status,  
    gss_OID mech_type,  
    gss_OID_set *mech_names)

Parameters
Input
mech_type Specifies the mechanism to be queried as follows:

- GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism
- GSSDCE_C_OID_KRBV5_DES - Beta Kerberos V5 mechanism
- GSSDCE_C_OID_KRBV5_DES_RFC - Kerberos V5 mechanism

Output
mech_names Returns the name types supported by the specified mechanism. The gss_OID_set returned for this parameter should be released by calling the gss_release_oid_set() routine when it is no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gss_inquire_names_for_mech() routine returns the set of name types which are supported by a particular security mechanism.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_BAD_MECH The requested mechanism is not supported.
gss_oid_to_str

Purpose
Converting a gss_OID object to a string representation of the object identifier.

Format
#include <dce/gssapi.h>

OM_uint32 gss_oid_to_str (  
    OM_uint32 *minor_status,  
    gss_OID input_oid,  
    gss_buffer_t output_string)

Parameters
Input
input_oid Specifies the gss_OID to be converted.

Output
output_string Returns the string representation of the object identifier. The gss_buffer_t returned for this parameter should be released by calling the gss_release_buffer() routine when it is no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gss_oid_to_str() routine converts a gss_OID object to a string representation of the object identifier. The string representation consists of a series of blank-separated numbers enclosed in braces. The gss_str_to_oid() routine can be used to convert the string representation back to a gss_OID object.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.

GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
gss_process_context_token

Purpose
Processes a context token received from the partner application.

Format
#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)

Parameters
Input
context_handle    Specifies the context which should be used when processing the token.
input_token       Specifies the token received from the partner application.

Output
minor_status      Returns a status code from the security mechanism.

Usage
The gss_process_context_token() routine processes tokens generated by the partner application. Tokens are usually associated with either the context establishment or with message security services. If the tokens are associated with the context establishment, they are processed by the gss_init_sec_context() and gss_accept_sec_context() routines. If the tokens are associated with message security services, they are processed by the gss_verify_mic() and gss_unwrap() routines. Tokens generated by the gss_delete_sec_context() routine, however, are processed by the gss_process_context_token() routine.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE     The routine completed successfully.
GSS_S_FAILURE      The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_NO_CONTEXT   The context handle does not refer to a valid security context.
GSS_S_BAD_SIG      The token signature was not correct.
GSS_S_DEFECTIVE_TOKEN   Consistency checks performed on the input token failed.
gss_release_buffer

Purpose
Releases storage associated with a gss_buffer_t buffer. The gss_buffer_desc structure itself is not released.

Format
#include <dce/gssapi.h>

OM_uint32 gss_release_buffer (  
  OM_uint32 *minor_status,  
  gss_buffer_t buffer)

Parameters

Input/Output
buffer Specifies the buffer to be released. Upon successful completion, the length and value fields will be set to zero.

Output
minor_status Returns a status code from the security mechanism.

Usage
The gss_release_buffer() routine releases storage associated with a gss_buffer_t buffer. It does not release the storage for the gss_buffer_desc structure itself.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
gss_release_cred

Purpose
Releases local data structures associated with a GSSAPI credential.

Format
#include <dce/gssapi.h>

OM_uint32 gss_release_cred (  
    OM_uint32 *minor_status,  
    gss_cred_id_t *cred_handle)

Parameters
Input/Output
cred_handle Specifies the credential to be released. Upon successful completion, the cred_handle value will be set to GSS_C_NO_CREDENTIAL. If the cred_handle value is GSS_C_NO_CREDENTIAL, the major status will be set to GSS_S_COMPLETE and nothing will be released.

Output
minor_status Returns a status code from the security mechanism.

Usage
The gss_release_cred() routine releases the local data structures for the specified credential. If GSS_C_NO_CREDENTIAL is specified for the cred_handle parameter, no credential will be released and GSS_S_COMPLETE will be returned for the major status.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_DEFECTIVE_CREDENTIAL Consistency checks performed on the credential structure failed.
GSS_S_NO_CRED The cred_handle parameter does not refer to a valid credential.
gss_release_name

Purpose
Releases storage associated with a gss_name_t internal name.

Format
#include <dce/gssapi.h>

OM_uint32 gss_release_name (
   OM_uint32  *minor_status,
   gss_name_t  *name)

Parameters
Input/Output

name
   Specifies the name to be released. Upon successful completion, the name value will be set to GSS_C_NO_NAME.

Output

minor_status
   Returns a status code from the security mechanism.

Usage
The gss_release_name() routine releases storage associated with a GSSAPI internal name.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE
   The routine completed successfully.

GSS_S_FAILURE
   The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_BAD_NAME
   The specified name is not valid.
Purpose
Releases the storage associated with a gss_OID object.

Format
#include <dce/gssapi.h>

OM_uint32 gss_release_oid (  
   OM_uint32 *minor_status,  
   gss_OID *oid)

Parameters
Input/Output

oid
Specifies the gss_OID to be released. Upon successful completion, the OID value will be set to GSS_C_NO_OID.

Output

minor_status
Returns a status code from the security mechanism.

Usage
The gss_release_oid() routine releases the storage associated with a gss_OID object. The gss_release_oid() routine must not be called to release a read-only OID which was returned by one of the GSSAPI routines.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE
The routine completed successfully.

GSS_S_FAILURE
The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
gss_release_oid_set

Purpose
Releases the storage associated with a gss_OID_set object.

Format
#include <dce/gssapi.h>

OM_uint32 gss_release_oid_set (  
    OM_uint32 *minor_status,  
    gss_OID_set *oid_set)

Parameters
Input/Output
   oid_set   Specifies the gss_OID_set to be released. Upon successful completion, the oid_set value will be set to GSS_C_NO_OID_SET.

Output
   minor_status    Returns a status code from the security mechanism.

Usage
The gss_release_oid_set() routine releases the storage associated with a gss_OID_set object.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE        The routine completed successfully.
GSS_S_FAILURE         The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
gss_str_to_oid

Purpose
Converts the string representation of an object identifier to a gss_OID object.

Format
#include <dce/gssapi.h>

OM_uint32 gss_str_to_oid (  
    OM_uint32 *minor_status,  
    gss_buffer_t input_string,  
    gss_OID *output_oid)

Parameters
Input
input_string Specifies the string to be converted.

Output
output_oid Returns the object identifier. The gss_OID returned for this parameter should be released by calling the gss_release_oid() routine when it is no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gss_str_to_oid() routine converts the string representation of an object identifier to a gss_OID object. The string representation is a series of blank-separated or period-separated numbers enclosed in braces. For example, the DCE security mechanism object identifier is represented as {1 3 24 9 8}.

While the blank-separated form should be used for portability, the gss_str_to_oid() routine will also accept the period-separated form for compatibility with other applications. However, the gss_oid_to_str() routine will always generate the blank-separated form.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
Purpose
Checks an OID set to see if a specified OID is in the set.

Note: The gss_test_oid_set_member() routine replaces the gssdce_test_oid_set_member() routine. While the gssdce_test_oid_set_member() routine is still available for compatibility with existing applications, new applications should use the gss_test_oid_set_member() routine.

Format
#include <dce/gssapi.h>

OM_uint32 gss_test_oid_set_member (  
    OM_uint32 *minor_status,  
    gss_OID member_oid,  
    gss_OID_set oid_set,  
    int *is_present)

Parameters

Input
member_oid Specifies the OID to search for in the OID set.
oid_set Specifies the OID set to check.

Output
is_present Is set to 1 if the OID is a member of the OID set and to zero otherwise.
minor_status Returns a status code from the security mechanism.

Usage
The gss_test_oid_set_member() routine checks an OID set to see if the specified OID is a member of the set. The gss_create_empty_oid_set() routine can be used to create an empty OID set and the gss_add_oid_set_member() routine can be used to add an OID to an existing OID set.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
**gss_unwrap**

**Purpose**
Unwraps a message sealed by the `gss_wrap()` routine and verifies the embedded signature.

**Note:** The `gss_unwrap()` routine replaces the `gss_unseal()` routine. While the `gss_unseal()` routine is still available for compatibility with existing applications, new applications should use the `gss_unwrap()` routine.

**Format**
```
#include <dce/gssapi.h>

OM_uint32 gss_unwrap (    
    OM_uint32 *minor_status,  
    gss_ctx_id_t context_handle,  
    gss_buffer_t input_message,  
    gss_buffer_t output_message,  
    int *conf_state,  
    gss_qop_t *qop_state)
```

**Parameters**

**Input**
- `context_handle` Specifies the context on which the message arrived.
- `input_message` Specifies the sealed message token generated by the `gss_wrap()` routine.

**Output**
- `output_message` Returns the unsealed message.
- `conf_state` Returns the level of confidentiality which was applied to the message. Specify `NULL` for this parameter if the confidentiality state is not needed. The return value will be set as follows:
  - `TRUE` - Both confidentiality and integrity services were applied.
  - `FALSE` - Only integrity services were applied.
- `qop_state` Returns the quality of protection which was applied to the message. Specify `NULL` for this parameter if the quality of protection is not needed.

The DCE security mechanism supports three quality-of-protection levels as follows:
- `GSSDCE_C_QOP_MD5` - Truncated MD5
- `GSSDCE_C_QOP_DES_MD5` - DES_MAC of an MD5 hash
- `GSSDCE_C_QOP_DES_MAC` - Normal DES_MAC algorithm

The Kerberos security mechanism supports three quality-of-protection levels as follows (in decreasing order or speed):
- `GSS_KRB5_INTEG_C_QOP_MD5` - Truncated MD5
- `GSS_KRB5_INTEG_C_QOP_DES_MD5` - DES_MAC of an MD5 hash
- `GSS_KRB5_INTEG_C_QOP_DES_MAC` - Normal DES_MAC algorithm

**Usage**
The `gss_unwrap()` routine extracts a message from the sealed token created by the `gss_wrap()` routine and verifies the embedded signature. The `conf_state` return parameter indicates whether the message had been encrypted.

**Status Codes**
This routine returns the following status codes:
- `GSS_S_COMPLETE` The routine completed successfully.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_FAILURE</td>
<td>The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.</td>
</tr>
<tr>
<td>GSS_S_NO_CONTEXT</td>
<td>The referenced context is not valid.</td>
</tr>
<tr>
<td>GSS_SCONTEXT_EXPIRED</td>
<td>The referenced context has expired.</td>
</tr>
<tr>
<td>GSS_S_CREDENTIALS_EXPIRED</td>
<td>The credentials associated with the referenced context have expired.</td>
</tr>
<tr>
<td>GSS_S_BAD_SIG</td>
<td>The token signature is not correct.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_TOKEN</td>
<td>Consistency checks performed on the input token failed.</td>
</tr>
<tr>
<td>GSS_S_DUPLICATE_TOKEN</td>
<td>The token is a duplicate of a token which has already been processed.</td>
</tr>
<tr>
<td>GSS_S_OLD_TOKEN</td>
<td>The token is too old to be checked for duplication against tokens which have already been processed.</td>
</tr>
<tr>
<td>GSS_S_UNSEQ_TOKEN</td>
<td>A later token has already been processed.</td>
</tr>
<tr>
<td>GSS_S_GAP_TOKEN</td>
<td>One or more predecessor tokens have not been processed.</td>
</tr>
</tbody>
</table>
**gss_verify_mic**

**Purpose**
Verifies that the cryptographic signature for a message is correct.

**Note:** The `gss_verify_mic()` routine replaces the `gss_verify()` routine. While the `gss_verify()` routine is still available for compatibility with existing applications, new applications should use the `gss_verify_mic()` routine.

**Format**
```c
#include <dce/gssapi.h>

OM_uint32 gss_verify_mic (  
    OM_uint32 *minor_status,  
    gss_ctx_id_t context_handle,  
    gss_buffer_t input_message,  
    gss_buffer_t input_token,  
    gss_qop_t *qop_state)
```

**Parameters**

**Input**
- `context_handle`: Specifies the context on which the message arrived.
- `input_message`: Specifies the message to be verified.
- `input_token`: Specifies the signature token generated by the `gss_get_mic()` routine.

**Output**
- `qop_state`: Returns the quality of protection which was applied to the message. Specify `NULL` for this parameter if the quality of protection is not needed.

The DCE security mechanism supports three quality-of-protection levels as follows:
- `GSSDCE_C_QOP_MD5` - Truncated MD5
- `GSSDCE_C_QOP_DES_MD5` - DES_MAC of an MD5 hash
- `GSSDCE_C_QOP_DES_MAC` - Normal DES_MAC algorithm

The Kerberos security mechanism supports three quality of protection levels as follows:
- `GSS_KRB5_INTEG_C_QOP_MD5` - Truncated MD5
- `GSS_KRB5_INTEG_C_QOP_DES_MD5` - DES_MAC of an MD5 hash
- `GSS_KRB5_INTEG_C_QOP_DES_MAC` - Normal DES_MAC algorithm

**minor_status**: Returns a status code from the security mechanism.

**Usage**
The `gss_verify_mic()` routine checks that the encrypted signature is the correct signature for the supplied message. This ensures that the message has not been modified since the signature was generated.

**Status Codes**
This routine returns the following status codes:

- **GSS_S_COMPLETE**: The routine completed successfully.
- **GSS_S_FAILURE**: The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_BAD_SIG**: The input token is not valid.
- **GSS_S_NO_CONTEXT**: The referenced context does not exist.
- **GSS_S_CONTEXT_EXPIRED**: The referenced context has expired.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_CREDENTIALS_EXPIRED</td>
<td>The credentials associated with the referenced context have expired.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_CREDENTIAL</td>
<td>The credential is defective.</td>
</tr>
<tr>
<td>GSS_S_DEFECTIVE_TOKEN</td>
<td>Consistency checks performed on the input token failed.</td>
</tr>
<tr>
<td>GSS_S_DUPLICATE_TOKEN</td>
<td>The input token is a duplicate of a token which has already been processed.</td>
</tr>
<tr>
<td>GSS_S_OLD_TOKEN</td>
<td>The input token is too old to be checked for duplication against tokens which have already been processed.</td>
</tr>
<tr>
<td>GSS_S_UNSEQ_TOKEN</td>
<td>A later token has already been processed.</td>
</tr>
<tr>
<td>GSS_S_GAP_TOKEN</td>
<td>One or more predecessor tokens have not been processed.</td>
</tr>
</tbody>
</table>
gss_wrap

Purpose
Cryptographically signs and optionally encrypts a message.

Note: The gss_wrap() routine replaces the gss_seal() routine. While the gss_seal() routine is still available for compatibility with existing applications, new applications should use the gss_wrap() routine.

Format
#include <dce/gssapi.h>

OM_uint32 gss_wrap (  
   OM_uint32 *minor_status,  
   gss_ctx_id_t context_handle,  
   int conf_req_flag,  
   gss_qop_t qop_req,  
   gss_buffer_t input_message,  
   int *conf_state,  
   gss_buffer_t output_message)  

Parameters

Input  
context_handle  
   Specifies the context to be associated with the message when it is sent to the partner application.

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 requested quality of protection for the message. Specify GSS_C_QOP_DEFAULT to use the default quality of protection as defined by the selected security mechanism.

   The DCE security mechanism supports three quality-of-protection levels as follows (in decreasing order of speed):
   
   GSSDCE_C_QOP_MD5 - Truncated MD5 (default)
   GSSDCE_C_QOP_DES_MD5 - DES_MAC of an MD5 hash
   GSSDCE_C_QOP_DES_MAC - Normal DES_MAC algorithm

   The Kerberos security mechanism supports three quality-of-protection levels as follows (in decreasing order or speed):
   
   GSS_KRB5_INTEG_C_QOP_MD5 - Truncated MD5
   GSS_KRB5_INTEG_C_QOP_DES_MD5 - DES_MAC of an MD5 hash (default)
   GSS_KRB5_INTEG_C_QOP_DES_MAC - Normal DES_MAC algorithm

input_message  
   Specifies the message to be wrapped.

Output  
conf_state  
   Returns the level of confidentiality which was applied to the message. Specify NULL for this parameter if the confidentiality state is not required. The return value will be set as follows:
   
   TRUE - Both confidentiality and integrity services have been applied
   FALSE - Only integrity services have been applied.

output_message  
   Returns the wrapped message. The buffer should be released when it is no longer needed by calling the gss_release_buffer() routine.

minor_status  
   Returns a status code from the security mechanism.
Usage

The `gss_wrap()` routine cryptographically signs and optionally encrypts a message. The token returned in the `output_message` parameter contains both the signature and the message. This token is then sent to the partner application which calls the `gss_unwrap()` routine to extract the original message and verify its authenticity.

If confidentiality is requested (the `conf_req_flag` is `TRUE`) but confidentiality services are not available for the security context, no error will be returned and only integrity services will be performed. The `conf_state` return parameter will indicate whether the requested confidentiality services were performed.

Status Codes

This routine returns the following status codes:

- **GSS_S_COMPLETE**: The routine completed successfully.
- **GSS_S_FAILURE**: The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_NO_CONTEXT**: The referenced context does not exist.
- **GSS_S_CONTEXT_EXPIRED**: The referenced context has expired.
- **GSS_S_CREDENTIALS_EXPIRED**: The credentials associated with the referenced context have expired.
- **GSS_S_BAD_QOP**: The quality-of-protection value is not valid.
Purpose
Determines the largest message which can be wrapped without exceeding a maximum size limit.

Format
#include <dce/gssapi.h>

OM_uint32 gss_wrap_size_limit(
    OM_uint32 *minor_status,
    gss_ctx_id_t context_handle,
    int conf_req,
    gss_qop_t qop_req,
    OM_uint32 size_req,
    OM_uint32 *max_size)

Parameters

Input
context_handle
Specifies the security context which will be associated with the messages.

conf_req
Specifies whether confidentiality services will be requested for the messages as follows:
- TRUE - Confidentiality services will be requested in addition to integrity and authentication services.
- FALSE - Only integrity and authentication services will be requested

qop_req
Specifies the quality of protection which will be used with the messages. Specify GSS_C_QOP_DEFAULT to use the default quality of protection as defined by the selected security mechanism.
The DCE security mechanism supports three quality-of-protection levels as follows (in decreasing order of speed but increasing protection):
- GSSDCE_C_QOP_MD5 - Truncated MD5 (default)
- GSSDCE_C_QOP_DES_MD5 - DES_MAC of an MD5 hash
- GSSDCE_C_QOP_DES_MAC - Normal DES_MAC algorithm

size_req
Specifies the maximum output token size.

Output
max_size
Returns the maximum message size which can be processed without exceeding the specified maximum token size.

minor_status
Returns a status code from the security mechanism.

Usage
The gss_wrap_size_limit() routine returns the maximum input message size which can be processed by the gss_wrap() routine without exceeding the specified output token size.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE
The routine completed successfully.

GSS_S_FAILURE
The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_NO_CONTEXT
The referenced context does not exist.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_CONTEXT_EXPIRED</td>
<td>The referenced context has expired.</td>
</tr>
<tr>
<td>GSS_S_BAD_QOP</td>
<td>The quality of protection requested is not valid.</td>
</tr>
</tbody>
</table>
gssdce_cred_to_login_context

Purpose
Obtains the DCE login context associated with a GSSAPI credential.

Format
#include <dce/gssapi.h>

OM_uint32 gssdce_cred_to_login_context (  
   OM_uint32  *minor_status,  
   gss_cred_id_t  cred_handle,  
   sec_login_handle_t  *login_context)

Parameters
Input
cred_handle Specifies the credential handle. Specify GSS_C_NO_CREDENTIAL to obtain the default login context.

Output
login_context Returns the DCE login context associated with the credential. The application should release the login context when it is no longer needed by calling the sec_login_release_context() routine.

minor_status Returns a status code from the security mechanism.

Usage
The gssdce_cred_to_login_context() routine allows an application to obtain the DCE login context associated with a GSSAPI credential. The credential must support the DCE security mechanism in order to have an associated DCE login context. In addition, only INITIATE and BOTH credentials have DCE login contexts. The gssdce_cred_to_login_context() routine may also be used to obtain the delegated login context created by the gss_accept_sec_context() routine when it receives delegated credentials from the initiator.

The application should release the login context when it is no longer needed by calling the sec_login_release_context() routine. However, if the login context is the default login context, the application should not release the login context unless it also wants to release the default login context. The login context will be the default login context if GSS_C_NO_CREDENTIAL is specified for the cred_handle parameter or if the GSSAPI credential was created using the default login context.

The application can request GSSAPI to release the login context when the credential is released by calling the gssdce_set_cred_context_ownership() routine. However, this has the same implications as calling sec_login_release_context() directly. That is, the application should not assign login context ownership to GSSAPI for the default login context unless the application wants the default login context to be released when the credential is released.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.

GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_BAD_MECH The referenced credential does not support the DCE security mechanism.

GSS_S_CREDENTIALS_EXPIRED The referenced credential has expired.

GSS_S_DEFECTIVE_CREDENTIAL The referenced credential is defective.

GSS_S_NO_CRED The referenced credentials are not valid.
gssdce_export_cred

Purpose
Create a portable version of a GSSAPI credential.

Format

```c
#include <dce/gssapi.h>
OM_uint32 gssdce_export_cred (
    OM_uint32 *minor_status,
    gss_cred_id_t cred_handle,
    gss_buffer_t export_data);
```

Parameters

Input
cred_handle Specifies the handle for the GSSAPI credential.

Output
export_data Returns the exported credential data. The length field will be set to the length of the data and the value field will be set to the address of the data. The application should free the data when it is no longer needed by calling the gss_release_buffer() routine.

minor_status Returns a status code from the security mechanism.

Possible status codes and their meanings:

Usage
The gssdce_export_cred() routine obtains a portable version of a GSSAPI credential created by the DCE security mechanism. This information may then be passed to another process running on the same system or on a different system. The GSSAPI credential which has been exported remains available on the local system at the completion of the export operation.

Status Codes
This routine returns the following status codes:

- **GSS_S_COMPLETE** The routine completed successfully.
- **GSS_S_FAILURE** The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_CREDENTIALS_EXPIRED** The referenced credential has expired.
- **GSS_S_DEFECTIVE_CREDENTIAL** The referenced credential is defective.
- **GSS_S_NO_CRED** The referenced credentials are not valid.
Purpose
Extracts DCE credentials from a GSSAPI acceptor security context.

Format
#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 security context containing the DCE credentials.

Output
output_cred Returns the DCE credentials.
minor_status Returns a status code from the security mechanism.

Usage
The gssdce_extract_creds_from_sec_context() routine extracts the context initiator's DCE credentials from a context acceptor's security context. The context must have been created using the DCE security mechanism. The DCE credentials contain the privilege attributes of the context initiator and are used by ACL managers to determine whether the initiator has the right to access the object to which an ACL refers.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_NOCONTEXT The referenced context does not exist.
GSS_S_BAD_MECH The security context was not created using the DCE security mechanism.
**gssdce_extract_PAC_from_cred**

**Purpose**
Extracts a DCE privilege attribute certificate (PAC) from a GSSAPI credential.

**Format**
```c
#include <dce/gssapi.h>

OM_uint32 gssdce_extract_PAC_from_cred (  
    OM_uint32 minor_status,  
    gss_cred_id_t cred_handle,  
    sec_id_pac_t *output_pac)
```

**Parameters**

**Input**
- `cred_handle`: Specifies the GSSAPI credential. Specify `GSS_C_NO_CREDENTIAL` to extract the PAC from the default login context.

**Output**
- `output_pac`: Returns the DCE privilege attribute certificate. The PAC substructures should be released by the application when they are no longer needed.
- `minor_status`: Returns a status code from the security mechanism.

**Usage**
The `gssdce_extract_PAC_from_cred()` routine extracts a DCE privilege attribute certificate (PAC) from a GSSAPI credential. The credential must support the DCE security mechanism in order to have an associated DCE login context. In addition, only `INITIATE` and `BOTH` credentials have DCE login contexts.

**Status Codes**
This routine returns the following status codes:
- **GSS_S_COMPLETE**: The routine completed successfully.
- **GSS_S_FAILURE**: The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_BAD_MECH**: The GSSAPI credential does not support the DCE security mechanism.
- **GSS_S_DEFECTIVE_CREDENTIAL**: The credential is defective.
- **GSS_S_NO_CRED**: The credential could not be accessed.
Purpose
Extracts a DCE privilege attribute certificate (PAC) from a GSSAPI acceptor security context.

Format
#include <dce/gssapi.h>

OM_uint32 gssdce_extract_PAC_from_sec_context (  
    OM_uint32 *minor_status,  
    gss_ctx_id_t context_handle,  
    sec_id_pac_t *output_pac)

Parameters
Input
context_handle Specifies the security context containing the GSSAPI credential.

Output
output_pac    Returns the DCE privilege attribute certificate. The PAC substructures should be released by the application when they are no longer needed.

minor_status Returns a status code from the security mechanism.

Usage
The gssdce_extract_PAC_from_sec_context() routine extracts a DCE privilege attribute certificate (PAC) from a security context created by the gss_accept_sec_context() routine. The PAC represents the principal associated with the security context initiator.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_BAD_MECH The security context was not created using the DCE security mechanism.
GSS_S_DEFECTIVE_CREDENTIAL The credential is defective.
GSS_S_NO_CRED The credential could not be accessed.
GSS_S_NO_CONTEXT The referenced context does not exist.
gssdce_import_cred

Purpose
Create a GSSAPI credential from the portable data created by the gssdce_export_cred() routine.

Format
#include <dce/gssapi.h>
OM_uint32 gssdce_import_cred (  
    OM_uint32 minor_status ,
    gss_buffer_t export_data ,
    gss_cred_id_t *cred_handle );

Parameters
Input
export_data The portable data created by the gssdce_export_cred() routine. The length field specifies the length of the data and the value field specifies the address of the data.

Output
cred_handle Returns the handle for the GSSAPI credential. The application should release the credential when it is no longer needed by calling the gss_release_cred() routine.
minor_status Returns a status code from the security mechanism.
Possible status codes and their meanings:

Usage
The gssdce_import_cred() routine will create a GSSAPI credential from the portable data created by the gssdce_export_cred() routine. The application is responsible for releasing the credential when it is no longer needed by calling the gss_release_cred() routine.

Status Codes
This routine returns the following status codes:
GGSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

Purpose
Creates a GSSAPI credential from a DCE login context.

Format
#include <dce/gssapi.h>

OM_uint32 gssdce_login_context_to_cred (  
  OM_uint32 *minor_status,  
  sec_login_handle_t login_context,  
  OM_uint32 time_req,  
  gss_OID_set desired_mechs,  
  gss_cred_id_t *output_cred_handle,  
  gss_OID_set *actual_mechs,  
  OM_uint32 *time_rec)

Parameters

Input

credentials
  Specifies the DCE login context to be used. Specify NULL to use the default DCE login context.

time_req
  Specifies the number of seconds that the credential should remain valid. Specify zero for the 
  default lifetime of two hours. Specify GSS_C_INDEFINITE to request the maximum lifetime.

desired_mechs
  Specifies the security mechanisms to be used with the credential. Specify 
  GSS_C_NO_OID_SET to use the default DCE mechanism. The following mechanism may be 
  specified:
    - GSSDCE_C_OID_DCE_KRBV5_DES - DCE mechanism

Output

output_cred_handle
  Returns the credential handle.

actual_mechs
  Returns the actual mechanisms which will be supported by the credential. Specify NULL for this 
  parameter if the actual mechanisms are not required.

time_rec
  Returns the number of seconds that the credential will remain valid. GSS_C_INDEFINITE will be 
  returned if there is no expiration time. Specify NULL for this parameter if the credential lifetime 
  is not required.

minor_status
  Returns a status code from the security mechanism.

Usage
The gssdce_login_context_to_cred() routine creates a GSSAPI credential from a DCE login context. The credential can be 
used to initiate or accept a security context.

Status Codes

This routine returns the following status codes:

GSS_S_COMPLETE  The routine completed successfully.

GSS_S_FAILURE   The routine failed for reasons which are not defined at the GSS level. The 
                minor_status return parameter contains a mechanism-dependent error code 
                describing the reason for the failure.

GSS_S_CREDENTIAL_EXPIRED  The login context has expired.
gssdce_register_acceptor_identity

Purpose
Registers a context acceptor's identity.

Format
#include <dce/gssapi.h>

OM_uint32 gssdce_register_acceptor_identity (  
  OM_uint32 *minor_status,  
  gss_name_t princ_name,  
  rpc_auth_key_retrieval_fn_t get_key_fn,  
  void *get_key_arg)

Parameters
Input
princ_name
Specifies the principal name to use for the context acceptor. The name must contain an internal representation for the DCE mechanism. The gss_import_name() routine can be used to create a GSSAPI internal name from an external representation.

get_key_fn
Specifies the address of a routine which will return the encryption key. Specify NULL to use the default key-retrieval routine.

get_key_arg
Specifies the argument to be passed to the key-retrieval routine. For the default key-retrieval routine, this is the name of the key table to be used and can be specified as NULL to use the default key table. Otherwise, the meaning of this parameter will depend upon the key-retrieval routine which is being used.

Output
minor_status
Returns a status code from the security mechanism.

Usage
The gssdce_register_acceptor_identity() routine registers the specified principal name as an identity claimed by the context acceptor. The supplied key-retrieval function will be called by the GSSAPI routines whenever they need to obtain the encryption key for the principal.

If NULL is specified for the get_key_fn parameter, the default key retrieval function will be used. This routine obtains the principal's key using a Kerberos key table. In this case, the get_key_arg parameter is the name of the key table in the format FILE:/pathname/filename. If NULL is specified for the get_key_arg parameter as well as for the get_key_fn parameter, the key will be obtained from the /krb5/v5srvtab key table will be used. Refer to "rpc_server_register_auth_info" on page 2-258 for more information on the key retrieval parameters.

You must specify NULL for the get_key_fn parameter if you want the GSSAPI routines to implicitly create a login context as part of acquiring a GSSAPI credential. For example, the default key-retrieval function must be used if the gss_acquire_cred() routine is called to obtain a GSS_C_BOTH credential.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
Purpose
Changes the ownership of a DCE login context which is associated with a GSSAPI credential.

Format
```c
#include <dce/gssapi.h>

OM_uint32 gssdce_set_cred_context_ownership (  
    OM_uint32 *minor_status,  
    gss_cred_id_t cred_handle,  
    int ownership)
```

Parameters
Input
- `cred_handle`: Specifies the GSSAPI credential whose associated DCE login context is to be changed.
- `ownership`: Specifies the ownership of the DCE login context as follows:
  - `GSSDCE_C_OWNERSHIP_GSSAPI`: The login context is owned by GSSAPI and will be released when it is no longer needed by the GSSAPI routines.
  - `GSSDCE_C_OWNERSHIP_APPLICATION`: The login context is owned by the application. It is the responsibility of the application to call the `sec_login_release_context()` routine when the login context is no longer needed.

Output
- `minor_status`: Returns a status code from the security mechanism.

Usage
The `gssdce_set_cred_context_ownership()` routine modifies the ownership of the DCE login context associated with a GSSAPI credential. Normally, login contexts which are created by GSSAPI routines will be released when they are no longer needed. However, for credentials created by the `gssdce_login_context_to_cred()` routine and credentials passed to the `gssdce_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. As a result, the GSSAPI routines will not delete these login contexts when the GSSAPI credential is released unless ownership has been assigned to GSSAPI.

If the credential context ownership is set to `GSSDCE_C_OWNERSHIP_GSSAPI`, the login context will be released when the GSSAPI credential is released. If the credential context ownership is set to `GSSDCE_C_OWNERSHIP_APPLICATION`, the application is responsible for releasing the login context when it is no longer needed. In this case, the application must not release the login context before it has released the associated GSSAPI credential since GSSAPI may still be using the login context.

Credential context ownership will revert to the application whenever a GSSAPI routine returns an external reference to the login context. Thus, the application should call the `gssdce_set_cred_context_ownership()` routine immediately before calling the `gss_release_cred()` routine to ensure that the ownership is set properly before GSSAPI releases the credential.

Status Codes
This routine returns the following status codes:

- **GSS_S_COMPLETE**: The routine completed successfully.
- **GSS_S_FAILURE**: The routine failed for reasons which are not defined at the GSS level. The `minor_status` return parameter contains a mechanism-dependent error code describing the reason for the failure.
- **GSS_S_NO_CRED**: The referenced credential does not exist.
- **GSS_S_DEFECTIVE_CREDENTIAL**: Consistency checks performed on the credential failed.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_S_BAD_MECH</td>
<td>The credential does not support the DCE security mechanism.</td>
</tr>
</tbody>
</table>
Purpose
Returns the Kerberos credentials cache associated with a GSSAPI credential.

Format
#include <dce/gssapi.h>

OM_uint32 gss_krb5_get_ccache (  
    OM_uint32 *minor_status,  
    gss_cred_id_t cred_handle,  
    krb5_ccache *ccache)

Parameters

Input
cred_handle Specifies the handle for the GSSAPI credential.

Output
ccache Returns the handle for the credentials cache. A NULL value will be returned if there is no credentials cache associated with the GSSAPI credential.

minor_status Returns a status code from the security mechanism.

Usage
The gss_krb5_get_ccache() routine returns the handle for the credentials cache which is associated with the GSSAPI credential. The application must not close or destroy this credentials cache. The returned handle will no longer be valid once the GSSAPI credential has been released.

Status Codes

This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.

GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.

GSS_S_NO_CRED The credential handle does not refer to a valid GSSAPI credential.
gss_krb5_get_tkt_flags

Purpose
Returns the Kerberos ticket flags from the service ticket.

Format
#include <dce/gssapi.h>

OM_uint32 gss_krb5_get_tkt_flags (  
    OM_uint32 *minor_status,  
    gss_ctx_id_t context_handle,  
    krb5_flags *tkt_flags)

Parameters
Input
context_handle Specifies the handle for the GSSAPI security context.

Output
tkt_flags Returns the ticket flags from the Kerberos ticket associated with the security context.
minor_status Returns a status code from the security mechanism.

Usage
The gss_krb5_get_tkt_flags() routine returns the ticket flags from the Kerberos ticket associated with the security context. Refer to Chapter 7, "Kerberos" on page 7-1 for a description of the various flags.

Status Codes
This routine returns the following status codes:

GSS_S_COMPLETE The routine completed successfully.
GSS_S_FAILURE The routine failed for reasons which are not defined at the GSS level. The minor_status return parameter contains a mechanism-dependent error code describing the reason for the failure.
GSS_S_NO_CONTEXT The context handle does not refer to a valid security context.
rdacl Routines

This section describes in detail the following rdacl APIs:

- rdACL_get_access
- rdACL_get_manager_types
- rdACL_get_mgr_types_semantics
- rdACL_get_printstring
- rdACL_get_referral
- rdACL_lookup
- rdACL_replace
- rdACL_test_access
- rdACL_test_access_on.behalf
**rdacl_get_access**

Reads a privilege attribute certificate.

**Format**

```c
#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. 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 between 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 one of the following errors:

  - `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.
  - `sec_rgy_bad_data`
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `rdacl_get_access` API determines the complete extent of access to the specified object by the calling process. Although the `rdacl_test_access` APIs are the preferred method of testing access, this API is useful for doing operations like the conventional UNIX access function.

The `rdacl_` APIs are provided for programmers who want to write an ACL manager. If a DCE application server is to contain an ACL manager, write the `rdacl_` API for the server portion of the DCE application. The function will be started by client `sec_acl` API calls. In addition, you must compile the `rdaclif.idl` file and link the `rdaclif` server stub generated into your server application.

This network interface is called on the client side through the `sec_acl` local interface. Application developers are responsible for writing the server side of this interface using the `rdacl` APIs. Test server code is included as a sample implementation.
rdacl_get_access

Related Information

Routines

ACL API          rdacl_test_access
rdacl_get_manager_types

Lists the types of ACLs protecting an object.

Format

```c
#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. Possible values are:
    - `sec_acl_type_object`
    - `sec_acl_type_default_object`
    - `sec_acl_type_default_container`

**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 the following error.
    - `sec_rgy_bad_data`
      - Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

   If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

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 to display the permissions supported by a specific manager.
The rdac1_ APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the rdac1_ API for the server portion of the DCE application, The function will be started by client sec_act API calls. In addition, you must compile the rdac1f.idl file and link the rdac1f server stub generated into your server application.

This network interface is called on the client side through the sec_act local interface. Application developers are responsible for writing the server side of this interface using the rdac1 APIs. Test server code is included as a sample implementation.

Related Information
Routines

<table>
<thead>
<tr>
<th>ACL API</th>
<th>rdac1_get_printstring</th>
</tr>
</thead>
</table>

rdacl_get_mgr_types_semantics

Lists the ACL manager types protecting an object and the POSIX semantics supported by each manager type

Format

```c
#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. Possible values are:
  - `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[]` 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. This is always equal to `size_used`.
- **num_types** An unsigned 32-bit integer containing the number of types returned in the `manager_types[]` array.
- **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.

Usage

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, etc.) 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.
The `rdacl_` APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the `rdacl_` API for the server portion of the DCE application. The function will be started by client `sec_acl` API calls. In addition, you must compile the `rdaclif.idl` file and link the `rdaclif` server stub generated into your server application.

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.

**Related Information**

**Routines**

`rdacl_get_printstring`

**Files**

`/usr/include/dce/rdaclif.idl`  The idl file from which `dce/rdaclif.h` was derived.
rdacl_get_printstring

Returns printable ACL strings.

Format

```c
#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. 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 between 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 help string for the given ACL manager.
- **tokenize**: When FALSE this variable indicates that the returned permission print strings are unambiguous and therefore may be concatenated when printed. 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 print strings supported by this ACL manager type.
- **size_used**: An unsigned 32-bit integer containing the number of permission entries returned in the `printstrings` array.
- **print strings**: An array of permission print strings 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 be represented with the companion print string.

The array consists of one such entry for each permission supported by the ACL manager identified by `manager_type`.

Chapter 6. Security and Related Services  6-151
**rdacl_get_printstring**

**status**

A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

- **sec_acl_unknown_manager_type**
  
  The manager type selected is not among those referred to by the input handle.

- **sec_rgy_bad_data**
  
  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `rdacl_get_printstring` API returns an array of printable representations (`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 print strings, 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`. However, when the `tokenize` variable is `TRUE`, it implies the print strings might be of a form like `read` or `write`, which must be displayed separated by spaces, colons, or some other character.

In any list of permission print strings, 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.

The `rdacl_` APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the `rdacl_` API for the server portion of the DCE application, The function will be started by client `sec_acl` API calls. In addition, you must compile the `rdaclif.idl` file and link the `rdaclif` server stub generated into your server application.

This network interface is called on the client side through the `sec_acl` local interface. Application developers are responsible for writing the server side of this interface using the `rdacl` APIs. Test server code is included as a sample implementation.

**Related Information**

**Routines**

- **ACL API**
  - `sec_acl_bind`
  - `rdacl_get_manager_types`

6-152 Application Development Reference Volumes 1 and 2
rdacl_get_referral

Obtains a referral to an ACL update site.

Format

```c
#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. 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 between 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. Possible values are:
  - `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 function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_acl_unknown_manager_type` - Manager type selected is not an available option.
  - `sec_rgy_bad_data` - Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `rdacl_get_referral` API obtains a referral to an ACL update site. This function is used when the current ACL site returns 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.
rdacl_get_referral

The rdacI_ APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the rdacI_ API for the server portion of the DCE application. The function will be started by client sec_act API calls. In addition, you must compile the rdacIif.idl file and link the rdacIif server stub generated into your server application.

This network interface is called on the client side through the sec_act local interface. Application developers are responsible for writing the server side of this interface using the rdacI APIs. Test server code is included as a sample implementation.

Related Information

Routines

ACL API
**rdacl_lookup**

Returns the ACL for an object.

**Format**

```c
#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. 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 between 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. Possible values are:
  - `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`, and if equal to `error_status_ok`, the union contains an ACL. Otherwise, the completion status indicates an error, and the union is empty.
  
The errors that `result.st` can return are:
  - `sec_acl_unknown_manager_type`
  - `sec_acl_cant_allocate_memory`

  This strategy avoids pointlessly sending an ACL across the network when there has been an error.

  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 it is larger than 32 bits, multiple `sec_acl__ts` will be pointed to.

**Usage**

The `rdacl_lookup` API loads into storage 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.

The storage containing the `sec_acl_t` structure is dynamically allocated. When an application is finished with an ACL, it may use `sec_acl_release` to return to the pool that the storage block associated with the ACL object.

The `rdacl_` APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the `rdacl_` API for the server portion of the DCE application, The function will be started by client
rdacl_lookup

sec_acl API calls. In addition, you must compile the rdclifidl file and link the rdclif server stub generated into your server application.

This network interface is called on the client side through the sec_acl local interface. Application developers are responsible for writing the server side of this interface using the rdcl APIs. Test server code is included as a sample implementation.

Related Information

Routines

<table>
<thead>
<tr>
<th>ACL API</th>
<th>sec_acl_bind</th>
<th>sec_acl_test_access</th>
<th>sec_acl_test_access_on_behalf</th>
</tr>
</thead>
</table>

Routines
rdacl_replace

Replaces an ACL.

Format

```c
#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. 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 between 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. Possible values are:
  - `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 ACL. 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 one of the following errors:
  - `sec_acl_unknown_manager_type`: Manager type selected is not an available option.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `rdacl_replace` API replaces the ACL indicated by the input handle with the information in the `sec_acl_list` structure. ACLs are thought of as immutable, and in order to change them, an editing application must read an entire ACL (using `sec_acl_lookup`), change it as needed, and replace it using this routine.

The `rdacl_` APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the `rdacl_` API for the server portion of the DCE application. The function will be started by client `sec_acl` API calls. In addition, you must compile the `rdaclif.idl` file and link the `rdaclif` server stub generated into your server application.
This network interface is called on the client side through the sec_acl local interface. Application developers are responsible for writing the server side of this interface using the rdacl APIs. Test server code is included as a sample implementation.

Related Information

Routines

<table>
<thead>
<tr>
<th>ACL API</th>
<th>sec_acl_bind</th>
<th>sec_acl_lookup</th>
</tr>
</thead>
</table>
rdacl_test_access

Tests access to an object.

Format

```c
#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. 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 between 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 one of the following errors:
  - `sec_acl_unknown_manager_type`: Manager type selected is not an available option.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `rdacl_test_access` API 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.

The `rdacl` APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the `rdacl` API for the server portion of the DCE application. The function will be started by client `sec_acl` API calls. In addition, you must compile the `rdaclif.idl` file and link the `rdaclif` server stub generated into your server application.

This network interface is called on the client side through the `sec_acl` local interface. Application developers are responsible for writing the server side of this interface using the `rdacl` APIs. Test server code is included as a sample implementation.

Related Information

Chapter 6. Security and Related Services 6-159
rdacl_test_access

Routines

ACL API rdacl_test_access_on_behalf
Tests access to an object on behalf of another process.

Format

```c
#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. 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 between them. Use `sec_acl_get_manager_types` to acquire a list of the manager types protecting a given object.
- **subject** An Extended Privilege Attribute Certificate (EPAC) for the subject process. The EPAC 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 EPAC 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 caller. 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 one of the following errors:
  - `sec_acl_unknown_manager_type` Manager type selected is not an available option.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.
rdacl_test_access_on_behalf

Usage

The rdacl_test_access_on_behalf API 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.

The rdacl_ APIs are provided for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the rdacl_ API for the server portion of the DCE application. The function will be started by client sec_acl API calls. In addition, you must compile the rdaclif.idl file and link the rdaclif server stub generated into your server application.

This network interface is called on the client side through the sec_acl local interface. Application developers are responsible for writing the server side of this interface using the rdacl APIs. Test server code is included as a sample implementation.

Related Information

Routines

ACL API           rdacl_test_access
SEC_ Routines

DCE Security Services allow application developers to create network services with access to the authentication and authorization capabilities of DCE Security services and facilities.

The calling 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 runs 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 Services API consists of the following sets of remote procedure calls (RPCs) used to communicate with various security-related services and facilities. They include:

**acl**
Implements an access control list protocol for the authorization of a principal to network services.

**epa**
Extracts privilege attributes from an opaque binding handle.

**era**
Maintains extended registry attributes.

**id**
Maps principal names to Universal Unique Identifiers (UUIDs) and vice versa.

**key**
Provides facilities for the maintenance of account keys for daemon principals.

**login**
Validates a principal's network identity and establish delegated identities.

**rgy**
Maintains the network registry of principal identities.

These calls begin with the sec_ prefix, and are the same calls used by various user-level tools provided as part of the DCE. For example, the secrtdb tool is written with sec_rgy calls, dcecp is written with sec_acl calls, and the login program, with which a user logs into a DCE system, is written using sec_login calls. Most sites will find the user-level tools adequate for their needs, and only need to use the Security API to customize or replace the functionality of these tools.

The rdacl_ APIs are provided as reference information for programmers who want to write an ACL manager. If the DCE application server is to contain an ACL manager, write the rdacl manager routine for the server portion of the DCE application. The function will be started by client sec_acl calls. In addition, you must compile the rdaclif.idl file and link the rdaclif server stub generated into your server application.

Although most of the calls in the Security API represent RPC calls, 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.

The following APIs are not supported in z/OS DCE:

- sec_login_get_pwent
- sec_login_init_first
- sec_acl_mgr_configure
- sec_acl_mgr_get_access
- sec_acl_mgr_get_manager_types
- sec_acl_mgr_get_mgr_types_semantics
- sec_acl_mgr_get_printstring
- sec_acl_mgr_is_authorized
- sec_acl_mgr_lookup
- sec_acl_mgr_replace

Data Types

The data types used in the Security Services APIs are listed in this section.
The following data types are used in sec_acl APIs:

**sec_acl_handle_t**
A pointer to an opaque handle bound to an ACL which 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 ACL, 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**, which 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** A pointer to 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** describing 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** describing 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:
- **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 (except **sec_acl_e_type_unauthenticated**).
- **sec_acl_e_type_any_other_deleg** The entry contains the identity of any principal in any cell, acting as a delegate.
- **sec_acl_e_type_extended** The entry contains extended data, formatted as 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).
- **sec_acl_e_type_for_group_deleg** The entry contains the identity of the specific group, identified by global group name, acting as a delegate. This type is described in the POSIX 1003.6 standard.
The entry contains the identity of any principal in the specified cell, acting as a delegate. This type is described in the POSIX 1003.6 standard.

The entry contains a key identifying a user and the foreign realm.

The entry contains a key identifying a group and the foreign realm.

The entry contains a key identifying a foreign realm. Any user that can authenticate to the foreign realm will be allowed access.

The entry contains a key identifying a group. This type is described in the POSIX 1003.6 standard.

The entry contains the identity of the specific group, identified by cell relative group name, acting as a delegate.

The entry contains permissions for the implied group object. This type is described in the POSIX 1003.6 standard.

The entry contains the identity of a group that is listed as the owner of the object, acting as a delegate.

The entry contains the maximum permissions for all entries other than mask_obj, unauthenticated, user_obj, 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.

The entry contains the identity of any principal in the local cell, acting as a delegate.

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.

You should take great care when allowing unauthenticated access to an object. The presence of this mask on an ACL essentially means that anyone can get at least as much access as allowed by the mask.

The entry contains a key identifying a user. This type is described in the POSIX 1003.6 standard.

The entry contains the specific principal, identified by cell relative principal name, acting as a delegate.

The entry contains permissions for the implied user object. This type is described in the POSIX 1003.6 standard.

The entry contains the identity that owns the object, acting as a delegate.

The contents of the tagged union depend on the entry type.

For the following 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_user_deleg
- sec_acl_e_type_group
- sec_acl_e_type_group_deleg
- sec_acl_e_type_foreign_other
- sec_acl_e_type_for_other_deleg

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_for_user_deleg
- sec_acl_e_type_foreign_group
- sec_acl_e_type_for_group_deleg

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 Distributed File Service (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 IPLing 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 changed.
- 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 (which 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 cannot perform the security service requested by an uninterruptible entry. 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 A byte array containing the pickled data.

`sec_acl_type_t` The sec_acl_type_t type differentiates between 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**

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** or the **sec_acl_mgr_get_printstring** functions will query an ACL manager for the print strings of the permissions it supports.

The structure consists of three components:

1. **printstring**
   A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.
2. **helpstring**
   A character string of maximum length **sec_acl_printstring_help_len** containing some text which may be used to describe the specified permission.
3. **permissions**
   A **sec_acl_permset_t** permission set describing the permissions which 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.

**sec_attr**

The following data types are used in **sec_attr** APIs:

The following data types are used in Extended Privilege Attribute calls and in the **sec_login_credential** 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_credential_get_delegate()** calls. This cursor is initialized by the **sec_cred_initialize_cursor()** or **sec_login_credential_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:

<table>
<thead>
<tr>
<th>If entry_type is:</th>
<th>Then tagged union is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_rstr_e_type_any_other</td>
<td>NULL</td>
</tr>
<tr>
<td>sec_rstr_e_type_foreign_other</td>
<td>foreign_id that identifies the foreign cell.</td>
</tr>
<tr>
<td>sec_rstr_e_type_user sec_rstr_e_type_group</td>
<td>id — a sec_id_t that identifies the user or group.</td>
</tr>
<tr>
<td>sec_rstr_e_type_foreign_user sec_rstr_e_type_foreign_group</td>
<td>foreign_id — a sec_id_foreign_t that identifies the foreign user or group.</td>
</tr>
</tbody>
</table>

sec_id_restriction_set_t

A structure 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.
Simple (impersonation) delegation is allowed.

A 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` 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` that contain the UUIDs of each group in the principal's groupset.

A 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` that contain the UUIDs of 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` that contain the UUIDs of each foreign group in the principal's groupset.

An enumerator that can be used to describe the PAC format.

A structure that contains UUIDs for principals, groups, or organizations and an optional printstring name. Since a UUID is a 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.

sec_id
No special data types are defined for sec_id APIs.

sec_key
The following data types are used in sec_key APIs:

sec_passwd_type_t
An enumerated set describing the currently supported key types. Possible values are:

- 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 Data Encryption Standard (DES) encrypted data.

sec_passwd_rec_t
A structure containing either a plaintext password or a pre-encrypted buffer of password data. The sec_passwd_rec_t structure consists of three components:

1. version_number: The version number of the password.
   
   **Note:** The maximum key version number is 255.
2. pepper: A character string combined with the password before an encryption key is derived from the password.
3. 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_key_mgmt_authn_service
A 32-bit unsigned integer whose purpose is to indicate the authentication service in use, because a server may have different keys for different types of authentication service. Possible values of this data type and their meanings are:

- 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).
- rpc_c_authn_dce_secret: DCE shared-secret key authentication.
The following data types are used in sec_login APIs:

**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 in the Kerberos security service 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 set. This set can take on the following two values:
- `sec_login_no_flags`: No special flags are set.
- `sec_login_credentials_private`: Restrict the validated network credentials to the current process. If this flag is not set, you can share credentials with descendents of current process.

**sec_login_auth_src_t**
An enumerated set describing how the login context was authorized. There is one possible value:
- `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_net_info_t**
The structure of network information obtainable from a login context. The EPAC contains UUIDs, and in some cases, string names. The UUIDs can be converted to string names by the `sec_id_gen_name_` call. An expiration date value of 0 means there is no expiration date, and the password, account, or identity is valid indefinitely. The following is the structure:

```c
typedef struct {
  sec_id_pac_t pac;
  unsigned32 acct_expiration_date;
  unsigned32 passwd_expiration_date;
  unsigned32 identity_expiration_date;
} sec_login_net_info_t;
```

**sec_passwd_rec_t**
A structure containing either a plaintext password or a pre-encrypted buffer of password data. The following is the structure:

```c
typedef struct {
  sec_passwd_version_t version_number;
  unsigned char *pepper;
  struct {
    sec_passwd_type_t key_type;
    union {
      unsigned char *plain;
      sec_passwd_des_key_t des_key;
    } tagged_union;
  } key;
} sec_passwd_rec_t;
```

The following are examples of how to use this structure in the `sec_rgy_acct_add` API.

For a non-encrypted key:

```c
sec_passwd_rec_t key_part;
key_part.version_number = 1;
key_part.pepper = NULL;
key_part.key.key_type = sec_passwd_plain;
key_part.key.tagged_union.plain = "mypwd";
```

For an encrypted key:

```c
sec_passwd_rec_t key_part;
key_part.version_number = 1;
key_part.pepper = NULL;
key_part.key.key_type = sec_passwd_des_key_t;
key_part.key.tagged_union.des_key = "/* Des Key data here */";
```
The structure consists of three components:

1. **version_number**
   The version number of the password.

2. **pepper**
   A character string combined with the password before an encryption key is derived from the password.

3. **key**
   Where the key_type can be:
   - **sec_passwd_plain**
     Indicates that a printable string of data is stored in `plain`.
   - **sec_passwd_des_key_t**
     Indicates that an array of data is stored in `des_key`.

### sec_rgy

The following data types are used in sec_rgy APIs:

- **sec_rgy_handle_t**
  A pointer to the registry server handle. The registry server is bound to a handle with `sec_rgy_site_open()`.

- **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:
      - **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, ncaen_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_cdmf_privacy — Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than rpc_c_protect_level_pkt_privacy. This is the second highest protection level, but is available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.

• 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:

• 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:

• 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 indicating seconds (sec) and microseconds (usec) since 0:00, Jan 1, 1980.
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 (principal, 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. On 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 struct 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: principal, group, or organization.

sec_rgy_pgo_flags_t
A 32-bit bit set containing administration flag's used as part of the administrator's information for any registry account. This set contains 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. The registry server account must be in the registry.
- 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.
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` bit set 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.

The portion of a registry account item that is only directly modifiable by an administrator. This structure consists of the following components:

- **creator**: The first four fields are all set by the server. 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. An expiration date value of 0 means there is no expiration date, and the account is valid indefinitely.
- **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 bit set containing flags controlling user-modifiable information. Only one flag is set:

- sec_rgy_acct_user_passwd_valid: If it is set, it indicates that the user password is valid. If not set, this flag forces the user to change the password on the next login attempt.

sec_rgy_acct_user_t

A structure containing registry account information that can be changed by the account owner or other authorized users. It contains seven components:

- 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 bit set 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. An expiration date value of 0 means there is no expiration date, and the password is valid indefinitely.
- 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 to the 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:

- read_version: A 32-bit integer describing the earliest version of the rgymd software that can read this registry.
- write_version: A 32-bit integer describing the version of the rgymd 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 authentication certificate (ticket-granting ticket in Kerberos), 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 bit set, 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 travelling over the network.

    sec_rgy_prop_embedded_unix_id

Indicates that all UUIDs in this registry contain an embedded UNIX number. Because UUIDs are immutable, the UNIX numbers of objects in the registry cannot be changed.

sec_rgy_override_t

A 32-bit integer used as a flag for registry override mode. Its possible values are the constants sec_rgy_no_override and sec_rgy_override. When this mode is enabled, override data specified by the node administrator will replace some of the data from the registry for a given person or account under certain conditions:

1. The registry permits the requested overrides to be set for this machine
2. The override data is intended for person or account at hand.

When the mode is override off, data from the registry is returned to the end user or application unchanged.

sec_rgy_mode_resolve_t

A 32-bit integer used as a flag for resolve mode. Its possible values are the constants sec_rgy_no_resolve_pname and sec_rgy_resolve_pname. When the mode is enabled, path names containing leading // (slashes) will be translated into a form that the local machine’s NFS can understand.

sec_rgy_unix_passwd_buf_t

A character array of UNIX password strings.

sec_rgy_attr

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 an 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 (below) 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_attr_bind_type_string</td>
<td>A pointer to an unsigned 32-bit character string specifying an attribute's RPC string binding.</td>
</tr>
<tr>
<td>sec_attr_bind_type_twrs</td>
<td>An attribute's tower binding representation of type sec_attr_twr_set_t.</td>
</tr>
<tr>
<td>sec_attr_bind_svrname</td>
<td>A pointer to a name of type sec_attr_bind_type_t that specifies a Cell Directory Service entry containing an attribute trigger's binding information.</td>
</tr>
</tbody>
</table>

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.
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:

- **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_cdmf_privacy** — Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument value. This level encrypts all user data in each cell and provides a lower level of packet privacy than **rpc_c_protect_level_pkt_privacy**. This is the second highest protection level, but is available only if one of the User Data Privacy optional features (DES and CDMF, or CDMF only) was installed.
- **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.

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:

- **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**.
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:

- **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).

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 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_i18n_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 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 (below) depend on the value of sec_attr_encoding_t.
- tagged_union: A tagged union whose contents depend on attr_encoding as follows:
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 | ACL Manager Type | Valid Permissions
--- | --- | ---
principal | 06ab9320-0191-11ca-a9e8-08001e039d7d | rcDnfmMaug
organization | 06ab9960-0191-11ca-a9e8-08001e039d7d | rctDnfmM
directory | 06ab9c80-0191-11ca-a9e8-08001e039d7d | rcidDn
replist | 2ac24970-60c3-11cb-b261-08001e039d7d | cidmAI

- **query_permset**: Data of type `sec_acl_permset_t` that defines the permission bits needed to access the attribute's value.
- **update_permset**: Data of type `sec_acl_permset_t` that defines the permission bits needed to update the attribute's value.
- **test_permset**: Data of type `sec_acl_permset_t` that defines the permission bits needed to test the attribute's value.
- **delete_permset**: Data of type `sec_acl_permset_t` that defines the permission bits needed to delete an attribute instance.

sec_attr_acl_mgr_info_p_t

A pointer to a `sec_attr_acl_mgr_info_t` structure.

sec_attr_acl_mgr_info_set_t

A structure that defines an attribute's ACL manager set. The structure consists of the following elements:

- **num_acl_mgrs**: An unsigned 32-bit integer that specifies the number of ACL managers in the ACL manager set.
- **mgr_info[]**: An array of pointers of type `sec_attr_mgr_info_p_t` that define the ACL manager types in the ACL manager set and the permission sets associated with the ACL manager type.
sec_attr_intercell_action_t  An enumerator that specifies the action that should be taken by the Privilege Service when it reads acceptable attributes from a foreign cell. A foreign attribute is acceptable only if there is either a schema entry for the foreign cell or if sec_attr_intercell_act_accept is set to true. This enumerator, which is used in conjunction with the sec_attr_schema_entry_t data type, is composed of the following elements:

sec_attr_intercell_act_accept  If the unique flag in the sec_attr_schema_entry_t data type is not set on, retain the attribute. If the unique flag is set on, retain the attribute only if its value is unique among all attribute instances of the same attribute type within the cell.

sec_attr_intercell_act_reject  Discard the input attribute.

sec_attr_intercell_act_evaluate  Use the binding information in the trig_binding field of this sec_attr_schema_entry_t data type to make a sec_attr_trig_query call to a trigger server. That server determines whether to retain the attribute value, discard the attribute value, or map the attribute to another value(s).

sec_attr_trig_type_t  Specifies the trigger type, a flag that determines whether an attribute trigger should be invoked for query operations. The data type, which is used in conjunction with the sec_attr_schema_entry_t data type, uses the following constants:

sec_attr_trig_type_query  The attribute trigger server is invoked for query operations.

sec_attr_trig_type_query  The attribute trigger server is invoked for update operations.

sec_attr_schema_entry_t  A structure that defines a complete attribute entry for the schema catalog. The entry is identified by both a unique string name and a unique attribute UUID. Although either can either can be used as a retrieval key, the string name should be used for interactive access to the attribute and the UUID for programmatic access. The attribute UUID is used to identify the semantics defined for the attribute type in the schema. The sec_attr_schema_entry_t data type consists of the following elements:

attr_name  A pointer to the attribute name.

attr_id  A value of type uuid_t that identifies the attribute type.

attr_encoding  An enumerator of type sec_attr_encoding_t that specifies the attribute’s encoding.

acl_mgr_set  A structure of type sec_attr_acl_mgr_info_set_t that specifies the ACL manager types that support the objects on which attributes of this type can be created and the permission bits supported by that ACL manager type.

schema_entry_flags  An unsigned integer of type sec_attr_sch_entry_flags_t that defines bitsets for the following flags:

unique  When set on, this flag indicates that each instance of this attribute type must have a unique value within the cell for the object type implied by the ACL Manager type. If this flag is not set on, uniqueness checks are not performed for attribute writes.

multi_valued  When set on, this flag indicates that this attribute type may be multi-valued; 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 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_unique**
  If set, indicates that the attribute 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 non-existent 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.

#### Constants

The constants used in the Security Services APIs are listed in this section.

**sec_acl**

The following constants are used in **sec_acl** APIs:

- **sec_acl_default_handle**: The value of an unbound ACL manager handle.
- **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 0x00000000 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** under **Data Types** in section "sec_acl" on page 6-164.
The following constants are used in sec_attr APIs:

**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** Constants

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** Constants

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** Constants

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 a 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.
**sec_id**

No special constants are defined for use with sec_id APIs.

**sec_key**

No special constants are defined for use with sec_key APIs.

**sec_login**

The following constants are used in sec_login APIs:

- sec_login_default_handle: The value of a login context handle before setup or validation.
- sec_login_no_flags: No special flags are set.
- sec_login_credentials_private: Restrict the validated network credentials to the current process. If this flag is not set, you can share credentials with descendents of current process.

These two constants are used in the sec_login_passwd_t structure for users from remote cells:

- sec_login_remote_uid: Used in the sec_login_remote_uid structure for users from remote cells.
- sec_login_remote_gid: Used in the sec_login_remote_uid structure for users from remote cells.

**sec_rgy**

The following constants are used in sec_rgy APIs:

- 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: The key is not valid.
  - 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 principal, 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 refers to a person.
  - sec_rgy_domain_group: The name refers to a group.
  - sec_rgy_domain_org: The name refers to an organization.

- sec_rgy_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 bit set when none of its flags is set.
- sec_rgy_acct_auth_flags_none: A 32-bit integer. This is the value of the sec_rgy_acct_auth_flags_t bit set when none of its flags is set.
sec_rgy_acct_user_flags_t A 16-bit integer. This is the value of the sec_rgy_acct_user_flags_t bit set when none of its flags is set.

sec_rgy_plcy_pwd_flags_t A 16-bit integer. This is the value of the sec_rgy_plcy_pwd_flags_t bit set when none of its flags is set.

sec_rgy_properties_flags_t A 16-bit integer. This is the value of the sec_rgy_properties_flags_t bit set when none of its flags is set.

sec_rgy_override A 32-bit integer that turns on the registry override mode. When this mode is enabled, override data specified by the node administrator will replace some of the data from the registry for a given person or account under certain conditions.

sec_rgy_no_override A 32-bit integer that turns off the registry override mode.

sec_rgy_resolve_pname A 32-bit integer that turns on the registry resolve mode. When the mode is enabled, path names containing leading // (slashes) will be translated into a form that the local machine's NFS can understand.

sec_rgy_no_resolve_pname A 32-bit integer that turns off the registry resolve mode.
sec_acl_bind

Returns a handle for an object’s ACL.

Format

#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 APIs 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 one of the following errors:
sec_acl_object_not_found The requested object could not be found.
sec_acl_no_acl_found No ACL is associated with the specified object.
sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an
input parameter, or a NULL pointer was supplied as a
required output parameter. Check the parameters
passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always
pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_acl_bind API returns a handle bound to the indicated object’s ACL and sets up an ACL handle for the caller. The
API does not check that the ACL or the object exists, this check is done when the caller uses one of the other sec_acl APIs.

Note: Storage held by the handle should be released by issuing sec_acl_release_handle when it is no longer needed.

This API is central to all the other sec_acl routines, each of which requires this handle to identify the ACL on which to operate.

If the specified name is both an actual object and an entry in the global namespace, two ACLs are 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.
sec_acl_bind_to_addr

Returns a handle to an object identified by its network address.

Format

#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 APIs 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 one of the following errors:

- 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 not valid.
- sec_acl_cant_allocate_memory Storage allocation failure.
- sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an
  input parameter, or a NULL pointer was supplied as a
  required output parameter. Check the parameters
  passed to the function returning this error.

Usage

The sec_acl_bind_to_addr API returns a handle bound to the indicated object’s ACL. This API and the sec_acl_bind API
are central to all the other sec_acl APIs, 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 the component name and to bind
with a nonexistent name. The purpose of this call is to eliminate the need to look up an object’s name. To validate the name,
use sec_acl_bind.

Note: Storage held by the handle should be released by issuing sec_acl_release_handle when it is no longer needed.
sec_acl_calc_mask

Returns the sec_acl_type_mask_obj entry for the specified ACL list.

Format

```c
#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` 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.

- `status` A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

  - **sec_acl_cant_allocate_memory** Storage allocation failure.
  - **sec_rgy_bad_data** Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The sec_acl_calc_mask API 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, through the sec_acl_get_mgr_types_semantics API.
sec_acl_get_access

Lists the access (permission set) that the caller has for an object.

Format

#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 specified. 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 among 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 one of the following errors:

- sec_acl_invalid_acl_handle
  The ACL handle is not valid.

- sec_rgy_bad_data
  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_acl_get_access API 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 APIs 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 API requires at least one permission of any kind on the object for which the access is to be returned.

Related Information

6-192 Application Development Reference Volumes 1 and 2
Routines

sec_acl_test_access  sec_acl_test_access_on_behalf
sec_acl_get_error_info

Returns error information from an ACL handle.

Format

```c
#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` API, which also specifies the name of the object to which the target ACL belongs.

Usage

The `sec_acl_get_error_info` API returns error information from the specified ACL handle.

During a call to a routine in the `sec_acl` 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` API returns the last error code stored in the ACL handle, for those clients who need to know exactly that cause of the error.

Return Values

This routine returns a value of type `error_status_t`, indicating the cause of the last error issued by the RPC runtime.

If the command is not successful, it returns the following error:

```c
sec_acl_invalid_acl_handle The ACL handle is not valid.
```

Related Information

Routines

`sec_acl_bind` `sec_acl_lookup`
sec_acl_get_manager_types

Lists the manager types of the ACLs protecting an object.

Format

```c
#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 between the various types of ACLs an object can possess for a given manager type. Possible values are:
  - `sec_acl_type_object`
  - `sec_acl_type_default_object`
  - `sec_acl_type_default_container`

**Output**

- **size_used**: An unsigned 32-bit integer containing the number of output entries returned in the `manager_types` array. This will be different from `num_types` when the number of manager types is different from the actual size of the `manager_type` 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 not enough space was 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 one of the following errors:
  - `sec_acl_invalid_acl_handle`: The ACL handle is not valid.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_acl_get_manager_types` API 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 to display the permissions supported by a specific manager.
Permissions Required

The sec_acl_get_manager_types API requires at least one permission of any kind on the object for which the ACL manager types are to returned.

Related Information

Routines

sec_acl_bind  sec_acl_get_printstring
sec acl get mgr types semantics

Lists the manager types of the ACLs protecting an object.

Format

#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. Possible values are:
    - 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. This will be different from num_types when the number of manager types is different from the actual size of the manager_type 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 one of the following errors:
    - sec acl invalid acl handle
      The ACL handle is not valid.
    - sec rgy bad data
      Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
sec_acl_get_mgr_types_semantics

Usage

The sec_acl_get_mgr_types_semantics API 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 supporting permissions that allow 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 to display the permissions supported by a specific manager.

Permissions Required

The sec_acl_get_mgr_types_semantics API requires at least one permission of any kind on the object for which the ACL manager types are to be returned.

Related Information

Routines

sec_acl_bind     sec_acl_get_printstring
sec_acl_get_printstring

Returns printable ACL strings.

Format

#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 between 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 print strings are unambiguous and therefore may be concatenated when printed. When TRUE, however, the strings need to be separated when printed or passed.

total_num_printstrings An unsigned 32-bit integer containing the total number of permission print strings 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 print strings 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 print string.
sec_acl_get_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 one of the following errors:

- sec_acl_invalid_acl_handle: The ACL handle is not valid.
- sec_acl_unknown_manager_type: The manager type selected is not among those referred to by the input handle.
- sec_rgy_bad_data: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_acl_get_printstring API returns an array of printable representations (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 print strings, this routine also returns instructions about how to print them. 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. When the tokenize variable is TRUE, it implies the print strings might be of a form like read or write, which must be displayed separated by spaces, colons, or similar characters.

In any list of permission print strings, 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.

Related Information

Routines

sec_acl_bind sec_acl_get_manager_types
sec_acl_lookup

Returns the ACL for an object.

Format

```c
#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. 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 between 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. Possible values are:
    - `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 ACL. 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 one of the following errors:
    - `sec_acl_invalid_acl_handle` The ACL handle is not valid.
    - `sec_acl_unknown_manager_type` Manager type selected is not an available option.
    - `sec_acl_cant_allocate_memory` Requested operation requires more storage than is available.
    - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_acl_lookup` API loads into storage 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.

The storage containing the `sec_acl_t` structure for each ACL is dynamically allocated. Use the `sec_acl_release` API to return each ACL's storage block to the pool when an application is finished with the ACL's.
sec_acl_lookup

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.

Related Information
Routines

sec_acl_bind  sec_acl_test_access  sec_acl_test_access_on_behalf
sec_acl_release

Releases ACL storage.

Format

#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 one of the following errors:

    sec_acl_invalid_acl_handle
        The ACL handle is not valid.

    sec_rgy_bad_data
        Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_acl_release API releases any local storage associated with the ACL object, returning it to the pool. This is strictly a local operation (because the storage in question is local), and has no effect on the remote object or its ACL.

Related Information

Routines

sec_acl_bind  sec_acl_lookup
sec_acl_release_handle

Removes an ACL handle.

Format

```c
#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 the following errors:
  - `sec_acl_invalid_acl_handle` The ACL handle is not valid.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_acl_release_handle` API removes the specified handle. This is strictly a local operation, and has no effect on the remote object or its ACL.

Related Information

Routines

- `sec_acl_bind`
sec_acl_replace

Replaces an ACL.

Format

```
#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. 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 between 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 between the various types of ACLs an object can possess for a given manager type. Possible values are:

  - 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 ACL. 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 one of the following errors:

  - sec_acl_invalid_acl_handle
    The ACL handle is not valid.
  - sec_acl_unknown_manager_type
    Manager type selected is not an available option.
  - sec_rgy_bad_data
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

  If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_acl_replace API replaces the ACL indicated by the input handle with the information in the sec_acl_list parameter.

To change ACLs an editing application must read an entire ACL (using sec_acl_lookup), change it as needed, and replace it using this routine.

Permissions Required

The sec_acl_replace API requires the c (control) permission on the object for which the ACL is to be replaced.
sec_acl_replace

Related Information

Routines

sec_acl_bind  sec_acl_lookup
sec_acl_test_access

Tests access to an object.

Format

```c
#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. 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 between 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 one of the following errors:

  - `sec_acl_invalid_acl_handle` The ACL handle is not valid.
  - `sec_acl_unknown_manager_type` Manager type selected is not an available option.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

  If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_acl_test_access` API determines whether 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` API requires at least one permission of any kind on the object for which the privileges are to be tested.

Return Values

Returns TRUE if the calling application program is authorized to access the target object with the privileges in `desired_permset`; FALSE otherwise.

Related Information
sec_acl_test_access

Routines

sec_acl_bind    sec_acl_test_access_on_behalf
sec_acl_test_access_on_behalf

Test access to an object on behalf of another process.

Format

```c
#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. 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 between them. Use `sec_acl_get_manager_types` to acquire a list of the manager types protecting a given object.

- **subject**: An extended privilege attribute certificate (EPAC) for the subject process. The EPAC 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 EPAC 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`, which indicates that no authentication protocol was used in the remote procedure call and the identity was transmitted from the caller to the callee. If an authentication protocol was used, then the flag is set to TRUE.) If a null EPAC 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 one of the following errors:

  - `sec_acl_invalid_acl_handle`: The ACL handle is not valid.
  - `sec_acl_unknown_manager_type`: Manager type selected is not an available option.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_acl_test_access_on_behalf` API determines whether 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 calling process and the subject identified in the call. An application generally only inquires after the minimum set of privileges needed to accomplish a specific task.
sec_acl_test_access_on_behalf

Permissions Required

The sec_acl_test_access_on_behalf API 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.

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 an unsuccessful completion status code and a return value of FALSE.

Related Information

Routines

sec_acl_bind  sec_acl_test_access
sec_attr_trig_query

Purpose
Reads attributes coded with an attribute trigger type of query.

Format
#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);
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. Possible status codes and their meanings are:

- sec_attr_not_all_avail Some of the requested attributes do not exist on the object.
- sec_attr_unauthorized Not authorized to perform this function.

Usage

The sec_attr_trig_query() routine reads attributes coded with an 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 it is not called directly, this information is provided for users who are writing the attribute trigger servers that will receive sec_attr_trig_query() input and supply its output.

Multi-valued 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 cursor parameter specifies a cursor of type sec_attr_trig_cursor_t that establishes the point in the attribute list at which to start processing the query. Use the sec_attr_trig_cursor_init function to initialize a list cursor. If cursor is uninitialized, the server begins processing the query at the first attribute that satisfies the search criteria. Note that generally, sec_attr_trig_cursor_init function makes a remote call to the specified server. To initialize the cursor without making this remote call, set the sec_attr_trig_cursor_init function valid parameter to 0.

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.

Related Information

Routines

sec_attr_trig_cursor_init sec_attr_trig_update
### Files

<table>
<thead>
<tr>
<th>File Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/include/dce/sec_attr_trig.idl</td>
<td>The IDL file from which <code>dce/sec_attr_trig.h</code> was derived.</td>
</tr>
</tbody>
</table>
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.

Format
#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.

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. Possible status codes are:
database read only
server unavailable
invalid/unsupported attribute type
invalid encoding type
value not unique
site read only
sec_attr_unauthorized Not authorized to perform this function.

Usage

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 information 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 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.

Related Information

Routines

sec_attr_trig_query

Files

/usr/include/dce/sec_attr_trig.idl The idl file from which dce/sec_attr_trig.h was derived.
sec_cred_free_attr_cursor

Purpose
Free the local resources allocated to a sec_attr_cursor_t used by the sec_cred_get_extended_attr() call.

Format
#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.

Usage
The sec_cred_free_attr_cursor() routine frees the resources associated with a cursor of type sec_cred_attr_cursor_t used by the sec_cred_get_extended_attr() call.

Related Information

Routines
sec_cred_get_extended_attr sec_cred_initialize_attr_cursor
sec_cred_free_cursor

Purpose
Release local resources allocated to a sec_cred_cursor_t used by the sec_cred_get_delegate() call.

Format
#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. Possible error codes are:
sec_no_memory Not enough memory to perform the operation.

Usage
The sec_cred_free_cursor() routine releases local resources allocated to a sec_cred_cursor_t used by the
sec_cred_get_delegate() call.

Related Information
Routines
sec_cred_get_delegate sec_cred_initialize_cursor
sec_cred_free_pa_handle

Purpose
Free the local resources allocated to a privilege attribute handle of type sec_cred_pa_handle_t used by the sec_cred_get_initiator() and sec_cred_get_delegate() calls.

Format
#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.

Usage
The sec_cred_free_pa_handle() routine frees the resources associated 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.

Related Information
Routines
sec_cred_get_initiator sec_cred_get_delegate
sec_cred_get_authz_session_info

Purpose
Returns session-specific information that represents an authenticated client's credentials.

Format
#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. Possible status codes and their meanings are:
    sec_cred_s_authz_cannot_comply
        The server is asking for information that the authorization service
        used for the call cannot supply.

Usage

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.

Related Information

Routines
rpc_binding_inq_auth_caller
sec_cred_get_client_princ_name

sec_cred_get_client_princ_name

Purpose
Returns the principal name associated with a credential handle

Format
#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. Possible status codes and their meanings are:

   sec_cred_s_authz_cannot_comply The server is asking for information that the authorization service used for the call cannot supply.

Usage
The sec_cred_get_client_princ_name() routine extracts the principal name associated with the credentials identified by callers_identity.

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.

Related Information

Routines
rpc_binding_inq_auth_caller
sec_cred_get_deleg_restrictions

Purpose
Returns delegate restrictions from a privilege attribute handle

Format
```
#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`. Otherwise, it returns an error. Possible status codes and their meanings are:

- `sec_cred_s_invalid_pa_handle`: Specified privilege attribute handle is invalid.

Usage

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.

Related Information

Routines

- `sec_cred_get_delegate`
- `sec_cred_get_initiator`
sec_cred_get_delegate

Purpose
Returns a handle to the privilege attributes of an intermediary in a delegation chain

Format
#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. Otherwise, it
returns an error. Possible status codes and their meanings are:

- sec_cred_s_invalid_auth_handle
  Specified credential handle is invalid.
- sec_cred_s_invalid_cursor
  Specified credential cursor is invalid.
- sec_cred_s_no_more_entries
  No more entries available.

Usage

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.
Related Information

Routines

- `rpc_binding_inq_auth_caller`
- `sec_cred_initialize_cursor`
- `sec_cred_get_deleg_restrictions`
- `sec_cred_get_delegation_type`
- `sec_cred_get_extended_attrs`
- `sec_cred_get_opt_restrictions`
- `sec_cred_get_req_restrictions`
- `sec_cred_get_tgt_restrictions`
- `sec_cred_get_v1_pac`
- `sec_cred_free_pa_handle`
sec_cred_get_delegation_type

Purpose
Returns the delegation type from a privilege attribute handle

Format
#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. Otherwise, it returns an error. Possible status codes and their meanings are:

sec_cred_s_invalid_pa_handle
Specified privilege attribute handle is invalid.

Usage
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.

Related Information

Routines
sec_cred_get_delegate sec_cred_get_initiator
sec_cred_get_extended_attrs

Purpose
Returns extended attributes from a privilege handle

Format
#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. Otherwise, it returns an error. Possible status codes and their meanings are:

    sec_cred_s_invalid_pa_handle
        Specified privilege attribute handle is invalid.

    sec_cred_s_invalid_cursor
        Specified credential cursor is invalid.

    sec_cred_s_no_more_entries
        No more entries available.

Usage

The sec_cred_get_extended_attrs() routine extracts extended registry attributes 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.

Related Information
sec_cred_get_extended_attrs

Routines

sec_cred_initialize_attr_cursor   sec_cred_get_initiator   sec_cred_get_delegate
sec_cred_get_initiator

Purpose
Returns the privilege attributes of the initiator of a delegation chain

Format
#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. Otherwise, it returns an error. Possible status codes and their meanings are:
          sec_cred_s_invalid_auth_handle
                  Specified credential handle is invalid.

Usage

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.

Related Information

Functions:

Routines

  rpc_binding_inq_auth_caller  sec_cred_get_extended_attrs  sec_cred_get_req_restrictions
  sec_cred_get_deleg_restrictions  sec_cred_get_opt_restrictions  sec_cred_get_tgt_restrictions
  sec_cred_get_delegation_type  sec_cred_get_pa_date  sec_cred_get_v1_pac
sec_cred_get_opt_restrictions

Purpose
Returns optional restrictions from a privilege handle

Format
#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. Otherwise, it returns an error. Possible status codes and their meanings are:

sec_cred_s_invalid_pa_handle
Specified privilege attribute handle is invalid.

Usage

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.

Related Information

Routines

sec_cred_get_delegate       sec_cred_get_initiator
sec_cred_get_pa_data

Purpose
Returns identity information from a privilege attribute handle

Format
#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. Otherwise, it returns an error. Possible status codes and their meanings are:

sec_cred_s_invalid_pa_handle
Specified privilege attribute handle is invalid.

Usage
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 principal’s local 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.

Related Information

Routines

sec_cred_get_delegate sec_cred_get_initiator
**sec_cred_get_req_restrictions**

**Purpose**
Returns required restrictions from a privilege attribute handle

**Format**
```c
#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`. Otherwise, it returns an error. Possible status codes and their meanings are:
  - `sec_cred_s_invalid_pa_handle`:
    Specified privilege attribute handle is invalid.

**Usage**
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.

**Related Information**

**Routines**
- `sec_cred_get_delegate`
- `sec_cred_get_initiator`
sec_cred_get_tgt_restrictions

Purpose
Returns target restrictions from a privilege attribute handle

Format
```
#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`. Otherwise, it returns an error. Possible status codes and their meanings are:
  - `sec_cred_s_invalid_pa_handle`: Specified privilege attribute handle is invalid.

Usage
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.

Related Information

Routines
- `sec_cred_get_delegate`
- `sec_cred_get_initiator`
sec_cred_get_v1_pac

Purpose
Returns pre-1.1 PAC from a privilege attribute handle

Format
```
#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*. Otherwise, it returns an error. Possible status codes and their meanings are:

- **sec_cred_s_invalid_pa_handle**  
  Specified privilege attribute handle is invalid.

Usage

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_pac_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.

Related Information

Routines

*sec_cred_get_delegate*  
*sec_cred_get_initiator*
sec_cred_initialize_attr_cursor

Purpose
Initialize a sec_attr_cursor_t used by the sec_cred_get_extended_attrs() call

Format
#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. Possible status codes and their meanings are:
        sec_no_memory  No memory available.

Usage
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.

Permissions Required
None

Related Information
Routines
    sec_cred_get_extended_attrs  sec_cred_free_attr_cursor()
sec_cred_initialize_cursor

Purpose
Initialize a sec_cred_cursor_t used by the sec_cred_get_delegate() call

Format
#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. Possible status codes and their meanings are:
sec_no_memory No memory available.

Usage
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.

Related Information
Routines
sec_cred_get_delegate sec_cred_free_cursor
sec_cred_is_authenticated

**Purpose**

Returns true if the supplied credentials are authenticated and false if they are not.

**Format**

```c
#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.

**Usage**

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.

**Return Values**

The routine returns `true` if the credentials are authenticated; `false` if they are not.

**Related Information**

**Routines**

`rpc_binding_inq_auth_caller`

**Files**

`/usr/include/dce/sec_cred.idl`  
The idl file from which `dce/sec_cred.h` was derived.
ID Mapping Routines

This section describes in detail the following ID mapping APIs:

- `sec_id_gen_group`
- `sec_id_gen_name`
- `sec_id_parse_group`
- `sec_id_parse_name`

**Note:** z/OS DCE locks the Registry server handle before making any RPC call to the Security daemon. This results in serialization of multithreaded applications, if the applications do not create a Registry server handle in each thread that issues a security Registry call. Consequently, any multithreaded DCE application that makes `sec_id_*` API calls in different threads, should make the appropriate `sec_rgy_bind*` call in each thread, prior to making the `sec_id_*` call.
sec_id_gen_group

Generates a global name from cell and group UUIDs.

Format

```c
#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_bind` to acquire a bound handle.

- `cell_idp`  
  A pointer to the UUID of the home cell of the group whose name is required.

- `group_idp`  
  A pointer to the UUID of the group whose name is required.

**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 one of the following errors:

  - `sec_rgy_bad_handle`  
    The registry context handle is not valid.

  - `sec_id_e_name_too_long`  
    Name is too long for current implementation.

  - `sec_id_e_bad_cell_uuid`  
    Cell UUID is not a valid cell name.

  - `sec_rgy_object_not_found`  
    The registry server could not find the specified group.

  - `sec_rgy_server_unavailable`  
    The DCE Registry Server is unavailable.

  - `sec_rgy_bad_data`  
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_gen_group` API 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 specified.

Related Information
sec_id_gen_group

Routines

ID MAPPING API
sec_id_gen_name
sec_id_parse_group
sec_id_parse_name
sec_id_global_gen_name
sec_id_global_parse_group
sec_id_global_parse_name
sec_id_gen_name

Generates a global name from cell and principal UUIDs.

Format

```c
#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_bind` to acquire a bound handle.

- `cell_idp`  
  A pointer to the UUID of the home cell of the principal whose name is required.

- `princ_idp`  
  A pointer to the UUID of the principal whose name is required.

**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 one of the following errors:

- `sec_rgy_bad_handle`  
  The registry context handle is not valid.

- `sec_id_e_name_too_long`  
  Name is too long for current implementation.

- `sec_id_e_bad_cell_uuid`  
  Cell UUID is not a valid cell name.

- `sec_rgy_object_not_found`  
  The registry server could not find the specified principal.

- `sec_rgy_server_unavailable`  
  The DCE Registry Server is unavailable.

- `sec_rgy_bad_data`  
  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_gen_name` API 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 specified.
### sec_id_gen_name

#### Permissions Required

The `sec_id_gen_name` API requires at least one permission of any kind on the account associated with the input cell and principal UUIDs.

#### Related Information

**Routines**

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_name</th>
<th>sec_id_global_parse_group</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sec_id_gen_group</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sec_id_parse_group</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_id_global_gen_group

Purpose

Generate a global name from cell and group UUIDs.

Format

```c
#include <dce/secidmap.h>
void sec_id_global_gen_group(
    sec_rgy_bind_auth_info_t *auth_info,
    uuid_p_t cell_idp,
    uuid_p_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**

- `auth_info`: Authentication information which will be used to bind to the security server. Specify NULL for this parameter to use the default authentication values when binding to the security server.
- `cell_idp`: UUID of the home cell for the group.
- `group_idp`: UUID of the group.

**Output**

- `global_name`: The global name of the group. Specify NULL for this parameter if the global name is not needed.
- `cell_namep`: The home cell name for the group. Specify NULL for this parameter if the cell name is not needed.
- `group_namep`: The group name relative to the cell. Specify NULL for this parameter if the group name is not needed.
- `status`: The completion status. On successful completion, the completion status will be set to `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_id_e_name_too_long`: Name is too long for current implementation.
  - `sec_id_e_bad_cell_uuid`: Cell UUID is not a valid cell name.
  - `sec_rgy_object_not_found`: The registry server could not find the specified group.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_global_gen_group()` routine generates a global name from a cell UUID and group UUID. For example, given a UUID specifying the cell `/.../world/brazil` and a UUID specifying the group `writers`, the `sec_id_global_gen_group()` routine would return the global name `/.../world/brazil/writers` as well as the cell name `/.../world/brazil` and the group name `writers`.

The `sec_id_global_gen_group()` routine is similar to the `sec_id_gen_group()` routine except that the first parameter is the authentication information for binding to the security server instead of the actual registry context handle.
sec_id_global_gen_group

Related Information

Routines

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_group</th>
<th>sec_id_global_gen_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_parse_name</td>
<td>sec_id_global_parse_name</td>
</tr>
<tr>
<td>sec_id_gen_name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_id_global_gen_name

Purpose

Generate a global name from cell and principal UUIDs.

Format

```c
#include <dce/secidmap.h>
void sec_id_global_gen_name(
    sec_rgy_bind_auth_info_t *auth_info,
    uuid_p_t cell_idp,
    uuid_p_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**

- **auth_info**
  Authentication information which will be used to bind to the security server. Specify NULL for this parameter to use the default authentication values when binding to the security server.

- **cell_idp**
  UUID of the home cell for the principal.

- **princ_idp**
  UUID of the principal.

**Output**

- **global_name**
  The global name of the principal. Specify NULL for this parameter if the global name is not needed.

- **cell_namep**
  The home cell name for the principal. Specify NULL for this parameter if the cell name is not needed.

- **princ_namep**
  The principal name relative to the cell. Specify NULL for this parameter if the principal name is not needed.

- **status**
  The completion status. On successful completion, the completion status will be set to error_status_ok. Otherwise, it returns one of the following errors:

  - **sec_id_e_name_too_long**
    Name is too long for current implementation.

  - **sec_id_e_bad_cell_uuid**
    Cell UUID is not a valid cell name.

  - **sec_rgy_object_not_found**
    The registry server could not find the specified principal.

  - **sec_rgy_server_unavailable**
    The DCE Registry Server is unavailable.

  - **sec_rgy_bad_data**
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The **sec_id_global_gen_name()** routine generates a global name from the cell and principal UUID values. For example, given a UUID specifying the cell /.../world/brazil and a UUID specifying the principal writers/tom, the **sec_id_global_gen_name()** routine would return the global name, /.../world/brazil/writers/tom, as well as the cell name,

.../world/brazil, and the principal name, writers/tom. The **sec_id_global_gen_name()** routine is similar to the **sec_id_gen_name()** routine except that the first parameter is the authentication information for binding to the security server instead of the actual registry context handle.
sec_id_global_gen_name

Related Information

Routines

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_group</th>
<th>sec_id_global_parse_group</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_parse_name</td>
<td>sec_id_global_parse_name</td>
</tr>
<tr>
<td>sec_id_gen_name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_id_global_is_group_member

Purpose
Test if a principal is a member of a group.

Format

```c
#include <dce/secidmap.h>
boolean32 sec_id_global_is_group_member(
    sec_rgy_bind_auth_info_t *auth_info,
    sec_rgy_name_t group,
    sec_rgy_name_t principal,
    error_status_t *status);
```

Parameters

Input
- **auth_info**: Authentication information which will be used to bind to the security server. Specify NULL for this parameter to use the default authentication values when binding to the security server.
- **group**: The global name of the group.
- **principal**: The global name of the principal.

Output
- **status**: A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_id_e_name_too_long`: Name is too long for current implementation.
  - `sec_rgy_object_not_found`: The registry server could not find the specified group.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage
The `sec_id_is_group_member()` routine will return TRUE if the specified principal is a member of the specified group and FALSE otherwise.

The `sec_id_global_is_group_member()` routine is similar to the `sec_id_is_group_member()` routine except that the first parameter is the authentication information for binding to the security server instead of the actual registry context handle.

Return Values
Returns TRUE if principal is member of group, FALSE otherwise.

Related Information
sec_id_global_is_group_member

Routines

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_gen_name</th>
<th>sec_id_global_parse_group</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_is_group_member</td>
<td>sec_id_parse_group</td>
<td>sec_id_global_parse_name</td>
</tr>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_parse_name</td>
<td></td>
</tr>
</tbody>
</table>
sec_id_global_parse_group

Purpose
Translate a global name into group and cell names and UUIDs.

Format

```c
#include <dce/secidmap.h>
void sec_id_global_parse_group (  
    sec_obj_t *auth_info,  
    sec_rgy_name_t global_name,  
    sec_rgy_name_t cell_namep,  
    uuid_p_t cell_idp,  
    sec_rgy_name_t group_namep,  
    uuid_p_t group_idp,  
    error_status_t *status );
```

Parameters

Input

- `auth_info` Authentication information which will be used to bind to the security server. Specify NULL for this parameter to use the default authentication values when binding to the security server.
- `global_name` The global name of the group.

Output

- `cell_namep` The home cell name for the group. Specify NULL for this parameter if the cell name is not needed.
- `cell_idp` The home cell UUID for the group. Specify NULL for this parameter if the cell UUID is not needed.
- `group_namep` The group name relative to the cell. Specify NULL for this parameter if the group name is not needed.
- `group_idp` The group UUID. Specify NULL for this parameter if the group UUID is not needed.
- `status` A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_id_e_name_too_long` Name is too long for current implementation.
  - `sec_id_e_bad_cell_uuid` Cell UUID is not a valid cell name.
  - `sec_rgy_object_not_found` The registry server could not find the specified group.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_global_parse_group()` routine translates a global group name into a cell name and a cell-relative group name. It is similar to the `sec_id_parse_group()` routine except that the first parameter is the authentication information for binding to the security server instead of the actual registry context handle.

The global group name is specified in one of the following formats:

- `./.../cell-name/group-name`
- `./group-name`
sec_id_global_parse_group

- group-name

**Related Information**

**Routines**

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_group</th>
<th>sec_id_global_gen_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_parse_name</td>
<td>sec_id_global_parse_name</td>
</tr>
<tr>
<td>sec_id_gen_name</td>
<td>sec_id_global_gen_group</td>
<td></td>
</tr>
</tbody>
</table>
sec_id_global_parse_name

Purpose

Translate a global name into principal and cell names and UUIDs.

Format

```c
#include <dce/secidmap.h>
void sec_id_global_parse_name (
    sec_rgy_bind_auth_info_t *auth_info, 
    sec_rgy_name_t global_name, 
    sec_rgy_name_t cell_namep, 
    uuid_p_t cell_idp, 
    sec_rgy_name_t princ_namep, 
    uuid_p_t princ_idp, 
    error_status_t *status );
```

Parameters

**Input**

- **auth_info**
  Authentication information which will be used to bind to the security server. Specify NULL for this parameter to use the default authentication values when binding to the security server.

- **global_name**
  The global name of the principal.

**Output**

- **cell_namep**
  The home cell name for the principal in the format /.../cell-name. Specify NULL for this parameter if the cell name is not needed.

- **cell_idp**
  The home cell UUID for the principal. Specify NULL for this parameter if the cell UUID is not needed.

- **princ_namep**
  The principal name relative to the cell. Specify NULL for this parameter if the principal name is not needed.

- **princ_idp**
  The principal UUID. Specify NULL for this parameter if the principal UUID is not needed.

- **status**
  A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - `sec_id_e_name_too_long` Name is too long for current implementation.
  - `sec_id_e_bad_cell_uuid` Cell UUID is not a valid cell name.
  - `sec_rgy_object_not_found` The registry server could not find the specified group.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The **sec_id_global_parse_name()** routine translates a global principal name into a cell name and a cell-relative principal name. It is similar to the **sec_id_parse_name()** routine except that the first parameter is the authentication information for binding to the security server instead of the actual registry context handle.

The global principal name is specified in one of the following formats:

- `../../cell-name/principal-name`
- `../principal-name`

Chapter 6. Security and Related Services 6-249
sec_id_global_parse_name

- principal-name

**Related Information**

**Routines**

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_group</th>
<th>sec_id_global_gen_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_parse_name</td>
<td>sec_id_global_parse_group</td>
</tr>
<tr>
<td>sec_id_gen_name</td>
<td>sec_id_global_gen_group</td>
<td></td>
</tr>
</tbody>
</table>
sec_id_is_group_member

Purpose
Test if a principal is a member of a group.

Format

```c
#include <dce/secidmap.h>
boolean32 sec_id_is_group_member (
    sec_rgy_handle_t context ,
    sec_rgy_name_t group ,
    sec_rgy_name_t principal ,
    error_status_t *status );
```

Parameters

**Input**

- **context**
  An opaque handle bound to a security server. Use the sec_rgy_site_bind() function to acquire a bound handle. Specify NULL for this parameter to use the default binding handle.

- **group**
  The global name of the group.

- **principal**
  The global name of the principal.

**Output**

- **status**
  A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - **sec_rgy_bad_handle**
    The registry context handle is not valid.

  - **sec_id_e_name_too_long**
    Name is too long for current implementation.

  - **sec_id_e_bad_cell_uuid**
    Cell UUID is not a valid cell name.

  - **sec_rgy_object_not_found**
    The registry server could not find the specified group.

  - **sec_rgy_server_unavailable**
    The DCE Registry Server is unavailable.

  - **sec_rgy_bad_data**
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_is_group_member()` routine will return TRUE if the specified principal is a member of the specified group and FALSE otherwise.

Return Values

Returns TRUE if principal is member of group, FALSE otherwise.

Related Information

Chapter 6. Security and Related Services 6-251
sec_id_is_group_member

Routines

ID MAPPING API  |  sec_id_global_is_group_member
**sec_id_parse_group**

Translates a global name into group and cell names and UUIDs.

**Format**

```c
#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_bind` 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 required.

- `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 required.

- `status`  
  A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - `sec_rgy_bad_handle`  
    The registry context handle is not valid.

  - `sec_id_e_name_too_long`  
    Name is too long for current implementation.

  - `sec_id_e_bad_cell_uuid`  
    Cell UUID is not a valid cell name.

  - `sec_rgy_object_not_found`  
    The registry server could not find the specified group.

  - `sec_rgy_server_unavailable`  
    The DCE Registry Server is unavailable.

  - `sec_rgy_bad_data`  
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

**Usage**

The `sec_id_parse_group` API 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 `global_name` parameter is of the form `//.../cellname/pgo_name`. For example, the `global_name` for the group `musical/composers` in the `european` cell would be `//.../european/musical/composers` instead of `//.../european/sec/group/musical/composers`.

The routine will not produce translations to any name for which a NULL pointer has been specified.

**Related Information**
sec_id_parse_group

Routines

<table>
<thead>
<tr>
<th>ID MAPPING API</th>
<th>sec_id_parse_name</th>
<th>sec_id_global_parse_group</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_id_gen_group</td>
<td>sec_id_global_gen_group</td>
<td>sec_id_global_parse_name</td>
</tr>
<tr>
<td>sec_id_gen_name</td>
<td>sec_id_global_gen_name</td>
<td></td>
</tr>
</tbody>
</table>
sec_id_parse_name

Translates a global name into principal and cell names and UUIDs.

Format

```c
#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_bind` to acquire a bound handle.

global_name

The global (full) name of the principal, in `sec_rgy_name_t` form. If the global name is not explicitly global, that is, it does not start with `l...`, or is not explicitly local, that is, it does not start with `l:`; z/OS DCE assumes the name is of an entity in the local cell.

**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 required.

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 required.

status

A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:

- **sec_rgy_bad_handle**  The registry context handle is not valid.
- **sec_id_e_name_too_long**  Name is too long for current implementation.
- **sec_id_e_bad_cell_uuid**  Cell UUID is not a valid cell name.
- **sec_rgy_object_not_found**  The registry server could not find the specified principal.
- **sec_rgy_server_unavailable**  The DCE Registry Server is unavailable.
- **sec_rgy_bad_data**  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_id_parse_name` API 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 `global_name` parameter is of the form `l.../cellname/pgo_name` For example, the `global_name` for the principal `harry` in the `european` cell would be `l.../european/harry` instead of `l.../european/sec/principal/harry`.

The routine will not produce translations to any name for which a NULL pointer has been specified.
sec_id_parse_name

Permissions Required

Only if princ_idp is requested as output does the sec_id_parse_name API 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.

Related Information

Routines

ID MAPPING API
sec_id_gen_group  sec_id_parse_group  sec_id_global_parse_group
sec_id_gen_name  sec_id_global_gen_group  sec_id_global_parse_name
Key Management Routines

This section describes in detail the following key management APIs:

- `sec_key_mgmt_change_key`
- `sec_key_mgmt_delete_key`
- `sec_key_mgmt_delete_key_type`
- `sec_key_mgmt_free_key`
- `sec_key_mgmt_garbage_collect`
- `sec_key_mgmt_gen_rand_key`
- `sec_key_mgmt_get_key`
- `sec_key_mgmt_get_next_key`
- `sec_key_mgmt_get_next_kvno`
- `sec_key_mgmt_initialize_cursor`
- `sec_key_mgmt_manage_key`
- `sec_key_mgmt_release_cursor`
- `sec_key_mgmt_set_key`

Key management operations that take a key data argument expect a pointer to a `sec_passwd_rec_t` structure, and those that take a `keytype` parameter (void*) expect a pointer to a `sec_passwd_type_t`. Key management operations that give a `keydata` parameter as output set the pointer to an array of `sec_passwd_rec_t`. (The array is ended 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 give a `keydata` argument as output set the pointer to an array of `sec_passwd_rec_t`. (The array is ended by an element with key type `sec_passwd_none`.)

Note: For DCE implementations that support Threads, programs with multiple threads that write data to the same file must ensure that each thread controls access to the file to preserve the integrity of the file's content. This does not change in the z/OS implementation of DCE APIs that change the content of a file. To handle this situation, you need to provide mutual exclusion through the use of mutexes when you call DCE functions which modify the same file within multiple threads.

The following key management APIs are affected:

- `sec_key_mgmt_change_key`
- `sec_key_mgmt_delete_key`
- `sec_key_mgmt_delete_key_type`
- `sec_key_mgmt_garbage_collect`
- `sec_key_mgmt_get_next_kvno`
- `sec_key_mgmt_initialize_cursor`
- `sec_key_mgmt_manage_key`
- `sec_key_mgmt_set_key`
sec_key_mgmt_change_key

Changes a principal’s key.

**Format**

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

void sec_key_mgmt_change_key (  
    sec_key_mgmt_authn_service authn_service ,  
    void *arg ,  
    unsigned 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).
    - `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

- **arg**
  This parameter can specify either the local *key file* or a parameter to the *get_key_fn* key acquisition routine of the *rpc_server_register_auth_info* API.
  
  A value of **NULL** specifies that the default key file (*krb5/v5srvtab*) should be used. A key file name specifies that file should be used as the key file. Do not specify FILE or WRFILE as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the *dcecp keytab* or *rgy_edit ktadd* command, or the *sec_key_mgmt_set_key* function.
  
  Any other value specifies a parameter for the *get_key_fn* key acquisition routine. See 
  [“rpc_server_register_auth_info” on page 2-258](#) 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 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 encrypted with the current or previous keys) expire. At that time, all obsolete keys may be *garbage collected*, because no valid tickets encrypted 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 one of the following errors:
    - `sec_key_mgmt_e_key_unavailable` Requested key not present.
    - `sec_key_mgmt_e_authn_invalid` Authentication protocol is not valid.
    - `sec_key_mgmt_e_authn_unavailable` Authentication service unavailable to update network database or to obtain necessary network credentials.
    - `sec_key_mgmt_e_unauthorized` Caller unauthorized to perform operation.
Usage

The **sec_key_mgmt_change_key** API 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 service (or network registry) current. Old keys for the principal are *garbage collected* if appropriate and no tickets encrypted with those keys will remain outstanding on the network.

---

**sec_key_mgmt_e_key_unsupported**  
Key type not supported.

**sec_key_mgmt_e_key_version_ex**  
Key with this version number already exists.

**sec_rgy_object_not_found**  
No principal was found with the given name.

**sec_rgy_server_unavailable**  
The DCE Registry Server is unavailable.

**sec_login_s_no_memory**  
A storage allocation error occurred.

**sec_rgy_bad_data**  
Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

**Note:** Any error condition will leave the key state unchanged.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated *error_status_t* variable where required.
sec_key_mgmt_change_key

Related Information

Routines

<table>
<thead>
<tr>
<th>Key Management API</th>
<th>sec_key_mgmt_gen_rand_key</th>
<th>sec_key_mgmt_set_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_key_mgmt_garbage_collect</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_key_mgmt_delete_key

Deletes a key from the local storage.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_delete_key(
    sec_key_mgmt_authn_service authn_service ,
    void *arg ,
    unsigned 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).
- `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

*arg* This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API.

A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify `FILE` or `WRFILE` as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.

Any other value specifies a parameter for the `get_key_fn` key acquisition routine. See `rpc_server_register_auth_info` on page 2-258 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 one of the following errors:

- `sec_key_mgmt_e_key_unavailable` Requested key not present.
- `sec_key_mgmt_e_authn_invalid` Authentication protocol is not valid.
- `sec_key_mgmt_e_unauthorized` Caller unauthorized to perform operation.
- `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

**Note:** Any error condition will leave the key state unchanged.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.
sec_key_mgmt_delete_key

Usage

The sec_key_mgmt_delete_key API 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, he or she should delete the key immediately with sec_key_mgmt_delete_key. This routine removes the key from the local key storage, invalidating all existing tickets encrypted 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 permitted 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.

Related Information

Routines

Key Management API sec_key_mgmt_delete_key_type sec_key_mgmt_garbage_collect
sec_key_mgmt_delete_key_type

Deletes a key version of a key type from the local key storage.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_delete_key_type(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    unsigned 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).
  - `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

- `arg` This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API. A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify FILE or WRFILE as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.

- `principal_name` A pointer to a character string indicating the name of the principal whose key type is to be deleted.

- `keytype` The pointer to a value of type `sec_passwd_type_t`. The value identifies the data encryption algorithm that is being used (for example, DES).

**Output**

- `status` A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_key_mgmt_e_key_unavailable` Requested key not present.
  - `sec_key_mgmt_e_authn_invalid` Authentication protocol is not valid.
  - `sec_key_mgmt_e_unauthorized` Caller unauthorized to perform operation.
  - `sec_key_mgmt_e_key_unsupported` The key type is not supported.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

**Note:** Any error condition will leave the key state unchanged.
sec_key_mgmt_delete_key_type

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_key_mgmt_delete_key_type API 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, invalidating all extant tickets encoded with the key. If the 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 permitted clients of a service.

Related Information

Routines

Key Management API
sec_key_mgmt_delete_key_type
sec_key_mgmt_delete_key
sec_key_mgmt_change_key
sec_key_mgmt_garbage_collect
sec_key_mgmt_free_key

Frees the storage used by a key value.

Format

```c
#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`. Otherwise, it returns the following error:

- `sec_rgy_bad_data`  
  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_key_mgmt_free_key` API 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.

Related Information

**Routines**

Key Management API  
`sec_key_mgmt_get_key`
sec_key_mgmt_garbage_collect

Deletes obsolete keys.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_garbage_collect(
    sec_key_mgmt_authn_service authn_service ,
    void *arg ,
    unsigned 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).
  - `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

- `arg` This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API.
  A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify `FILE` or `WRFILE` as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.
  Any other value specifies a parameter for the `get_key_fn` key acquisition routine. See \[rpc_server_register_auth_info\] on page 2-258 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 one of the following errors:
  - `sec_key_mgmt_e_authn_invalid` Authentication protocol is not valid.
  - `sec_key_mgmt_e_unauthorized` Caller not authorized to perform 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 storage allocation error occurred.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.
Usage

The `sec_key_mgmt_garbage_collect` API discards any obsolete key information for this principal. An obsolete key is one that can only be used to decrypt tickets that are not valid. As an example, consider a key that was in use on Monday, and was only used to encrypt tickets whose maximum lifetime was one day. If that key was changed at eight o'clock Tuesday morning, it would become obsolete by eight o'clock Wednesday morning; at that time, there can be no valid tickets outstanding that are encrypted with this key.

Related Information

Routines

- Key Management API: `sec_key_mgmt_delete_key`
sec_key_mgmt_gen_rand_key

Generates a new random key of a specified key type.

Format

#include <dce/keymgmt.h>

void sec_key_mgmt_gen_rand_key(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    unsigned 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).
- rpc_c_authn_dce_private DCE private key authentication (an implementation of the Kerberos system).

arg This parameter can specify either the local key file or a parameter to the get_key_fn key acquisition routine of the rpc_server_register_auth_info API.

A value of NULL specifies that the default key file (/krb5/v5srvtab) should be used. A key file name specifies that file should be used as the key file. Do not specify FILE or WRFILE as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the dcecp keytab or rgy_edit ktadd command, or the sec_key_mgmt_set_key function.

Any other value specifies a parameter for the get_key_fn key acquisition routine. See "rpc_server_register_auth_info" on page 2-258 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 one of the following errors:

- 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 Cannot allocate storage.
**sec_key_mgmt_gen_rand_key**

**sec_rgy_bad_data**
Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `sec_key_mgmt_gen_rand_key` API 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.

**Related Information**

**Routines**

- Key Management API
- `sec_key_mgmt_free_key`
- `sec_key_mgmt_set_key`
- `sec_key_mgmt_change_key`
sec_key_mgmt_get_key

Retrieves a key from local storage.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_get_key(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    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).
  - **rpc_c_authn_dce_private**: DCE private key authentication (an implementation of the Kerberos system).
- **arg** This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API. A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify FILE or WRFILE as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function. Any other value specifies a parameter for the `get_key_fn` key acquisition routine. See "rpc_server_register_auth_info" on page 2-258 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` function.
- **status** A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - **sec_key_mgmt_e_key_unavailable**: Requested key not present.
  - **sec_key_mgmt_e_authn_invalid**: Authentication protocol is not valid.
  - **sec_key_mgmt_e_unauthorized**: Caller unauthorized to perform operation.
  - **sec_s_no_memory**: Cannot allocate storage.
sec_key_mgmt_get_key

sec_rgy_bad_data  Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_key_mgmt_get_key API extracts the specified key from the local key store.

Related Information

Routines

Key Management API  sec_key_mgmt_free_key
sec_key_mgmt_get_next_key

Retrieves successive keys from the local key storage.

Format

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

void sec_key_mgmt_get_next_key(
    void *cursor,
    unsigned char **principal_name,
    unsigned32 *key_vno,
    void **keydata,
    error_status_t *status);
```

Parameters

**Input/Output**

cursor

A pointer to the current cursor position in the local key storage. The cursor position is set through 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 used for `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 one of the following errors:

- `sec_key_mgmt_e_unauthorized`: Caller unauthorized to perform operation.
- `sec_key_mgmt_e_key_unavailable`: Requested key not present.
- `sec_s_no_memory`: Cannot allocate storage.
- `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_key_mgmt_get_next_key` API 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.

Related Information
<table>
<thead>
<tr>
<th>Routines</th>
<th>sec_key_mgmt_get_key</th>
<th>sec_key_mgmt_initialize_cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Management API</td>
<td>sec_key_mgmt_get_key</td>
<td></td>
</tr>
<tr>
<td>sec_key_mgmt_free_key</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_key_mgmt_get_next_kvno

Retrieves the next eligible key version number for a key.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_get_next_kvno(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    unsigned 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).
- `rpc_c_authn_dce_private`: DCE private key authentication (an implementation of the Kerberos system).

`arg` This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API.

A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that the file should be used as the key file. Do not specify `FILE` or `WRFILE` as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.

Any other value specifies a parameter for the `get_key_fn` key acquisition routine. See "rpc_server_register_auth_info" on page 2-258 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 one of the following errors:

- `sec_key_mgmt_e_key_unavailable`: Requested key not present.
- `sec_key_mgmt_e_authn_invalid`: Authentication protocol is not valid.
- `sec_key_mgmt_e_unauthorized`: Caller unauthorized to perform operation.
- `sec_rgy_server_unavailable`: No principal was found with the given name.
- `sec_rgy_object_not_found`: No principal was found with the given name.
sec_key_mgmt_get_next_kvno

sec_rgy_bad_data

Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_key_mgmt_get_next_kvno API 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 through its associated authentication protocol, principal name, and key type. The key file in which the key is stored is also specified.

Related Information

Routines

Key Management API
sec_key_mgmt_initialize_cursor

Repositions the cursor in the local key store.

Format

#include <dce/keymgmt.h>

void sec_key_mgmt_initialize_cursor(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    unsigned 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).
- **rpc_c_authn_dce_private**: DCE private key authentication (an implementation of the Kerberos system).

arg

This parameter can specify either the local key file or a parameter to the get_key_fn key acquisition routine of the rpc_server_register_auth_info API.

A value of NULL specifies that the default key file (/krb5/v5srvtab) should be used. A key file name specifies that file should be used as the key file. Do not specify FILE or WFILE as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the dcecp keytab or rgy_edit ktadd command, or the sec_key_mgmt_set_key function.

Any other value specifies a parameter for the get_key_fn key acquisition routine. See "rpc_server_register_auth_info" on page 2-258 for more information.

principal_name

A pointer to a character string indicating the name of the principal whose key is to be changed.

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 function.

status

A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- **sec_s_no_memory**: Cannot allocate storage.
- **sec_key_mgmt_e_authn_invalid**: Authentication protocol is not valid.
- **sec_key_mgmt_e_unauthorized**: Caller not authorized to perform operation.
- **sec_rgy_bad_data**: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Note: If the keytab file is accessible and readable by the caller, sec_key_mgmt_initialize_cursor always returns a cursor to the start of the key table specified. This API does not check whether the principal exists within the key table.
If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `sec_key_mgmt_initialize_cursor` API resets the cursor in the local key store. Use this routine to reposition the cursor before performing a scan of the local store with `sec_key_mgmt_get_next_key`. The returned cursor value is specified as input to `sec_key_mgmt_get_next_key`.

**Related Information**

**Routines**

- Key Management API
- `sec_key_mgmt_get_next_key`
- `sec_key_mgmt_release_cursor`
**sec_key_mgmt_manage_key**

Automatically changes a principal’s key before it expires.

**Format**

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_manage_key(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    unsigned 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).
  - `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

- `arg` This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API.
  - A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify `FILE` or `WRFILE` as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.
  - Any other value specifies a parameter for the `get_key_fn` key acquisition routine. See [`rpc_server_register_auth_info` on page 2-258](#) 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 one of the following errors:
  - `sec_key_mgmt_e_key_unavailable` Requested key not present; cannot use it 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` Authentication protocol is not valid.
  - `sec_key_mgmt_e_unauthorized` Caller not authorized to perform operation.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.
  - `sec_rgy_object_not_found` No principal was found with the given name.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.
Usage

The `sec_key_mgmt_manage_key` API 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 called 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 shortly 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 calls `sec_key_mgmt_garbage_collect` as needed.

Related Information

Routines

Key Management API  
- `sec_key_mgmt_garbage_collect`  
- `sec_key_mgmt_gen_rand_key`
sec_key_mgmt_release_cursor

Releases the storage used by an initialized cursor value.

Format

```c
#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`. Otherwise, it returns one of the following errors:
  - `sec_key_mgmt_e_unauthorized` The caller is not authorized to perform the operation.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_key_mgmt_release_cursor` API 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.

Related Information

Routines

- Key Management API `sec_key_mgmt_initialize_cursor`
sec_key_mgmt_set_key

Inserts a key value into the local storage.

Format

```c
#include <dce/keymgmt.h>

void sec_key_mgmt_set_key(
    sec_key_mgmt_authn_service authn_service,
    void *arg,
    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).
  - `rpc_c_authn_dce_private` DCE private key authentication (an implementation of the Kerberos system).

- `arg` This parameter can specify either the local key file or a parameter to the `get_key_fn` key acquisition routine of the `rpc_server_register_auth_info` API. A value of `NULL` specifies that the default key file (`/krb5/v5srvtab`) should be used. A key file name specifies that file should be used as the key file. Do not specify `FILE` or `WRFILE` as a prefix to the file name because the API automatically selects the appropriate prefix. The file must have been created with the `dcecp keytab` or `rgy_edit ktadd` command, or the `sec_key_mgmt_set_key` function.

- `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. If `key_vno` has a value of zero (0), then the routine returns the error `sec_key_mgmt_e_key_unavailable`.

- `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 one of the following errors:
  - `sec_key_mgmt_e_key_unavailable` Requested key not present. This error is also returned if `key_vno` has a value of zero (0).
  - `sec_key_mgmt_e_authn_invalid` Authentication protocol is not valid.
  - `sec_key_mgmt_e_unauthorized` Caller unauthorized to perform operation.
  - `sec_key_mgmt_e_key_unsupported` Key type not supported.
  - `sec_key_mgmt_e_key_version_ex` Key with this version number already exists.
The `sec_key_mgmt_set_key` API performs all local activities necessary to update a principal’s key to the specified value. This API will not update the authentication service’s value for the principal’s key.

This API does not check if the account exists in the DCE registry. The key is created even if the account does not exist.

In some circumstances, a server may only want 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. Because these servers together represent only one service, they should all share the same key. A user with a ticket to use the database can choose whichever server is least busy. To change the database key, the master server signals 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.

### Related Information

#### Routines

- **Key Management API**
  - `sec_key_mgmt_change_key`
  - `sec_key_mgmt_gen_rand_key`
Login Routines

This section describes in detail the following login APIs:

- sec_login_become_delegate
- sec_login_become_impersonator
- sec_login_become_initiator
- sec_login_certify_identity
- sec_login_context_token_owner
- sec_login_create_context_token
- sec_login_cred_get_delegate
- sec_login_cred_init_cursor
- sec_login_delete_context_token
- sec_login_disable_delegation
- sec_login_expand_context_token
- sec_login_export_context
- sec_login_export_context_data
- sec_login_free_net_info
- sec_login_get_current_context
- sec_login_get_expiration
- sec_login_get_groups
- sec_login_pwent

Note: For DCE implementations that support Threads, programs with multiple threads that write data to the same file must ensure that each thread controls access to the file to preserve the integrity of the file's content. This does not change in the z/OS implementation of DCE APIs that change the content of a file. To handle this situation, you need to provide mutual exclusion through the use of mutexes when you call DCE functions which modify the same file within multiple threads.

The following login APIs are affected:

- sec_login_certify_identity
- sec_login_get_current_context
- sec_login_get_expiration
- sec_login_get_group
- sec_login_purge_context
- sec_login_refresh_identity
- sec_login_set_context
- sec_login_set_extended_attrs
- sec_login_setup_first
- sec_login_valid_and_cert_identity
- sec_login_valid_from_keytable
- sec_login_valid_from_system
- sec_login_validate_first
- sec_login_validate_identity
sec_login_become_delegate

Purpose
Causes an intermediate server to become a delegate in traced delegation chain.

Format
#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.

degression_type_permitted
A value of sec_id_delegation_type_t that specifies the type of delegation to be enabled. The types available are:

- 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:

  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. Possible status codes and their meanings are:

  sec_login_s_context_invalid
    Login context is not valid.

  sec_login_s_compound_delegate
    Login context already specifies a delegation chain.

  sec_login_s_invalid_deleg_type
    Invalid delegation type selection.

  sec_login_s_invalid_compat_mode
    Invalid compatibility mode selection.

  sec_login_s_deleg_not_enabled
    Delegation has not been enabled.

Usage
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 be used 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.

Related Information
sec_login_become_delegate

**Routines**

- sec_login_become_initiator
- sec_login_become_impersonator
- rpc_binding_inq_auth_caller
- sec_login_get_current_context
- sec_login_setup_identity
- sec_login_validate_identity()

**Files**

/usr/include/dce/sec_login.idl

The idl file from which dce/sec_login.h was derived.
sec_login_become_impersonator

Purpose
Causes an intermediate server to become a delegate in a simple delegation chain.

Format
#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_idrestriction_set_t *delegate_restrictions,
    sec_idrestriction_set_t *target_restrictions,
    sec_idopt_req_t *optional_restrictions,
    sec_idopt_req_t *required_restrictions,
    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:

- sec_id_deleg_type_none
  No delegation.

- sec_id_deleg_type_traced
  Traced delegation.

- sec_id_deleg_type_impersonation
  Simple (impersonation) delegation.

  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_idrestriction_set_t that supplies a list of servers that can act as delegates for the client becoming the delegate. These servers are added to the delegates permitted by the delegate_restrictions parameter of the sec_login_become_initiator call.

target_restrictions A pointer to a sec_idrestriction_set_t that supplies a list of servers that can act as targets for the client becoming the delegate. These servers are added to targets specified by the target_restrictions parameter of the sec_login_become_initiator call.
sec_login_become_impersonator

**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 becoming the delegate. 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 client becoming the delegate. These restrictions are added to the restrictions identified `required_restrictions` parameter of the `sec_login_become_initiator` call.

**Output**

**status**
A pointer to the completion status. On successful completion, `status` is assigned `error_status_ok`. Otherwise, it returns an error. Possible status codes and their meanings are:

- `sec_login_s_invalid_deleg_type`
  Invalid delegation type selection.

- `sec_login_s_context_invalid`
  Login context is not valid.

- `sec_login_s_deleg_not_enabled`
  Delegation has not been enabled.

- `sec_login_s_invalid_compat_mode`
  Invalid compatibility mode selection.

**Usage**

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 be used 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 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`.

**Related Information**

**Routines**

- `sec_login_become_initiator`
- `rpc_binding_inq_auth_caller`

**Files**

- `/usr/include/dce/sec_login.idl`
  The idl file from which `dce/sec_login.h` was derived.
sec_login_become_initiator

Purpose
Constructs a new login context that enables delegation for the calling client.

Format
```
#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 authenticated 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:
  - `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:
  - `sec_id_compat_mode_none`: Compatibility mode is off.
sec_login_become_initiator

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

A pointer to the completion status. On successful completion, status is assigned error_status_ok. Otherwise, it returns an error. Possible status codes and their meanings are:

- **sec_login_s_context_invalid**
  Login context is not valid.

- **sec_login_s_invalid_deleg_type**
  Invalid delegation type selection.

- **sec_login_s_invalid_compat_mode**
  Invalid compatibility mode selection.

Usage

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 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).

Related Information

Routines

- sec_login_become_delegate
- sec_login_become_impersonator
- sec_login_get_current_context
- sec_login_setup_identity
- sec_login_validate_identity()

Files

- /usr/include/dce/sec_login.idl
  The idl file from which dce/sec_login.h was derived.
sec_login_certify_identity

Certifies that the network authentication service context is authentic.

Format

```c
#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.

**Output**

- `status` A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_config` The DCE configuration (`dce_config`) information is not available.
  - `sec_login_s_context_invalid` Login context is not valid.
  - `sec_login_s_default_use` Use of the default login handle is not allowed.
  - `sec_login_s_handle_invalid` The login context handle is not valid.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_login_certify_identity` API certifies that the Security Server used to set up and validate a login context is authentic. The authentic Security Server is one that knows the host system’s *secret key*. On some systems, this may be a privileged operation.

Information may be retrieved through `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* (LFS) to attacks by imposter Security Servers returning suspect local process credentials (UUID, group IDs). This operation updates the local registry with the login context credentials if the certification check succeeds.

Return Values

The routine returns a `boolean32` value, TRUE, if the certification was successful; FALSE otherwise.

Examples

Applications wanting to perform a straightforward login could 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_wierd_auth_svc);
    sec_login_set_context(login_context, &st);
    if (auth_src != sec_login_auth_src_network){
      printf("no network credentials");
      exit(0);}
    if (reset_passwd) {
      ... get new password from user, reset registry record ...
    }
    ... any other application specific login valid actions ...
  }
} else {
  sec_login_purge_context(&login_context, &st);
  ... application specific login failure actions ...
}

Related Information

Routines

Login API
sec_login_get_expiration
sec_login_get_groups

sec_login_valid_from_keytable
sec_login_context_token_owner

Purpose

Change ownership of a context token.

Format

```c
#include <dce/sec_login.idl>
void sec_login_set_context_token_owner (  
   sec_bytes_t *context_token ,  
   char *system_userid ,  
   error_status_t *status );
```

Parameters

Input

- `context_token` Specifies the context token returned by `sec_login_create_context_token()`.
- `system_userid` Specifies the MVS user id which will be the new owner of the context token.

Output

- `status` A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_context_invalid` Login context is not valid
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_set_context_token_owner()` routine changes the owner for a context token. The routine must be called by the current context token owner or by a process with uid 0. In addition, the `sec_login_set_context_token_owner()` routine must be called on the same system where the context token was created.

Return Values

None.

Related Information

Routines

- Login API:
  - `sec_login_create_context_token`
  - `sec_login_delete_context_token`
  - `sec_login_expand_context_token`
sec_login_create_context_token

Purpose
Create a context token for a DCE login context.

Format
#include <dce/sec_login.h>
void sec_login_create_context_token (
    sec_login_handle_t login_context,
    sec_bytes_t *context_token,
    error_status_t *status);

Parameters

Input
login_context Specifies the opaque handle for the login context.

Output
context_token Returns the context token. The application is responsible for deleting the context token when it
is no longer needed by calling the sec_login_delete_context_token() routine.

status A pointer to the completion status. On successful completion, the function returns
error_status_ok. Otherwise, it returns one of the following errors:
sec_login_s_handle_invalid Login context handle is not valid
sec_login_s_context_invalid Login context is not valid
sec_rgy_server_unavailable The DCE Registry Server is unavailable.
sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an
input parameter, or a NULL pointer was supplied as a
required output parameter. Check the parameters
passed to the function returning this error.

Usage
The sec_login_create_context_token() routine creates a context token for the specified DCE login context. This token can
then be passed to another process running on the same or a different system in the same sysplex. The
sec_login_expand_context_token() routine can then be used to recreate the DCE login context from the context token. The
application is responsible for calling the sec_login_delete_context_token() routine when the context token is no longer
needed.

The process which creates the context token becomes the context token owner. The context token owner can be changed by
calling the sec_login_set_context_token_owner() routine.

Return Values
None.

Related Information
Routines

Login API

sec_login_create_context_token
sec_login_context_token_owner
sec_login_delete_context_token
sec_login_expand_context_token
sec_login_cred_get_delegate

Purpose
Returns a handle to the privilege attributes of an intermediary in a delegation chain.
Used by clients.

Format
#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. Possible status codes and their meanings are:

sec_cred_s_invalid_cursor
Specified credential cursor is invalid.

sec_cred_s_no_more_entries
No more entries available (informational message).

Usage
The sec_login_cred_get_delegate() routine returns a handle of type sec_login_handle_t to 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() and sec_cred_free_cursor() calls to free the resources allocated to
sec_cred_pa_handle_t and cursor.
Related Information

Routines

- `sec_login_cred_init_cursor`
- `sec_cred_get_pa_date`
- `sec_cred_get_extended_attrs`
- `sec_cred_get_v1_pac`
- `sec_cred_get_tgt_restrictions`
- `sec_cred_get_deleg_restrictions`
- `sec_cred_get_opt_restrictions`
- `sec_cred_get_req_restrictions`
- `sec_cred_get_delegation_type`

Files

- `/usr/include/dce/sec_login.idl`

The idl file from which `dce/sec_login.h` was derived.
sec_login_cred_get_initiator

Purpose
Returns information about the delegation initiator in a specified login context.

Format
#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.

Usage
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.

Error Conditions
sec_login_s_context_invalid    Login context is not valid.
error_status_ok                Successful operation.

Related Information
Routines
sec_cred_get_pa_date
sec_cred_get_extended_attrs
sec_cred_get_v1_pac

sec_cred_get_tgt_restrictions
sec_cred_get_deleg_restrictions
sec_cred_get_opt_restrictions

sec_cred_get_req_restrictions
sec_cred_get_delegation_type

Files
/usr/include/dce/sec_login.idl    The idl file from which dce/sec_login.h was derived.
sec_login_cred_init_cursor

Purpose
Initialize a sec_cred_cursor_t.

Format
#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. Possible status codes and their meanings are:

sec_cred_s_invalid_cursor Specified credential cursor is invalid.

sec_login_s_no_memory No more memory available.

Usage
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.

Related Information
Routines

sec_login_cred_get_delegate
sec_login_delete_context_token

Purpose
Delete a context token created by the sec_login_create_context_token() routine.

Format

```
#include <dce/sec_login.h>
void sec_login_delete_context_token (  
    sec_bytes_t *context_token ,  
    error_status_t *status ) ;
```

Parameters

Input

- `context_token`: Specifies the context token returned by the `sec_login_create_context_token()` routine. The storage allocated to the context token will be released upon successful completion of this routine.

Output

- `status`: A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_context_invalid`: Login context is not valid
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_delete_context_token()` routine will delete a context token returned by the `sec_login_create_context_token()` routine and release the storage associated with the token. The application must not attempt to use the context token after it has been deleted.

A context token may be deleted only on the system where it was created and only by the context owner or by a process with a uid of 0.

Return Values

None.

Related Information

Routines

- `Login API`  
  - `sec_login_create_context_token`  
  - `sec_login_expand_context_token`

- `sec_login_context_token_owner`
sec_login_disable_delegation

Purpose
Disables delegation for a specified login context.

Format
#include <dce/sec_login.h>

sec_logon_handle_t CcNHIXsec_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. Possible status codes and their meanings are:

- sec_login_s_context_invalid
  The login handle is valid, but the login context is not valid.

Usage
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.

Related Information

Routines
sec_login_become_initiator sec_login_become_delegate sec_login_become_impersonator

Files
/usr/include/dce/sec_login.idl
The idl file from which dce/sec_login.h was derived.
sec_login_expand_context_token

sec_login_expand_context_token

Purpose
Expand a context token returned by the sec_login_create_context_token() routine.

Format

```c
#include <dce/sec_login.h>
void sec_login_expand_context_token (  
    sec_bytes_t *context_token,  
    sec_login_handle_t *login_context,  
    error_status_t *status );
```

Parameters

Input

- `context_token`: Specifies the context token returned by the sec_login_create_context_token() routine or a copy of the context token.

Output

- `login_context`: Returns the opaque handle to the login context which was created from the context token. The application is responsible for releasing the login context by calling sec_login_release_context() or sec_login_purge_context() when it is no longer needed.

- `status`: A pointer to the completion status. On successful completion, the function returns error_status_ok. Otherwise, it returns one of the following errors:
  - `sec_login_s_handle_invalid`: Login context handle is not valid
  - `sec_login_s_context_invalid`: Login context is not valid
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.

Usage

The sec_login_expand_context_token() routine will create a new DCE login context from a context token. The application must release the login context when it is no longer needed by calling either sec_login_release_context() or sec_login_purge_context(). Note that purging this login context has no effect on the original login context which was used to create the context token.

Return Values
None.

Related Information
### Chapter 6. Security and Related Services

#### Routines

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_create_context_token</th>
<th>sec_login_delete_context_token</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_context_token_owner</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_login_export_context

Creates an exportable login context.

Format

#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.

buf_len
An unsigned 32-bit integer containing the allocated length (in bytes) of the buffer to contain the
login context.

Output

buf
An idl_byte array that contains the exportable login context on 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 one of the following errors:

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 not valid.

sec_login_s_context_invalid The login handle is valid, but the login context is not valid.

sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an
input parameter, or a NULL pointer was supplied as a
required output parameter. Check the parameters passed to the function returning this error.

Usage

The sec_login_export_context API obtains an exportable version of the login context information. This information may be
passed to another process running on the same machine.

Related Information
Routines

Login API

sec_login_export_context

sec_login_import_context
sec_login_export_context_data

Purpose

Create a portable version of a DCE login context.

Format

#include <dce/sec_login.h>

void sec_login_export_context_data (sec_login_handle_t login_context,
                                   sec_bytes_t *context_data,
                                   error_status_t *status);

Parameters

Input

login_context An opaque handle to the login context.

Output

context_data Returns the exported context data. The num_bytes field will be set to the length of the data and the bytes fields will be set to the address of the data. The application should free the data when it is no longer needed by calling the free() routine.

status A pointer to the completion status. On successful completion, the function returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_login_s_handle_invalid Login context handle is not valid
- sec_login_s_context_invalid Login context is not valid
- sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The sec_login_export_context_data() routine obtains a portable version of the DCE login context. This information may then be passed to another process running on the same system or on a different system. The sec_login_export_context_data() routine differs from the sec_login_export_context() routine in that sec_login_export_context_data() returns the actual context data while sec_login_export_context() returns just a token identifying the data. As a result, the sec_login_export_context() routine can be used to pass a login context only to another process on the same system.

Return Values

None.

Related Information

6-306 Application Development Reference Volumes 1 and 2
Routines

Login API

sec_login_export_context_data

sec_login_import_context_data
sec_login_free_net_info

Frees storage allocated for a principal's network information.

Format

```
#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.

Usage

The `sec_login_free_net_info` API frees any storage allocated for a principal's network information. Network information is returned by a previous successful call to `sec_login_inquire_net_info`.

This routine does not return any completion codes. Make sure that you specify a valid `sec_login_net_info_t` address. The routine simply frees a range of storage beginning at the specified address, without regard to the actual contents of the storage.

Related Information

**Routines**

Login API  
`sec_login_inquire_net_info`
sec_login_get_current_context

Returns a handle to the current login context.

Format

```c
#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.

- `status`: A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_handle_invalid`: The login context handle is not valid.
  - `sec_login_s_no_current_context`: There was no current context to retrieve. For information about how to set up, validate, and use a login context, see "sec_login_setup_identity" on page 6-333.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_get_current_context` API 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.
sec_login_get_current_context

Examples

The following example illustrates use of the `sec_login_get_current_context` API as part of a process to change the groupset:

```c
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...)

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)) {
... application specific error handling ...
}

Related Information

Routines

Login API

sec_login_setup_identity
sec_login_get_expiration

Returns the ticket-granting ticket (TGT) lifetime for an authenticated identity.

Format

```c
#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.

**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. On successful completion, the routine returns *error_status_ok*. Otherwise, it returns one of the following errors:

- **sec_login_s_context_invalid**: The login handle is valid, but the login context is not valid.
- **sec_login_s_default_use**: Use of the default login handle is not allowed.
- **sec_login_s_handle_invalid**: The login context handle is not valid.
- **sec_login_s_not_certified**: The information returned is not certified and may not be accurate.
- **sec_login_no_current_context**: The calling process has no context of its own.
- **sec_rgy_bad_data**: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated *error_status_t* variable where required.

Usage

The **sec_login_get_expiration** API 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 validated contexts.

Examples

Because the authenticated network identity for a process has a finite lifetime, there is a risk they 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

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

Routines

Login API sec_login_get_current_context
**sec_login_get_groups**

Returns the groupset from a login context.

**Format**

```c
#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.

**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. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

- `sec_login_s_context_invalid`  
The login handle is valid, but the login context is not valid.
- `sec_login_s_handle_invalid`  
The login context handle is not valid.
- `sec_login_s_info_not_avail`  
UNIX information is not available.
- `sec_login_s_default_use`  
Use of the default login handle is not allowed.
- `sec_login_s_not_certified`  
The information returned is not certified and may not be accurate.
- `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.
- `sec_rgy_bad_data`  
Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `sec_login_get_groups` API returns the groups contained in the specified 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.

**Examples**

The following example illustrates use of the `sec_login_get_groups` API as part of a process to change the groupset:
sec_login_get_groups

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...

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\n");

... application specific error handling ...
}

Related Information
Routines
Login API sec_rgy_acct_get_projlist
sec_login_import_context

Imports a login context.

Format

```c
#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.
- `status`: A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_context_invalid`: The login handle is valid, but the login context is not valid.
  - `sec_login_s_default_use`: Use of the default login context is not allowed.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_import_context` API imports a context obtained though a call to `sec_login_export_context` performed on the same machine. This routine is generally reserved to the privileged user.

Related Information

**Routines**

- Login API: `sec_login_export_context`
sec_login_import_context_data

Purpose

Create a DCE login context from the portable context created by the sec_login_export_context_data() routine.

Format

#include <dce/sec_login.h>

void sec_login_import_context_data (
    sec_bytes_t *context_data,
    sec_login_handle_t *login_context,
    error_status_t *status);

Parameters

Input

context_data
The portable context data created by the sec_login_export_context_data() routine. The num_bytes field specifies the length of the data and the bytes field specifies the address of the data.

Output

login_context
Returns the opaque handle to the login context. The application should release the login context when it is no longer needed by calling either the sec_login_release_context() routine or the sec_login_purge_context() routine.

status
A pointer to the completion status. On successful completion, the function returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_login_s_context_invalid: Login context is not valid
- sec_rgy_bad_data: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The sec_login_import_context_data() routine will create a DCE login context using the portable context data created by the sec_login_export_context_data() routine. The new login context will not be related to the original login context in any way other than both contexts will identify the same DCE principal, credentials, and delegation chain. One of the login contexts can be purged without affecting the other login context. The application is responsible for releasing a login context when it is no longer needed by calling either the sec_login_release_context() or the sec_login_purge_context() routine.

Return Values

None.

Related Information
Routines

Login API sec_login_export_context_data
sec_login_inquire_net_info

Returns a principal's network information.

Format

```c
#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.

**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:
  ```c
  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. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_default_use` Use of the default context is not allowed.
  - `sec_login_s_not_certified` The operation failed, the authentication service is not authentic.
  - `sec_login_s_context_invalid` The login handle is valid, but the login context is not valid.
  - `sec_login_s_handle_invalid` The login context handle 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.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_inquire_net_info` API returns network information for the principal identified by the specified login context. The network information consists of:

- The Extended Privilege Attribute Certificate (EPAC) that describes the identity and group memberships of the principal as UUIDs. You can obtain the corresponding group names using the `sec_id_gen_group` API.
- 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 "sec_login_get_expiration" on page 6-311.

An expiration date value if 0 means there is no expiration date, and 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.

Related Information

## Routines

- **Login API**
  - sec_login_free_net_info
  - sec_login_get_expiration
sec_login_newgroups

Changes the group list for a login context.

Format

```c
#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

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>login_context</td>
<td>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.</td>
</tr>
<tr>
<td>flags</td>
<td>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. When it is set it 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.</td>
</tr>
<tr>
<td>num_local_groups</td>
<td>An unsigned 32-bit integer containing the number of local group identities to include in the new context.</td>
</tr>
<tr>
<td>local_groups</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>restricted_context</td>
<td>An opaque handle to the login context containing the changed group list.</td>
</tr>
<tr>
<td>status</td>
<td>A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_s_groupset_invalid</td>
<td>The input list of group names is not valid. There may be groups to which the caller does not belong, or the list may contain groups which do not exist.</td>
</tr>
<tr>
<td>sec_login_s_default_use</td>
<td>Use of the default security login handle is not valid.</td>
</tr>
<tr>
<td>sec_login_s_context_invalid</td>
<td>The login handle is valid, but the login context is not valid.</td>
</tr>
<tr>
<td>sec_login_s_handle_invalid</td>
<td>The login context handle is not valid.</td>
</tr>
<tr>
<td>sec_login_s_auth_local</td>
<td>Cannot apply new groups to a local context.</td>
</tr>
<tr>
<td>sec_rgy_bad_data</td>
<td>Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.</td>
</tr>
</tbody>
</table>

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
Usage

The `sec_login_newgroups` API 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.

Currently you can have only groups from the local cell.

Return Values

This routine returns TRUE when the new login context is successfully established; FALSE otherwise.

Examples

The following example illustrates use of the `sec_login_newgroups` API as part of a process to change the groupset:

```c
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... 

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\n");
    ... application specific error handling ...
}
```

Related Information

Routines

Login API: `sec_login_get_groups` `sec_login_purge_context`
sec_login_purge_context

Destroys a login context and frees its storage.

Format

```c
#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. Because a pointer to the handle is submitted, the handle may be reset to NULL on successful completion.

**Output**

- **status**
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_context_invalid` The login handle is valid, but the login context is not valid.
  - `sec_login_s_handle_invalid` The login context handle is not valid.
  - `sec_login_s_default_use` Use of the default login handle in not allowed.
  - `sec_rgy_bad_data` Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_login_purge_context` API frees any storage allocated for the specified login context and destroys the associated network credentials, if any exist.

Applications must be cautious when purging the current context as this destroys network credentials for all processes that share the credentials.

Examples

The following example illustrates use of the `sec_login_purge_context` API 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");
        exit();
        if (reset_passwd) {
            ... get new password from user, reset registry record ...
        }
        ... any other application specific login valid actions ...
    } else {
        sec_login_purge_context(&login_context, &st);
        ... application specific login failure actions ...
    }
} else {
    sec_login_purge_context(&login_context, &st);
    ... application specific login failure actions ...
}

Related Information

Routines

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_setup_identity</th>
<th>sec_login_validate_identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_set_context</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_login_refresh_identity

Changes an authenticated identity for a login context.

**Format**

```c
#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 one of the following errors:
  - `sec_login_s_handle_invalid` - The login context handle is not valid.
  - `sec_login_s_context_invalid` - The login handle is valid, but the login context is not valid.
  - `sec_rgy_object_not_found` - Principal no longer exists.
  - `sec_rgy_server_unavailable` - Network registry not available.
  - `sec_login_s_auth_local` - Operation not valid on local context.
  - `sec_login_s_default_use` - Use of the default login handle is not allowed.
  - `sec_rgy_bad_data` - Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `sec_login_refresh_identity` API 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 addition of the identity as a member of 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.

**Examples**

Because the authenticated network identity for a process has a finite lifetime, there is a risk that the identify will expire during some long network operation, preventing the operation from completing.

For a server application that must run with an authentication network identity because it sometimes acts 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. Using these `sec_login` calls will prevent interruptions in the server’s operation caused by 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

Routines

Login API              sec_validate_identity
sec_login_release_context

Frees storage allocated for a login context.

Format

```c
#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.

- **status**
  
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - `sec_login_s_default_use`  
    The login context handle is not valid.
  
  - `sec_login_s_context_invalid`  
    The login handle is valid, but the login context is not valid.
  
  - `sec_login_s_handle_invalid`  
    The login context handle is not valid.
  
  - `sec_rgy_bad_data`  
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_login_release_context` API frees any storage 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.

Related Information

Routines

- Login API  
  
  `sec_login_purge_context`
sec_login_set_context

Creates network credentials for a login context.

Format

```c
#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.

Output

- `status`  
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - `sec_login_s_context_invalid`  
    The login handle is valid, but the login context is not valid.

  - `sec_login_s_handle_invalid`  
    The login context handle is not valid.

  - `sec_login_s_default_use`  
    Use of the default login handle is not allowed.

  - `sec_login_s_auth_local`  
    The Operation is not valid on local context.

  - `sec_rgy_bad_data`  
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_login_set_context` API 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 because those routines return previously validated contexts.

The `sec_login_set_context` API creates an entry in the `krb5ccname` file.

Note: You cannot perform this function if the context is marked `sec_login_credentials_private`. Such a context can only be used as an explicit context in userspace applications.

Examples

The following example illustrates use of the `sec_login_set_context` API 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");
        exit(0);
        
        if (reset_passwd) {
            ... get new password from user, reset registry record ...
        }
        
        ... any other application specific login valid actions ...
    }

} else {
    sec_login_purge_context(&login_context, &st);
    
    ... application specific login failure actions ...
}

Related Information

Routines

Login API  sec_login_setup_identity  sec_login_validate_identity
sec_login_set_extended_attrs

Purpose
Constructs a new login context that contains extended registry attributes.

Format
#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. Possible status codes and their meanings are:

- sec_login_s_context_invalid Login context is not valid.

Usage
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 chapter titled "The Extended Registry Attribute API" in the DCE Application Development Guide.

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.

Related Information
sec_login_set_extended_attrs

Routines

<table>
<thead>
<tr>
<th>sec_rgy_attr... calls</th>
<th>sec_login_setup_identity</th>
<th>sec_login_set_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_become_impersonator</td>
<td>sec_login_validate_identity</td>
<td></td>
</tr>
</tbody>
</table>

Files

/usr/include/dce/sec_login.idl  The idl file from which dce/sec_login.h was derived.
sec_login_setup_first

Sets up the default network context.

Format

#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.

status

A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_login_s_default_use: Use of the default login handle is not allowed.
- sec_login_s_no_current_context: The calling process has no context of its own.
- sec_login_s_privileged: Called in unprivileged process.
- sec_login_s_config: DCE configuration (dce_config) information not available.
- sec_rgy_object_not_found: Principal does not exist.
- sec_rgy_server_unavailable: Network registry not available.
- sec_rgy_s_no_memory: Storage allocation failure.
- sec_rgy_bad_data: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The sec_login_setup_first API sets up a login handle which points to a newly-created machine context credentials cache file for the host that it is run on.

The error message sec_login_s_privileged is returned if this API is called by a non-privileged user.

Note to Users

If you are unfamiliar with DCE internals, you should not use this API.

The sec_login_setup_first API 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 dced process to start the default credentials for the host system process hierarchy.

Because this routine uses the host name available through the local dce_config interface as the principal name for the setup, it does not need a principal name as input.
sec_login_setup_first

Return Values
The routine returns a boolean32 value: TRUE if the setup was successful; FALSE otherwise.

Related Information
Routines
Login API sec_login_validate_first
sec_login_setup_identity

Sets up the user's network identity.

Format

```c
#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.

- **status**: A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `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 storage available to complete the operation.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

   If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_login_setup_identity` API 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 by `sec_login_validate_identity`.

This routine retrieves sealed network credentials (or certificates) that are unsealed during the operation of validating an identity. In a Kerberos environment, this operation acquires an appropriate ticket-granting ticket.

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. Because the operations are independent, the user’s password need not be sent across the network.

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.
sec_login_setup_identity

Return Values

The routine returns TRUE if the identity has been successfully established; FALSE otherwise.

Examples

The following example illustrates use of the sec_login_setup_identity API as part of a straightforward login process.

```c
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");
            exit(0);
        }
        if (reset_passwd) {
            ... get new password from user, reset registry record ...
        }
        ... any other application specific login valid actions ...
    }
} else {
    sec_login_purge_context(&login_context, &st);
    ... application specific login failure actions ...
}
```

Related Information

Routines

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_valid_from_keytable</th>
<th>sec_login_valid_from_system</th>
<th>sec_login_validate_identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_set_context</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_login_valid_and_cert_ident

Validates and certifies a login context.

Format

```
#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.

- `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.

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. On 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.

- `status`: A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_handle_invalid`: Use of the default login handle is not allowed, or the login context handle is not valid.
  - `sec_rgy_passwd_invalid`: The input string does not match the account password.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.
  - `sec_rgy_s_acct_invalid`: The account is not valid or has expired.
  - `sec_login_s_privileged`: This is a privileged operation and was called by an unprivileged process.
  - `sec_login_s_null_password`: The input string is NULL.
  - `sec_login_s_default_use`: Use of the default login handle is not allowed.
  - `sec_login_s_already_valid`: Login context is validated.
  - `sec_login_s_unsupp_passwd_type`: The password type is not supported.
  - `sec_rgy_bad_data`: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.
sec_login_valid_and_cert_ident

Usage

The `sec_login_valid_and_cert_ident` routine validates and certifies a login context established by `sec_login_setup_identity`. The caller must know the user's password for this operation to succeed.

This routine combines the operations of the `sec_login_validate_identity` and `sec_login_certify_identity` APIs. 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.

Because this routine is a privileged operation, the caller must be in a privileged process when calling the routine. The caller is in a privileged process when logged in as the root.

Return Values

The routine returns TRUE if the login identity has been successfully validated; FALSE otherwise.
Examples

The following example illustrates use of the sec_login_valid_and_cert_ident API as part of a system login process:

```c
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

Routines

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_setup_identity</th>
<th>sec_login_validate_identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_certify_identity</td>
<td>sec_login_valid_from_keytable</td>
<td></td>
</tr>
</tbody>
</table>
sec_login_valid_from_keytable

Purpose
Validates a login context's identity using input from a specified keytable file.

Format
#include <dce/sec_login.h>

void sec_login_valid_from_keytable(
    sec_login_handle_t login_context,
    sec_key_mgmt_authn_service 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 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 file name
specifies using that file as the keytab file. Do not specify FILE or WRFILE as a prefix to the file
name because the API automatically selects the appropriate prefix. 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
rpc_server_register_auth_info() 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 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 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. Possible status codes and their meanings are:

- **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**
  Use of the default login handle is not allowed.

- **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**
  No enough memory is available to complete the operation.

**Usage**

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 tries 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.

**Examples**

The following example illustrates use of the `sec_login_valid_from_keytable()` routine as part of a straightforward login process:
sec_login_valid_from_keytable

if (sec_login_setup_identity(user_name,
    sec_login_no_flags, &login_context, &st)) {
    /*get password from local keytable*/
    sec_login_valid_from_keytable(login_context,
        authn_service, arg, try_kvno, &used_kvno, &reset_passwd,
        &auth_src, &st);
    if (st == error_status_ok) {
        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

Routines

sec_login_validate_identity        sec_login_setup_identity  sec_login_valid_from_system
sec_login_certify_identity        sec_login_valid_and_cert_identity

Files

/usr/include/dce/sec_login.idl  The idl file from which dce/sec_login.h was derived.
sec_login_valid_from_system

**Purpose**

Validate a login context's identity using a userid which has been authenticated by the system.

**Format**

```c
#include <dce/sec_login.h>

void sec_login_valid_from_system (
    sec_login_handle_t login_context,
    boolean32 reset_passwd,
    sec_login_auth_src_t auth_src,
    error_status_t *status);
```

**Parameters**

**Input**

- `login_context` An opaque handle to the login context.

**Output**

- `reset_passwd` Returns TRUE if the account password has expired and should be reset, otherwise returns FALSE.
- `auth_src` Returns the authorization source. The return value will always be `sec_login_auth_src_network` for the current implementation of this function.
- `status` A pointer to the completion status. On successful completion, the function returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_login_s_handle_invalid` Login context handle is not valid.
  - `sec_login_s_context_invalid` Login context is not valid.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.
  - `sec_login_s_acct_invalid` Account is not valid or has expired.
  - `sec_login_s_default_use` Use of the default login handle is not allowed.
  - `sec_login_s_already_valid` Login context is validated.

**Usage**

The `sec_login_valid_from_system()` routine validates the login context established by the `sec_login_setup_identity()` routine. The authenticated system identity is used to determine the associated DCE principal. This must be the same principal as was specified when the login context was established. For z/OS, the system identity is obtained from the ACEE (Accessor Environment Element) associated with the current thread or process. Typically, an ACEE is created by the system login processor or the batch job initiator.

Refer to the 'RACF Interoperability and Single Signon' section of the [z/OS DCE Administration Guide](https://www.ibm.com/support/docview.wss?uid=swg21387974) for more information on how to establish the connection between the RACF user and the DCE principal. When using the `sec_login_valid_from_system()` routine, the DCE password does not need to be stored in the RACF database and DCE autologin does not need to be enabled.

There must be a DCE security server running on the local system in order for an application to use the `sec_login_valid_from_system()` routine. The completion status will be set to `sec_rgy_server_unavailable` if the local security server is not available. In addition, the current RACF user must have a DCE segment, otherwise the completion status will be set to `sec_rgy_object_not_found`.

**Return Values**

None.
sec_login_valid_from_system

Related Information

Routines

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_setup_identity</th>
<th>sec_login_valid_from_keytable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_validate_identity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
sec_login_validate_first

Validates the initial login context.

Format

```c
#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 is that of the host system initial process.

Output

- **reset_passwd**
  A pointer to a 32-bit boolean 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. On 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.

- **status**
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

  - `sec_login_s_handle_invalid`
    The context handle is not valid.
  - `sec_login_s_privileged`
    Called in unprivileged process.
  - `sec_rgy_server_unavailable`
    The network authentication service was unavailable.
  - `sec_rgy_bad_data`
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Usage

The `sec_login_validate_first` API validates the default login context established by `sec_login_setup_first`. Typically, this operation is called from the `sec_clientd` helper process to validate the default credentials for the host system process hierarchy. Because this operation uses the password for the local host, it does not require a password parameter.

Return Values

The routine returns a `boolean32` value: TRUE if the setup was successful; FALSE otherwise.

Related Information
sec_login_validate_first

Routines

Login API  sec_login_setup_first
sec_login_validate_identity

Validates a login context's identity.

Format

#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.

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.

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 login context was authorized. The sec_login_auth_src_t data type distinguishes the
various ways the login context was authorized. There is one possible value:

sec_login_auth_src_network

status A pointer to the completion status. On successful completion, the routine returns
error_status_ok. Otherwise, it returns one of the following errors:

sec_rgy_password_invalid Input string does not match the account password.
sec_rgy_server_unavailable No data with which to compare the input string.
sec_login_s_acct_invalid Account is not valid or has expired.
sec_login_s_null_password Input string is NULL.
sec_login_s_default_use Use of the default login handle is not allowed.
sec_login_s_handle_invalid The login context handle is not valid.
sec_login_s_already_valid Login context is validated.
sec_login_s_unsupp_passwd_type Password type is not supported.
sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an
input parameter, or a NULL pointer was supplied as a
required output parameter. Check the parameters
passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always
pass a pointer to an allocated error_status_t variable where required.
sec_login_validate_identity

Usage

The **sec_login_validate_identity** API validates the login context established with **sec_login_setup_identity**. The caller must know the user’s password for this operation to succeed. This operation must be called before the network credentials can be used.

When a network identity is set, only state information for network operations has been established. The local operating system identity has not been changed. The caller must 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. Because the operations are independent, the user’s password need not be sent across the network. The identity validation performed by **sec_login_validate_identity** is a local operation.

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 5 minutes of each other this call can return a password validation error.

After the **sec_login_validate_identity** and **sec_login_valid_and_cert_ident** APIs, the password in the *passwd* record is overwritten to minimize the duration of the password in storage.

Return Values

The routine returns TRUE if the login identity has been successfully validated; FALSE otherwise.
Examples

The following example illustrates use of the `sec_login_validate_identity` API as part of a straightforward login process.

```c
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");
            exit(0);
        } else {
            sec_login_purge_context(&login_context, &st);
            printf("application specific login failure actions ...
        } else {
            exit(error_wierd_auth_svc);
        }
    }
}
```

Related Information

**Routines**

<table>
<thead>
<tr>
<th>Login API</th>
<th>sec_login_setup_identity</th>
<th>sec_login_valid_from_keytable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_login_certify_identity</td>
<td>sec_login_valid_and_cert_ident</td>
<td>sec_login_valid_from_system</td>
</tr>
</tbody>
</table>
sec_pwd_mgmt_free_handle

Purpose
Frees storage allocated for a password management handle.

Format
#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. Possible status codes and their meanings are:

- sec_rgy-era_pwd_mgmt_auth_type: The pwd_mgmt_binding ERA must contain authentication
  information.
- sec_pwd_mgmt_srv_unavail: The password management server is unavailable.
- sec_pwd_mgmt_srv_error: Generic error returned from password management server.
  An administrator should check the password management server’s log file for more information.

Usage
The sec_pwd_mgmt_free_handle() routine frees any memory allocated for the contents of a password management handle.

Related Information
Functions
sec_intro
sec_pwd_mgmt_setup

Files
/usr/include/dce/sec_pwd_mgmt.idl The idl file from which dce/sec_pwd_mgmt.h was derived.
sec_pwd_mgmt_gen_pwd

**Purpose**
Generate a set of passwords.

**Format**
```c
#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. One possible status code is:
    ```c
    sec_rgy Era_pwd_mgmt_auth_type
    ```
    The pwd_mgmt_binding ERA must contain authentication information.

- Various RPC communication errors can be returned if there are failures when binding to the password management server.

**Usage**
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.

**Related Information**

- Functions:
  - sec_intro, sec_pwd_mgmt_setup, pwd_strengthd

- Files
  - `/usr/include/dce/sec_pwd_mgmt.idl` The idl file from which `dce/sec_pwd_mgmt.h` was derived.
**sec_pwd_mgmt_get_val_type**

**Purpose**
Gets user's password validation type.

**Format**
```
#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:
  - 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.

Various RPC communication errors can be returned if there are failures when binding to the password management server.

**Usage**
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*.

**Related Information**

**Functions**
- sec_intro
- sec_pwd_mgmt_setup

**Files**
- /usr/include/dce/sec_pwd_mgmt.idl: The idl file from which *dce/sec_pwd_mgmt.h* was derived.
sec_pwd_mgmt_setup

Purpose
Sets up the user's password policy information.

Format

```c
#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. Possible status codes and their meanings are:

- **sec_s_no_memory** Not enough memory is available to complete the operation.
- **sec_rgy_server_unavailable** The network registry is not available.

Usage

The **sec_pwd_mgmt_setup()** routine collects the data required to perform remote password management calls to the password management server.

Related Information

Functions:

- sec_intro,
- sec_pwd_mgmt_free_handle,
- sec_pwd_mgmt_gen_pwd,
- sec_pwd_mgmt_get_val_type,
- pwd_strengthd
sec_pwd_mgmt_setup

Files
/usr/include/dce/sec_pwd_mgmt.idl The idl file from which dce/sec_pwd_mgmt.h was derived.
Registry Routines

This section describes the following registry APIs in detail:

- `sec_rgy_acct_add`
- `sec_rgy_acct_admin_replace`
- `sec_rgy_acct_delete`
- `sec_rgy_acct_get_projlist`
- `sec_rgy_acct_passwd`
- `sec_rgy_acct_rename`
- `sec_rgy_acct_replace_all`
- `sec_rgy_acct_user_replace`
- `sec_rgy_auth_plcy_get_effective`
- `sec_rgy_auth_plcy_get_info`
- `sec_rgy_auth_plcy_set_info`
- `sec_rgy_cell_bind`
- `sec_rgy_cursor_reset`
- `sec_rgy_login_get_effective`
- `sec_rgy_login_get_info`
- `sec_rgy_pgo_add`
- `sec_rgy_pgo_add_member`
- `sec_rgy_pgo_delete`
- `sec_rgy_pgo_delete_member`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_get_members`
- `sec_rgy_pgo_get_next`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_is_member`
- `sec_rgy_pgo_name_to_id`
- `sec_rgy_pgo_name_to_unix_num`
- `sec_rgy_pgo_rename`
- `sec_rgy_pgo_replace`
- `sec_rgy_pgo_unix_num_to_id`
- `sec_rgy_pgo_unix_num_to_name`
- `sec_rgy_plcy_get_effective`
- `sec_rgy_plcy_get_info`
- `sec_rgy_plcy_set_info`
- `sec_rgy_properties_get_info`
- `sec_rgy_properties_set_info`
- `sec_rgy_site_bind`
- `sec_rgy_site_bind_query`
- `sec_rgy_site_bind_update`
- `sec_rgy_site_binding_get_info`
- `sec_rgy_site_close`
- `sec_rgy_site_get`
- `sec_rgy_site_is_readonly`
- `sec_rgy_site_open`
- `sec_rgy_site_open_query`
- `sec_rgy_site_open_update`
- `sec_rgy_unix_getgrgid`
- `sec_rgy_unix_getgrnam`
- `sec_rgy_unix_getpwnam`
- `sec_rgy_unix_getpwuid`
- `sec_rgy_wait_until_consistent`

**Note:** z/OS DCE locks the Registry server handle before making any RPC call to the Security daemon. This results in serialization of multithreaded applications, if the applications do not create a Registry server handle in each thread that issues a security Registry call. Consequently, any multithreaded DCE application that makes `sec_rgy_*` API calls in different threads, should make the appropriate `sec_rgy_bind*` call in each thread, prior to to making the `sec_rgy_*` call.
sec_rgy_acct_add

sec_rgy_acct_add

Purpose
Adds an account for a login name.

Format
#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 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. No abbreviations are allowed, and the only acceptable 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 changed by, the account owner. 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 on login for the account.
  shell The default shell to use on login.
  passwd_version_number The password version number, a 32-bit 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.
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.
caller_key A key to use 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.
The type of the new key. The server uses this parameter to decide how to encode new_key if it is sent as plain text.

**Output**

The key version number returned by the server. If the client requests a particular key version number (through the version_number field of the new_key input parameter), the server returns the requested version number to the client.

A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- **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**
  - No account will be added by this API because one already exists.

- **sec_rgy_server_unavailable**
  - The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

**Usage**

The sec_rgy_acct_add API adds an account with a specified login name. The login name is given in three parts, corresponding to the principal, group, and organization names for the account.

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. No abbreviations are allowed.

The constituent principal, group, and organization items for an account must be added before the account can be created. Also, the principal must have been added as a member of the specified group and organization. For information on how to add a person to a group or organization, see sec_rgy_pgo_add_member on page 6-430.

**Required Permissions**

The sec_rgy_acct_add API requires the following permissions on the account (principal) that is to be added:

- m (mgmt_info) permission to change management information
- a (auth_info) permission to change authentication information
- u (user_info) permission to change user information

**Related Information**

**Routines**

- sec_rgy_acct_delete
- sec_rgy_login_get_info
- sec_rgy_pgo_add
- sec_rgy_pgo_add_member
- sec_rgy_site_open
Purpose
Replaces administrative account data.

Format
#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 wild cards, matching any entry. The principal name must be entered.

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. No abbreviations are allowed, and the only acceptable 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, and can be changed 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 changed 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 changed (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. An expiration date value of 0 means there is no expiration date, and the account is valid indefinitely.

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 Registry API under “sec_rgy” on page 6-187 for a complete description of these flags.
**sec_rgy_acct_admin_replace**

**authentication_flags**
Contains flags controlling use of authentication services. This field is in `sec_rgy_acct_auth_flags_t`. See Registry API under "sec_rgy" on page 6-187 for a complete description of these flags.

**Output**

**status**
A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

- **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.

- **sec_rgy_bad_data**
Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

**Usage**

The `sec_rgy_acct_admin_replace` API 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 changed 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`.

**Required Permissions**

The `sec_rgy_acct_admin_replace` API requires the following permissions on the account principal:

- m (mgmt_info), if flags or expiration date are to be changed
- a (auth_info), if authentication flags or good-since-date are to be changed.

All users need the write (w) privilege in the appropriate ACL entry to change any account information.

**Related Information**

**Routines**

- `sec_rgy_acct_user_replace`
- `sec_rgy_acct_replace_all`
- `sec_rgy_acct_lookup`
sec_rgy_acct_delete

Purpose
Deletes an account.

Format

```c
#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 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 one of the following errors:

  - **sec_rgy_notAuthorized**
    The client program is not authorized to delete the specified account.
  - **sec_rgy_objectNotFound**
    The account to be deleted does not exist.
  - **sec_rgy_serverUnavailable**
    The DCE Registry Server is unavailable.
  - **sec_rgy_badData**
    Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

Note: If the principal, group, and organization are defined in `login_name`, but the account does not exist in the registry, this API returns `error_status_ok` indicating that the routine completed successfully.

Usage

The `sec_rgy_acct_delete` API deletes from the registry the account corresponding to the specified login name.

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`.

Required Permissions

The `sec_rgy_acct_delete` API requires the following permissions on the account principal:

- **m** (mgmt_info) permission to remove management information
- **a** (auth_info) permission to remove authentication information
- **u** (user_info) permission to remove user information.
Related Information

Routines

sec_rgy_acct_add

sec_rgy_pgo_delete
sec_rgy_acct_get_projlist

Purpose
Returns the projects in an account's project list.

Format
#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 wild cards matching any entry. The principal name must be entered.

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
A pointer indicating a project in an account's project list. The sec_rgy_acct_get_projlist API returns the project indicated by projlist_cursor, and advances the cursor to point to the next project on 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 specified 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 one of the following errors:

sec_rgy_no_more_entries
The cursor is at the end of the list of projects.
Usage

The sec_rgy_acct_get_projlist API 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 API updates this variable to point to the next project in the project list. To return an entire project list, reset projlist_cursor on the initial call and then issue successive calls until all the projects are returned.

Several different types of cursors are used in the registry API. Some cursors point to PGO item, some point to members in a membership list, and some 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.

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.

Required Permissions

The sec_rgy_acct_get_projlist API requires the r (read) permission on the account principal for which the project list data is to be returned.

Related Information

Routines

sec_rgy_cursor_reset          sec_rgy_pgo_get_next
sec_rgy_acct_lookup

sec_rgy_acct_lookup

Purpose
Returns data for a specified account.

Format

```c
#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 names corresponding to the account. Blank strings serve as wild cards, 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` was 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. No abbreviations are allowed, and the only acceptable 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 changed by, the account owner.

admin_part
A pointer to the `sec_rgy_acct_admin_t` structure containing the administrative part of the account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, and can be changed only by an administrator.
status       A pointer to the completion status. On successful completion, the routine returns
error_status_ok. Otherwise, it returns one of the following errors:

- **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.
- **sec_rgy_bad_data** Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

**Usage**

The **sec_rgy_acct_lookup** API 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 placeholder in the registry database. The routine 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 was reset with sec_rgy_cursor_reset, sec_rgy_acct_lookup returns information about the first account in the database.

Several different types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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.

Cursors are specific to a server. A cursor pointing into one replica of the registry database are 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.

**Related Information**

**Required Permissions**

The **sec_rgy_acct_lookup** API requires the r (read) permission on the account principal to be viewed.

**Routines**

- sec_rgy_cursor_reset
- sec_rgy_acct_replace_all
- sec_rgy_acct_admin_replace
- sec_rgy_acct_user_replace
sec_rgy_acct_passwd

sec_rgy_acct_passwd

Purpose
Changes the password for an account.

Format

#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 names corresponding to the account. Only the principal name is required to change the password.

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, this parameter is not necessary, and the packet encryption is sufficient to ensure security.

Output
new_key_version
The key version number, returned by the server. If the client requests a particular key version number (through the version_number field of the new_key input parameter), the server returns the requested version number to the client.

Note: The maximum key version number is 255.

status
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- **sec_rgy_notAuthorized**: The client program is not authorized to change the password of this account.
- **sec_rgy_object_not_found**: The account to be changed was not found by the registry server.
- **sec_rgy_server_unavailable**: The DCE Registry Server is unavailable.
- **sec_rgy_bad_data**: Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
Usage

The sec_rgy_acct_passwd API 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.

sec_rgy_acct_passwd does not check the policy of the registry as to the password length, lifetime, expiration date or limitations on spelling prior to adding the account. Before using this command, call the registry policy APIs to set and check the registry policy, and to perform any required checking on the user and administration parts of the data passed in.

Required Permissions

The sec_rgy_acct_passwd API requires the u (user_info) permission on the account principal whose password is to be changed.

Related Information

Routines

Registry API
sec_rgy_acct_rename

sec_rgy_acct_rename

Purpose
Changes an account login name.

Format
#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 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 entered.

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. No abbreviations are allowed, and the only acceptable 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 one of the following errors:

- sec_rgy_not_authorized The client program is not authorized to make the changes.
- sec_rgy_object_not_found The account to be changed 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.
- sec_rgy_bad_data Incorrect data or a NULL pointer was supplied as an input parameter, or a NULL pointer was supplied as a required output parameter. Check the parameters passed to the function returning this error.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
Usage

The `sec_rgy_acct_rename` API 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. 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`.

The `sec_rgy_acct_rename` API 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. Also, the principal must have been added as a member of the specified group and organization. See "sec_rgy_pgo_add_member" on page 6-430 for information on how to add a person to a group or organization.

Required Permissions

The `sec_rgy_acct_rename` API requires the `m` (mgmt_info) permission on the account principal to be renamed.

Related Information

Routines

`sec_rgy_acct_add`
sec_rgy_acct_replace_all

Purpose
Replaces all account data for an account.

Format
#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 entered.

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 changed by, the account owner. 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 on login for the account.
- shell The default shell to use on login.
- passwd_version_number The password version number, a 32-bit 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.

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. This information can be changed 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.
**sec_rgy_acct_replace_all**

- **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 changed 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 changed (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. An expiration date value of 0 means there is no expiration date, and the account is valid indefinitely.

- **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 Registry API under "sec_rgy" on page 6-187 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. See Registry API under "sec_rgy" on page 6-187 for a complete description of these flags.

- **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, 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. No abbreviations are allowed, and the only acceptable 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 (through the version_number field of the new_key input parameter), the server returns the requested version number to the client.

- **status**
  A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

  - **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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
Usage

The `sec_rgy_acct_replace_all` API 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 such information as the account home directory and default shell. Typically, the administrative information can only be changed by a registry administrator (users with `admin_info` privileges for an account), while the user information can be changed by the account owner (users with `user_info` privileges for an account).

Use the `set_password` parameter to reset the account password. If you set this parameter to TRUE, the account's password will be 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`.

Required Permissions

The `sec_rgy_acct_replace_all` API requires the following permissions on the account principal:

- `m` (mgmt_info), if flags or expiration date is to be changed
- `a` (auth_info), if authentication flags or good-since-date is to be changed
- `u` (user_info), if user flags, gecos, home directory, shell, or password is to be changed.

All users need the `write` privilege to change any account information.

Related Information

Routines

- `sec_rgy_acct_add`
- `sec_rgy_acct_rename`
- `rgy_acct_user_replace`
- `rgy_acct_admin_replace`
sec_rgy_acct_user_replace

Purpose
Replaces user account data.

Format

```c
#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 entered.

- **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 changed by, the account owner. 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 on login for the account.

  - **shell**
    The default shell to use on login.

  - **passwd_version_number**
    The password version number, a 32-bit 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.

- **set_passwd**
  The password reset parameter. 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`. 
sec_rgy_acct_user_replace

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 (through the version_number field of the new_key input parameter), the server returns the requested version number to the client.

status
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- **sec_rgy_not_authorized**
  The client program is not authorized to change 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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The **sec_rgy_acct_user_replace** API replaces the user information in the account record specified by the input login name. The user information contains such information as the account home directory and default shell. Typically, the user information can be changed by the account owner (users with user_info (u) privileges for an account).

Use the **set_password** parameter to reset the account password. If you set this parameter to TRUE, the account's password will be changed to the value specified in new_key.

Required Permissions

The **sec_rgy_acct_user_replace** API requires the u (user_info) permission on the account principal.

All users need the write (w) privilege to change any account information.

Related Information

Routines

- sec_rgy_acct_add
- sec_rgy_acct_rename
- rgy_acct_replace_all
- rgy_acct_admin_replace
sec_rgy_attr_cursor_alloc

Purpose
Allocates resources to a cursor used by the sec_rgy_attr_lookup_by_id call.

Format
#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 the following error:

  sec_rgy_object_not_found Registry object specified not found.
  sec_attr_no_memory Not enough memory to perform the operation.

Usage
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

Related Information

Routines
sec_rgy_attr_cursor_init sec_rgy_attr_cursor_reset sec_rgy_attr_lookup_by_id
sec_rgy_attr_cursor_release

Files
/usr/include/dce/sec_attr_base.idl Theidlfile from which dce/sec_attr_base.h was derived.
sec_rgy_attr_cursor_init

sec_rgy_attr_cursor_init

Purpose
Initialize a cursor used by the sec_rgy_attr_lookup_by_id call.

Format
#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 *st_p);

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.
st_p A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns the following error:
    sec_rgy_object_not_found Registry object specified not found.
    sec_attr_no_memory Not enough memory to perform the operation.

Usage
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.
Related Information

Routines

sec_rgy_attr_lookup_by_id       sec_rgy_attr_cursor_release
sec_rgy_attr_cursor_release

Purpose
Release a cursor of type sec_attr_cursor_t that was allocated with the sec_rgy_attr_cursor_init() or sec_rgy_attr_cursor_alloc() call.

Format
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_release (  
    sec_attr_cursor_t *cursor,  
    error_status_t *st_p);  

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  
st_p  
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns the following errors:

<table>
<thead>
<tr>
<th>Error Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_rgy_object_not_found</td>
<td>Registry object specified not found.</td>
</tr>
</tbody>
</table>

Usage
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 no remote calls.

Permissions Required
None.

Related Information

Routines
sec_rgy_attr_cursor_init  
sec_rgy_attr_cursor_alloc  
sec_rgy_attr_lookup_by_id
sec_rgy_attr_cursor_reset

Purpose
Re-initializes a cursor that has been allocated with either sec_rgy_attr_cursor_init() or sec_rgy_attr_cursor_alloc().

Format
```c
#include <dce/sec_attr_base.h>

void sec_attr_cursor_reset(
    sec_attr_cursor_t *cursor,
    error_status_t *st_p);
```

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.

Output
st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns the following error:
- sec_attr_bad_cursor Specified cursor is invalid.
- sec_attr_no_memory Not enough memory to perform the operation.

Usage
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 re-allocating it. This is a local operation and makes no remote calls.

Permissions Required
None.

Related Information
Routines
- sec_rgy_attr_cursor_init
- sec_rgy_attr_cursor_alloc
- sec_rgy_attr_lookup_by_id

Files
/usr/include/dce/sec_rgy_attr.idl The idl file from which dce/sec_rgy_attr.h was derived.
sec_rgy_attr_delete

Purpose
Deletes specified attributes for a specified object.

Format
#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 *st_p);

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.

st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_attr_unauthorized: Identity being used does not have the right to perform this operation.
- 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.
- sec_rgy_server_unavailable: The DCE Registry Server is unavailable.
- sec_attr_bad_type: invalid/unsupported attribute type
- sec_acl_site_read_only: The DCE security server is a read-only replica and not the master for the cell. As a result, the server receiving the update request refuses it.
Usage

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 multi-valued and to delete all instances of a multi-valued 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 multi-valued attribute, supply the UUID and value that uniquely identify the multi-valued 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.

Related Information

Routines

`sec_rgy_attr_update`

Files

`/usr/include/dce/sec_rgy_attr.idl`  The idl file from which `dce/sec_rgy_attr.h` was derived.
sec_rgy_attr_get_effective

Purpose
Returns the UUIDs of a specified object's effective attributes.

Format
#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 st_p);

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.

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, 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.

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

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:
  1. If the attribute exists for the object, the attribute value is returned.
  2. If the attribute does not exist and the object is a principal with an account, the organization named in the account is checked. If the attribute exists for the organization, the attribute value is returned.

For multi-valued 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`.

Permissions Required

The `sec_rgy_attr_get_effective()` routine requires the query permission 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.

Related Information

Files

/usr/include/dce/sec_rgy_attr.idl The idl file from which dce/sec_rgy_attr.h was derived.
sec_rgy_attr_lookup_by_id

Purpose
Reads a specified object's attribute(s), expanding attribute sets into individual member attributes.

Format
#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 *st_p);

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 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.
sec_rgy_attr_lookup_by_id

attrs An array of values of type sec_attr_t that contains the attributes 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 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.

st_p 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 one of the following errors:

- sec_rgy_not_authorized The client program is not authorized to perform this function.
- sec_rgy_server_unavailable The DCE Registry Server is unavailable.
- sec_attr_trig_svr_unavailable Trigger server is not available.

Usage

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, all of the object's attributes that the caller is authorized to see are returned. This routine is useful for programmatic access.

For multi-valued 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:

- 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. 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.

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 server begins processing the query at the first attribute that satisfies the search criteria.

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.
sec_rgy_attr_lookup_by_id

Permissions Required

The **sec_rgy_attr_lookup_by_id()** 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.

Related Information

Routines

- **sec_rgy_attr_lookup_no_expand**
- **sec_rgy_attr_attr_lookup_by_name**

Files

- `/usr/include/dce/sec_rgy_attr.idl`  
  The idl file from which `dce/sec_rgy_attr.h` was derived.
sec_rgy_attr_lookup_by_name

Purpose
Read a single attribute instance for a specific object.

Format
#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 *st_p);

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
An 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.

st_p
A pointer to the completion status. The completion status can be one of the following:

- **error_status_ok** if all instances of the value are returned with no errors.
- **more_available** if a multi-valued attribute was specified as name and the routine completed successfully. For multi-valued attributes, this routine returns the first instance of the attribute.
- **attribute_set_instance** if an attribute set was specified as name and the routine completed successfully.
- An error message if the routine did not complete successfully:
  - **sec_rgy_not_authorised** The client program is not authorized to perform this function.
  - **sec_rgy_server_unavailable** The DCE Registry Server is unavailable.
  - **sec_attr_trig_svr_unavailable** Trigger server is not available.
sec_rgy_attr_lookup_by_name

Usage

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 multi-valued 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 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 attribute set UUID returned in the *attr* parameter.

Caution

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 query permission set for the attribute type of the attribute instance identified by *attr_name*. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

Related Information

Routines

- **sec_rgy_attr_lookup_by_id**
- **sec_rgy_attr_lookup_no_expand**
sec_rgy_attr_lookup_no_expand

Purpose
Reads a specified object's attribute(s), without expanding attribute sets into individual member attributes.

Format
#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 *st_p);

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, 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 attr_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 attr_sets[] 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.
sec_rgy_attr_lookup_no_expand

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.

st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_rgy_not_authorized: The client program is not authorized to perform this function.
- sec_rgy_server_unavailable: The DCE Registry Server is unavailable.
- sec_attr_trig_svr_unavailable: Trigger server is not available.

Usage
The sec_rgy_attr_lookup_noexpand() 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_noexpand() 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 server begins processing the query with the first attribute that satisfies the search criteria.

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_noexpand() 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.

Related Information
Routines
sec_rgy_attr_lookup_by_id  sec_rgy_attr_lookup_by_name

Files
/usr/include/dce/sec_rgy_attr.idl The idl file from which dce/sec_rgy_attr.h was derived.
sec_rgy_attr_sch_aclmgr_strings

Purpose
Returns printable ACL strings associated with an ACL manager protecting a bound to schema object.

Format
#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 *st_p);

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.
aclemanger_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_mgns() 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
aclemanger_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.
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.

**Related Information**

**Routines**

`sec_rgy_attr_sch_get_acl_mgrs`

**Files**

`/usr/include/dce/sec_rgy_attr_sch.idl` The idl file from which `dce/sec_rgy_attr_sch.h` was derived.
sec_rgy_attr_sch_create_entry

**Purpose**
Create a schema entry.

**Format**
```c
#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 *st_p);
```

**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**

- **st_p**
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sec_attr_bad_name</code></td>
<td>Attribute name specified is NULL or exceeds 1024 characters.</td>
</tr>
<tr>
<td><code>sec_attr_bad_encoding_type</code></td>
<td>Specified coding type is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_acl_mgr_set</code></td>
<td>Specified ACL manager set is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_acl_mgr_type</code></td>
<td>Specified ACL manager type is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_permset</code></td>
<td>One or more invalid permission sets was specified for this type of ACL.</td>
</tr>
<tr>
<td><code>sec_attr_bad_intercell_action</code></td>
<td>Specified intercell action must be accept, reject, or evaluate.</td>
</tr>
<tr>
<td><code>sec_attr_trig_bind_info_missing</code></td>
<td>If the unique flag is set to true, and a query trigger is used, the intercell action cannot be set to accept.</td>
</tr>
<tr>
<td><code>sec_attr_bad_bind_info</code></td>
<td>Specified binding information is invalid for an attribute update operation.</td>
</tr>
<tr>
<td><code>sec_attr_bad_bind_svr_name</code></td>
<td>Server name specified in binding_auth information is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_bind_prot_level</code></td>
<td>Protection level specified in binding_auth information is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_bind_authn_svc</code></td>
<td>Authentication service specified in binding_auth information is invalid.</td>
</tr>
<tr>
<td><code>sec_attr_bad_bind_authz_svc</code></td>
<td>Invalid authz_svc parameter specified in a request to the privilege server.</td>
</tr>
<tr>
<td><code>sec_attr_bad_uniq_query_accept</code></td>
<td>Invalid combination of unique_flag=true, query trigger, and intercell action=accept. Intercell action can only be reject or evaluate.</td>
</tr>
</tbody>
</table>
sec_rgy_attr_sch_create_entry

sec_attr_bad_scope  Attribute scope specified exceeds 1024 characters.
sec_attr_bad_comment Attribute comment specified exceeds 1024 characters.
sec_attr_type_id_exists Attribute with specified ID already exists in the registry.
sec_attr_name_exists  Attribute with specified name already exists in the registry.
sec_attr_unauthorized Identity does not have the right to perform this operation.
sec_attr_svr_read_only Server is read-only.
sec_attr_svr_unavailable Server is not available.
sec_attr_no_memory Not enough memory to perform the operation.

Usage

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.

Related Information

Routines
sec_rgy_attr_sch_delete_entry  sec_rgy_attr_sch_update

Files
/usr/include/dce/sec_rgy_attr_sch.idl The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_cursor_alloc

Purpose
Allocates resources to a cursor used with the sec_rgy_attr_sch_scan call.

Format

```c
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_att_sch_cursor_alloc(
    dce_attr_cursor_t *cursor,
    error_status_t *st_p);
```

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 the following error:
  - `sec_attr_no_memory` Not enough memory to perform the operation.

Usage

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.

Related Information

**Routines**
- `sec_rgy_attr_sch_cursor_init`
- `sec_rgy_attr_sch_cursor_release`
- `sec_rgy_attr_sch_scan`

**Files**

`/usr/include/dce/sec_rgy_attr_sch.idl` The idl file from which `dce/sec_rgy_attr_sch.id` was derived.
sec_rgy_attr_sch_cursor_init

**Purpose**

Initialize and allocate a cursor used with the `sec_rgy_attr_sch_scan` call.

**Format**

```c
#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 st_p);
```

**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.
- `st_p` A pointer to the completion status. On successful completion, the call returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_attr_unauthorized` Identity does not have the right to perform this operation.
  - `sec_attr_svr_unavailable` Server is not available.
  - `sec_attr_no_memory` Not enough memory to perform this operation.

**Usage**

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 input 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.

**Related Information**

6-394 Application Development Reference Volumes 1 and 2
Routines

sec_rgy_attr_sch_cursor_release  sec_rgy_attr_sch_scan  sec_rgy_attr_sch_cursor_alloc

Files

/usr/include/dce/sec_rgy_attr_sch.idl  The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_cursor_release

sec_rgy_attr_sch_cursor_release

Purpose
Release states associated with a cursor used by the sec_rgy_attr_sch_scan routine.

Format
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_cursor_release(
   sec_attr_cursor_t *cursor,
   error_status_t *st_p);

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

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

Usage

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.

Related Information

Routines
sec_rgy_attr_sch_cursor_init   sec_rgy_attr_sch_cursor_allocate   sec_rgy_attr_sch_scan

Files
/usr/include/dce/sec_rgy_attr_sch.idl    The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_cursor_reset

Purpose
Resets a cursor that has been allocated with either sec_rgy_attr_sch_cursor_init() or sec_rgy_attr_sch_cursor_alloc().

Format
#include <dce/sec_rgy_attr_sch.h>

void dce_attr_cursor_reset(
    sec_attr_cursor_t *cursor,
    error_status_t *st_p);

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.

Output
st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns the following error:

sec_attr_bad_cursor
Specified cursor is invalid.

Usage
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 re-allocating it. This is a local operation and makes no remote calls.

Permissions Required
None.

Related Information

Routines
sec_rgy_attr_sch_cursor_init sec_rgy_attr_sch_cursor_alloc sec_rgy_attr_sch_scan

Files
/usr/include/dce/sec_rgy_attr_sch.idl
The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_delete_entry

Purpose
Delete a schema entry.

Format
#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 *st_p);

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
st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_attr_sch_entry_not_found
  Specified schema entry does not exist.
- sec_attr_unauthorized
  Identity does not have the right to perform this operation.
- sec_attr_svr_read_only
  Server is read-only.
- sec_attr_svr_unavailable
  Server is not available.
- sec_attr_no_memory
  Not enough memory to perform the operation.

Usage
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.

Related Information
Routines
sec_rgy_attr_sch_create_entry  sec_rgy_attr_sch_update_entry
Files
/usr/include/dce/sec_rgy_attr_sch.idl  The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_get_acl_mgrs

Purpose
Retrieve the manager types of the ACLs protecting the objects dominated by a named schema.

Format
#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 *st_p);

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_manager_types[] array.
num_acl_mgr_types An unsigned 32-bit integer containing the number of types returned in the acl_manager_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_manager_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.
st_p A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
  sec_attr_unauthorized Identity does not have the right to perform this operation.
  sec_attr_svr_unavailable Server is not available.
  sec_attr_no_memory Not enough memory to perform the operation.

Usage
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.
Related Information

Routines

sec_rgy_attr_sch_aclmgr_strings

Files

/usr/include/dce/sec_rgy_attr_sch.idl  The idl file from which dce/sec_rgy_attr_sch.h was derived.
sec_rgy_attr_sch_lookup_by_id

Purpose
Read a schema entry identified by UUID.

Format
#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 *st_p);

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. the
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.
st_p  A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_attr_sch_entry_not_found  Specified schema entry does not exist.
- sec_attr_unauthorized  Identity does not have the right to perform this operation.
- sec_attr_svr_unavailable  Server is not available.
- sec_attr_no_memory  Not enough memory to perform the operation.

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

Permissions Required
The sec_rgy_attr_sch_lookup_by_id() routine requires the r permission on the attr_schema object.

Related Information
Routines
sec_rgy_attr_sch_lookup_by_name  sec_rgy_attr_sch_scan
### Files

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/include/dce/sec_rgy_attr_sch.idl</td>
<td>The idl file from which dce/sec_rgy_attr_sch.h was derived.</td>
</tr>
</tbody>
</table>
sec_rgy_attr_sch_lookup_by_name

**Purpose**
Read a schema entry identified by name.

**Format**

```c
#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 *st_p);
```

**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`.

- **st_p**
  A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_attr_sch_entry_not_found`: Specified schema entry does not exist.

**Usage**

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.

**Permissions Required**

The `sec_rgy_attr_sch_lookup_by_name()` routine requires the `r` permission on the `attr_schema` object.

**Related Information**

**Routines**

- `sec_rgy_attr_sch_lookup_by_id`
- `sec_rgy_attr_sch_scan`

**Files**

- `/usr/include/dce/sec_rgy_attr_sch.idl`
  The idl file from which `dce/sec_rgy_attr_sch.h` was derived.
sec_rgy_attr_sch_scan

Purpose
Read a specified number of schema entries.

Format

```c
#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 *st_p);
```

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.
- **st_p**
  - A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - **sec_attr_bad_cursor**
    - Specified cursor is invalid.
  - **sec_attr_unauthorized**
    - Identity does not have the right to perform this operation.
  - **sec_attr_svr_unavailable**
    - Server is not available.
  - **sec_attr_no_memory**
    - Not enough memory to perform the operation.

Usage

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 initialized, `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`. 
secrgy_attr_sch_scan

This routine is useful as a browser.

Permissions Required

The secrgy_attr_sch_scan() routine requires r permission on the attr_schema object.

Related Information

Routines

secrgy_attr_sch_cursor_init secrgy_attr_sch_cursor_alloc secrgy_attr_sch_cursor_release

Files

/usr/include/dce/secrgy_attr_sch.idl The idl file from which dce/secrgy_attr_sch.h was derived.
sec_rgy_attr_sch_update_entry

Purpose
Update a schema entry.

Format
```c
#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 *st_p);
```

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
- **st_p** A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_attr_bad_acl_mgr_type` Specified ACL manager type is invalid.
  - `sec_attr_bad_acl_mgr_set` Specified ACL manager set is invalid.
  - `sec_attr_bad_bind_authn_svc` Authentication service specified in binding_auth information is invalid.
  - `sec_attr_bad_bind_authz_svc` Invalid authz_svc parameter specified in a request to the privilege server.
  - `sec_attr_bad_bind_info` Specified binding information is invalid for an attribute update operation.
  - `sec_attr_bad_bind_prot_level` Protection level specified in binding_auth information is invalid.
  - `sec_attr_bad_bind_svr_name` Server name specified in binding_auth information is invalid.
  - `sec_attr_bad_comment` Attribute comment specified exceeds 1024 characters.
  - `sec_attr_bad_intercell_action` Attribute comment specified exceeds 1024 characters.
  - `sec_attr_bad_permset` One or more invalid permission sets was specified for this type of ACL.
  - `sec_attr_bad_uniq_query_accept` Invalid combination of unique_flag=true, query trigger, and intercell action=accept. Intercell action can only be reject or evaluate.
**sec_rgy_attr_sch_update_entry**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_attr_field_no_update</td>
<td>Field cannot be modified.</td>
</tr>
<tr>
<td>sec_attr_name_exists</td>
<td>Attribute with specified name already exists in the registry.</td>
</tr>
<tr>
<td>sec_attr_no_memory</td>
<td>Not enough memory to perform the operation.</td>
</tr>
<tr>
<td>sec_attr_sch_entry_not_found</td>
<td>Specified schema entry does not exist.</td>
</tr>
<tr>
<td>sec_attr_svr_read_only</td>
<td>Server is read-only.</td>
</tr>
<tr>
<td>sec_attr_svr_unavailable</td>
<td>Server is not available.</td>
</tr>
<tr>
<td>sec_attr_trig_bind_info_missing</td>
<td>A trigger was specified without supplying binding information for it.</td>
</tr>
<tr>
<td>sec_attr_unauthorized</td>
<td>Identity does not have the right to perform this operation.</td>
</tr>
</tbody>
</table>

**Usage**

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 listed below:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- 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 permission on the `attr_schema` object.

**Related Information**

**Routines**

- `sec_rgy_attr_sch_delete_entry`
- `sec_rgy_attr_sch_create_entry`

**Files**

- `/usr/include/dce/sec_rgy_attr_sch.idl` The idl file from which `dce/sec_rgy_attr_sch.h` was derived.
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.

Format
#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 *st_p);

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.

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 to be updated. This integer must be greater than 0.

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 either the test_attrs[] or 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.

st_p
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

sec_rgy_not_authorized The client program is not authorized to perform this function.

sec_rgy_server_unavailable The DCE Registry Server is unavailable.
sec_rgy_attr_test_and_update

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_attr_bad_type</td>
<td>Invalid/unsupported attribute type.</td>
</tr>
<tr>
<td>sec_attr_bad_encoding_type</td>
<td>Specified coding type is invalid.</td>
</tr>
<tr>
<td>sec_attr_not_all_avail</td>
<td>One or more of the test attributes did not match the value stored in the registry database.</td>
</tr>
<tr>
<td>sec_attr_rgy_obj_not_found</td>
<td>The named registry object was not found.</td>
</tr>
<tr>
<td>sec_attr_unauthorized</td>
<td>The user is not authorized to perform the operation.</td>
</tr>
<tr>
<td>sec_attr_no_more_entries</td>
<td>One or more of the test attributes were not found in the registry database.</td>
</tr>
<tr>
<td>sec_attr_sch_entry_not_found</td>
<td>A schema entry was not found for one of the attributes.</td>
</tr>
<tr>
<td>sec_attr_not_unique</td>
<td>Attribute value is not unique.</td>
</tr>
<tr>
<td>sec_attr_trig_svr_unavailable</td>
<td>Trigger server is not available.</td>
</tr>
<tr>
<td>sec_acl_site_read_only</td>
<td>Operation not valid on a read-only site.</td>
</tr>
</tbody>
</table>

Usage

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 update_attrs[], the existing attribute information is overwritten by the new information. For multi-valued 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.

Notes:
1. Attributes in test_attrs[] with a query trigger will be ignored.
2. Attributes in update_attrs[] with an update trigger will be written to the registry database but the trigger server will not be invoked. z/OS DCE does not support synchronization of the registry database with an external trigger server.

Permissions Required

The sec_rgy_attr_test_and_update() routine routine requires the query permission for each attribute in test_attrs[] and the update permission for each attribute in update_attrs[]. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

Related Information
Routines

sec_rgy_attr_update  sec_rgy_attr_delete

Files

/usr/include/dce/sec_rgy_attr.idl  The idl file from which dce/sec_rgy_attr.h was derived.
sec_rgy_attr_update

Purpose
Creates and updates attribute instances for a specified object.

Format
#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 attrs[], 
    unsigned32 *num_returned, 
    sec_attr_t out_attrs[], 
    unsigned32 *num_left, 
    signed32 *failure_index, 
    error_status_t *st_p); 

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 inAttrs array. This integer must be greater than 0.

space_avail
A 32-bit unsigned integer that specifies the size of the out_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 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 outAttrs array.

out_attrs
An array of values of type sec_attr_t that specifies the updated attribute instances. Not that only if these attributes were processed by an update attribute trigger server will they differ from the attributes in the inAttrs[] array. The size of the array is determined by space_avail and the length by num_returned.

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.
In the event of an error, \textit{failure_index} is a pointer to the element in the \textit{in_attrs[]} array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of \textit{failure_index} is -1.

A pointer to the completion status. On successful completion, the routine returns \texttt{error_status_ok}. Otherwise, it returns one of the following errors:

- \texttt{sec_attr_unauthorized}: Identity does not have the right to perform this operation.
- \texttt{sec_acl_site_read_only}: Either the master site is in maintenance mode or the site associated with the handle is a read-only (query) site.
- \texttt{sec_rgy_server_unavailable}: The DCE Registry Server is unavailable.
- \texttt{sec_attr_bad_param}: Bad parameter for schema or attribute operation.
- \texttt{sec_attr_bad_type}: Invalid/unsupported attribute type
- \texttt{sec_attr_bad_encoding_type}: Specified encoding type is invalid.
- \texttt{sec_attr_no_memory}: Not enough memory to perform the operation.
- \texttt{sec_attr_not_unique}: Attribute value is not unique.
- \texttt{sec_attr_inst_exists}: Attribute instance already exists.
- \texttt{sec_attr_trig_srv_unavailable}: Trigger server is not available.

### Usage

The \texttt{sec_rgy_attr_update()} routine creates new attribute instances and updates existing attribute instances attached to an object specified by name and Registry domain. The instances to be created or updated are passed as an array of \texttt{sec_attr_t} data types. This is an atomic operation: if the creation of any attribute in the \textit{in_attrs[]} array fails, all updates are aborted. The attribute causing the update to fail is identified in \textit{failure_index}. If the failure cannot be attributed to a given attribute, \textit{failure_index} contains -1.

The \textit{in_attrs} array, which specifies the attributes to be created, contains values of type \texttt{sec_attr_t}. These values are:

- \texttt{attr_id}, a UUID that identifies the attribute type
- \texttt{attr_value}, values of \texttt{sec_attr_value_t} that specify the attribute's encoding type and values.

If an attribute instance already exists which is identical in both \texttt{attr_id} and \texttt{attr_value} to an attribute specified in \textit{in_attrs}, the existing attribute information is overwritten by the new information. For multi-valued attributes, every instance with the same \texttt{attr_id} is overwritten with the supplied values.

If an attribute instance does not exist, it is created.

For multi-valued attributes, because every instance of the multi-valued 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 multi-valued attributes, create individual \texttt{sec_attr_t} data types to define each multi-valued attribute instance and then pass all of them in the input array.

If an input attribute is associated with an update attribute trigger server, the attribute trigger server is invoked (by the \texttt{sec_attr_trig_update()} function) and the \textit{in_attrs[]} array is supplied as input. The output attributes from the update attribute trigger server are stored in the registry database and returned in the \textit{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 \textit{out.attrs[]} array differ from the values in the \textit{in.attrs[]} array.

### Permissions Required

The \texttt{sec_rgy_attr_update()} routine requires the update permission set for each attribute type identified in the \textit{in.attrs[]} array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.
sec_rgy_attr_update

Related Information

Routines
sec_rgy_attr_delete  sec_rgy_attr_test_and_update

Files
/usr/include/dce/sec_rgy_attr.idl
The idl file from which dce/sec_rgy_attr.h was derived.
sec_rgy_auth_plcy_get_effective

Purpose
Returns the effective authentication policy for an account.

Format
#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 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 one of the following errors:

- sec_rgy_object_not_found The specified account could not be found.
- sec_rgy_server_unavailable The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_rgy_auth_plcy_get_effective API 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.

Required Permissions

The sec_rgy_auth_plcy_get_effective API 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.

Related Information
sec_rgy_auth_plcy_get_effective

Routines

sec_rgy_auth_plcy_get_info        sec_rgy_auth_plcy_set_info
sec_rgy_auth_plcy_get_info

Purpose
Returns the authentication policy for an account.

Format
#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 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 one of the following errors:
               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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_auth_plcy_get_info API returns the authentication policy for the specified account. If no account is specified, the registry's authentication policy is returned.

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.

Required Permissions
The sec_rgy_auth_plcy_get_info API requires the r (read) permission on the policy object or account principal from which the data is to be returned.

Related Information
sec_rgy_auth_plcy_get_info

Routines

sec_rgy_auth_plcy_get_effective  sec_rgy_auth_plcy_set_info
sec_rgy_auth_plcy_set_info

Purpose
Sets the authentication policy for an account.

Format

```c
#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 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 returns `error_status_ok`. Otherwise, it returns one of the following errors:
  
  - **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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_auth_plcy_set_info` API 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.

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, because the effective policy is the most restrictive combination of the principal and registry authentication policies. See "sec_rgy_auth_plcy_get_effective" on page 6-415 for information on getting the authentication policy for an account.

Required Permissions

The `sec_rgy_auth_plcy_set_info` API requires the `a` (auth_info) permission on the policy object or account principal for which the data is to be set.

Related Information
sec_rgy_auth_plcy_set_info

Routines

sec_rgy_auth_plcy_get_effective  sec_rgy_auth_plcy_get_info
sec_rgy_cell_bind

Purpose
Binds to a registry in a cell.

Format

```c
#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. On 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 path name 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 "rpc_binding_set_auth_info" on page 2-79 for information on setting up a server binding handle.

Output

- **context**
  A pointer to a `sec_rgy_handle_t` variable. On 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 one of the following errors:

  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.

Usage

The `sec_rgy_cell_bind` API establishes a relationship with a registry site, at an arbitrary level of security. The `cell_name` parameter identifies the target cell.

Related Information

Routines

- `sec_rgy_site_bind`
sec_rgy_cursor_reset

Purpose
Resets the registry database cursor.

Format
#include <dce/misc.h>
void sec_rgy_cursor_reset(
    sec_rgy_cursor_t *cursor );

Parameters
Input/Output
cursor A pointer into the registry database.

Usage
The sec_rgy_cursor_reset API 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 a routine is 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 replica, 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 call. 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.

On successful completion, the routine returns error_status_ok. Otherwise, it returns the following error:
sec_rgy_server_unavailable The DCE Registry Server is unavailable.

Examples
The following example illustrates the 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. Each iteration of the loop calls sec_rgy_pgo_get_next to return the next PGO item in the registry. Each sec_rgy_pgo_get_next call updates 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

Routines

- sec_rgy_acct_get_projlist
- sec_rgy_acct_lookup
- sec_rgy_pgo_get_by_id
- sec_rgy_pgo_get_by_name
- sec_rgy_pgo_get_by_unix_num
- sec_rgy_pgo_get_members
- sec_rgy_pgo_get_next
sec_rgy_login_get_effective

Purpose
Returns the effective login data for an account.

Format

```c
#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 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 entered.

**Output**

- `key_parts` A pointer to the minimum abbreviation allowed when logging in to the account. No abbreviations are allowed, and the only acceptable 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 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_plcy_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 one of the following errors:

- sec_rgy_object_not_found The specified account could not be found.
- sec_rgy_server_unavailable The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_rgy_login_get_effective API 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.

Related Information

Routines
sec_rgy_acct_add                sec_rgy_login_get_info

Files: passwd_override
sec_rgy_login_get_info

Purpose
Returns login information for an account.

Format
#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 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 entered.

Output
key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. No abbreviations are allowed, and the only acceptable value is sec_rgy_acct_key_person.

sid
A pointer to a sec_rgy_sid_t structure to receive the UUIDs 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.
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.

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.

A pointer indicating the total number of projects in the specified account's project list.

The name of the account’s cell.

The UUID for the account’s cell.

A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- secrgy_object_not_found: The specified account could not be found.
- secrgy_server_unavailable: The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

The secrgy_login_get_info API 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 secrgy_login_get_effective.

The secrgy_login_get_info API requires the r (read) permission on the account principal from which the data is to be returned.

Related Information

Routines

tsecrgy_acct_add

tsecrgy_login_get_effective
sec_rgy_pgo_add

sec_rgy_pgo_add

Purpose
Adds a PGO item to the registry database.

Format
#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 PGO item identified by the given name. The valid values are:

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 one of the following errors:

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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_rgy_pgo_add API adds a principal, group, or organization item to the registry database.

The PGO data consists of:
- The 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. If the registry uses embedded UNIX IDs (where a subset of the UUID bits represent the UNIX ID), then the specified ID must match the UUID, if both are specified. Use a value of -1 for the UNIX number to match any value.
- The creation 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

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 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. tom must be a member of writers, which must be a member of hp. See [sec_rgy_acct_add on page 6-354] for information on adding an account to the registry.

**Required Permissions**

The sec_rgy_pgo_add API requires the i (insert) permission on the parent directory in which the PGO item is to be created.

**Related Information**

**Routines**

- sec_rgy_pgo_delete
- sec_rgy_pgo_replace
- sec_rgy_acct_add
- sec_rgy_pgo_rename
Purpose
Adds a person to a group or organization.

Format

```c
#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 person_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 PGO item identified by the given name. The valid values are:
    - `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 person will be added.

- **person_name**
  - A character string (type `sec_rgy_name_t`) containing the name of the person to be added to the membership list of the group or organization specified by `go_name`.

Output

- **status**
  - A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
    - `sec_rgy_bad_domain`
      - The domain specified is not valid. A member can be added only to a group or organization, not person.
    - `sec_rgy_not_authorized`
      - The client program is not authorized to add members to the specified group or organization.
    - `sec_rgy_object_not_found`
      - The registry server could not find the specified name.
    - `sec_rgy_server_unavailable`
      - The DCE Registry Server is unavailable.

Usage

The `sec_rgy_pgo_add_member` API adds a member to the membership list of a group or organization in the registry database.

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 established. For example, to establish an account with person name `tom`, group name `writers`, and organization name `hp`, all three names must exist as independent PGO items in the database. `tom` must be a member of `writers`, and a member of `hp`. See "sec_rgy_acct_add" on page 6-354 for information on adding an account to the registry.
Required Permissions

The `sec_rgy_pgo_add_member` API 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 `person_name`.

Related Information

Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_delete_member`
- `sec_rgy_pgo_get_members`
- `sec_rgy_pgo_is_member`
sec_rgy_pgo_delete

Purpose
Deletes a PGO item from the registry database.

Format
#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 the PGO item identified by the given name. The valid values are:
- 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 one of the following errors:
- 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 name.
- sec_rgy_server_unavailable The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_pgo_delete API deletes a principal, group, or organization item from the registry database. Any account depending on the deleted PGO item is also deleted.

Required Permissions
The sec_rgy_pgo_delete API requires the following permissions:
- d (delete) permission on the parent directory that contains the PGO item to be deleted.
- D (Delete_object) permission on the PGO item itself.

Related Information
Routines

sec_rgy_pgo_add
Purpose
Deletes a member of a group or organization.

Format

```c
#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 person_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 PGO item identified by the given name. The valid values are:
    - `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 person will be deleted.

- **person_name**
  - A character string (type `sec_rgy_name_t`) containing the name of the person to be deleted from the membership list of the group or organization specified by `go_name`.

**Output**

- **status**
  - A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
    - `sec_rgy_not_authorized`
      - The client program is not authorized to delete the specified member.
    - `sec_rgy_bad_domain`
      - The domain specified is not valid. Members can exist only for groups and organizations, not persons.
    - `sec_rgy_object_not_found`
      - The specified group or organization was not found.
    - `sec_rgy_server_unavailable`
      - The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_pgo_delete_member` API deletes a member from the membership list of a group or organization. Any accounts in which the person holds the deleted group or organization membership are also deleted.

Required Permissions

The `sec_rgy_pgo_delete_member` API requires the M (Member_list) permission on the group or organization item specified by `go_name`. 
Related Information

Routines

sec_rgy_pgo_add  sec_rgy_pgo_add_member
sec_rgy_pgo_get_by_eff_unix_num

sec_rgy_pgo_get_by_eff_unix_num

Purpose
Returns the name and data for a PGO item identified by its effective UNIX number.

Format
#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 *st_p);

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.

allowAliases 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 one of the following errors:

- `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.

Usage

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.
Related Information

Routines

- sec_rgy_pgo_add
- sec_rgy_pgo_get_by_id
- sec_rgy_pgo_get_by_name
- sec_rgy_pgo_get_by_unix_num
- sec_rgy_pgo_get_next
- sec_rgy_pgo_id_to_name
- sec_rgy_pgo_id_to_unix_num
- sec_rgy_pgo_name_to_id
- sec_rgy_pgo_unix_num_to_id
- sec_rgy_cursor_reset

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.
sec_rgy_pgo_get_by_id

Purpose
Returns the name and data for a PGO item identified by its UUID.

Format
#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 PGO item identified by the given name. The valid values are:

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. 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 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 API returns the PGO item indicated by item_cursor, and advances the cursor to the point to the next item in the database. When the end of the list of entries is reached, the routine returns the 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, has a concurrent group set.

name
A pointer to a sec_rgy_name_t character string containing the returned name for the PGO item.
sec_rgy_pgo_get_by_id

status  A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- 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.

Usage

The sec_rgy_pgo_get_by_id API 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 placeholder in the database. The routine 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.

Several different types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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 rgy_pgo_get_next. The behavior in this case is undefined.

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.

Required Permissions

The sec_rgy_pgo_get_by_id API requires the r (read) permission on the PGO item to be viewed.

Related Information

Routines

| sec_rgy_pgo_add | sec_rgy_pgo_get_next | sec_rgy_pgo_name_to_id |
| sec_rgy_pgo_get_by_name | sec_rgy_pgo_id_to_name | sec_rgy_pgo_unix_num_to_id |
| sec_rgy_pgo_get_by_unix_num | sec_rgy_pgo_id_to_unix_num | sec_rgy_cursor_reset |

6-440  Application Development Reference Volumes 1 and 2
sec_rgy_pgo_get_by_name

Purpose
Returns the data for a named PGO item.

Format
#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 PGO item identified by the given name. The valid values are:

    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.

Output
item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The sec_rgy_pgo_get_by_name API returns the PGO item indicated by item_cursor, and advances the cursor to the point to the next item in the database. When the end of the list of entries is reached, the routine returns the sec_rgy_no_more_entries in the status parameter. Use sec_rgy_cursor_reset to reset the cursor.

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 one of the following errors:

    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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
sec_rgy_pgo_get_by_name

Usage

The sec_rgy_pgo_get_by_name API 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 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.

Several different types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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 rgy_pgo_get_next. The behavior in this case is undefined.

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.

Required Permissions

The sec_rgy_pgo_get_by_name API requires the r (read) permission on the PGO item to be viewed.

Related Information

Routines

sec_rgy_pgo_add  sec_rgy_pgo_get_next  sec_rgy_pgo_name_to_id
sec_rgy_pgo_get_by_id  sec_rgy_pgo_id_to_name  sec_rgy_pgo_unix_num_to_id
sec_rgy_pgo_get_by_unix_num  sec_rgy_pgo_id_to_unix_num  sec_rgy_cursor_reset
sec_rgy_pgo_get_by_unix_num

Purpose
Returns the name and data for a PGO item identified by its UNIX ID.

Format
```c
#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 PGO item identified by the given name. The valid values are:
  - `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. 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` API returns the PGO item indicated by `item_cursor`, and advances the cursor to the point to the next item in the database. When the end of the list of entries is reached, the routine returns the `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.
A pointer to the completion status. On successful completion, the routine returns
\textit{error\_status\_ok}. Otherwise, it returns one of the following errors:

- \textit{sec\_rgy\_no\_more\_entries}: The cursor is at the end of the list of PGO items.
- \textit{sec\_rgy\_object\_not\_found}: The specified PGO item was not found.
- \textit{sec\_rgy\_server\_unavailable}: The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated \textit{error\_status\_t} variable where required.

Usage

The \texttt{sec\_rgy\_pgo\_get\_by\_unix\_num} API returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UNIX number.

The \texttt{item\_cursor} parameter specifies the starting point for the search through the registry database. It provides an automatic \textit{place holder} in the database. The routine 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.

Several types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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 \texttt{sec\_rgy\_acct\_get\_projlist} and \texttt{rgy\_pgo\_get\_next}. The behavior in this case is undefined.

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 \texttt{sec\_rgy\_cursor\_reset} to renew a cursor for use with another call or for another server.

Required Permissions

The \texttt{sec\_rgy\_pgo\_get\_by\_unix\_num} API requires the \texttt{r} (read) permission on the PGO item to be viewed.

Related Information

Routines

- \texttt{sec\_rgy\_pgo\_add}
- \texttt{sec\_rgy\_pgo\_get\_by\_id}
- \texttt{sec\_rgy\_pgo\_get\_by\_name}
- \texttt{sec\_rgy\_pgo\_get\_next}
- \texttt{sec\_rgy\_pgo\_id\_to\_name}
- \texttt{sec\_rgy\_pgo\_id\_to\_unix\_num}
- \texttt{sec\_rgy\_pgo\_name\_to\_id}
- \texttt{sec\_rgy\_pgo\_unix\_num\_to\_id}
- \texttt{sec\_rgy\_cursor\_reset}
sec_rgy_pgo_get_members

Purpose
Returns the membership list for a group or organization, or returns the set of groups in which the specified principal is a member.

Format
#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 registry server. Use sec_rgy_site_open to acquire a bound handle.

name_domain This variable identifies the type of the PGO item identified by the given name. The valid values are:

    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 a principal.

go_name A character string (type sec_rgy_name_t) containing the name of the group or organization or principal whose members are to be returned. 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 of which the principal is a member. Contrast this with the sec_rgy_acct_get_proj call, which returns only those groups of 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 returned by a single call.

Input/Output
member_cursor An opaque pointer indicating a specific member entry in the membership list of the specified group or organization. The returned list of members begins with the entry specified by member_cursor. On return, the cursor points to the next entry after the last one returned. If there are no more entries in the specified membership list, 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 in the membership list, multiple calls must be made to return all of the members or groups.

number_supplied A pointer to a signed32 variable to receive the number of members actually returned in member_list.
sec_rgy_pgo_get_members

description:

- **number_members**: A pointer to a `signed32` variable to receive the total number of members or groups in the specified group or organization. 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, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_rgy_no_more_entries`: The cursor points to the end of the membership list for the group or organization.
  - `sec_rgy_object_not_found`: The specified group or organization could not be found.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.

Usage:

The `sec_rgy_pgo_get_members` API 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 placeholder in the database. The routine updates this variable 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.

Several different types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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 `rgy_pgo_get_next`. The behavior in this case is undefined.

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.

Required Permissions:

The `sec_rgy_pgo_get_members` API requires the `r` (read) permission on the group, organization or principal object specified by `go_name`.

Return Values:

The routine returns:

- The name 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 of which the principal is a member in (in `number_members`).

Related Information
Chapter 6. Security and Related Services

sec_rgy_pgo_get_members

Routines

sec_rgy_pgo_add_member  sec_rgy_cursor_reset  sec_rgy_pgo_is_member
sec_rgy_pgo_get_next

sec_rgy_pgo_get_next

Purpose
Returns the next PGO item in the registry database.

Format

```c
#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 PGO item identified by the given name. The valid values are:
  - `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. 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` API returns the PGO item indicated by `item_cursor`, and advances the cursor to the point to the next item in the database. When the end of the list of entries is reached, the routine returns the `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 one of the following errors:

- **sec_rgy_status_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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The **sec_rgy_pgo_get_next** API returns the data and name for the principal, group, or organization (PGO) in the registry database indicated by the **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 UUID of the PGO item.
- The UNIX number for the PGO item.
- The quota for sub-accounts.
- 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.

Several different types of cursors are used in the registry API. Some cursors point to PGO items, some point to members in a membership list, and some 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 **rgy_pgo_get_next**. The behavior in this case is undefined.

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.

Required Permissions

The **sec_rgy_pgo_get_next** API requires the r (read) permission on the PGO item to be viewed.

Return Values

The routine returns the data for the returned PGO item in **pgo_item** and the name in **name**.

Related Information

Routines

| sec_rgy_pgo_add | sec_rgy_pgo_get_by_name | sec_rgy_pgo_id_to_unix_num |
| sec_rgy_cursor_reset | sec_rgy_pgo_get_by_unix_num | sec_rgy_pgo_unix_num_to_id |
| sec_rgy_pgo_get_by_id | | |
sec_rgy_pgo_id_to_name

sec_rgy_pgo_id_to_name

Purpose
Returns the name for a PGO item identified by its UUID.

Format
#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 PGO item identified by the given name. The valid values are:
- sec_rgy_domain_person: The item_id parameter identifies a parameter.
- 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

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 one of the following errors:
- sec_rgy_object_not_found: No item with the specified UUID could be found.
- sec_rgy_server_unavailable: The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_pgo_id_to_name API returns the name of the principal, group, or organization item having the specified UUID.

Required Permissions
The sec_rgy_pgo_id_to_name API requires at least one permission of any kind on the PGO item to be viewed.

Related Information
Chapter 6. Security and Related Services

Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_name_to_id`
- `sec_rgy_pgo_unix_num_to_id`
sec_rgy_pgo_id_to_unix_num

Purpose
Returns the UNIX number for a PGO item identified by its UUID.

Format

```c
#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 PGO item identified by the given name. The valid values are:
  
  - `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.

**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 one of the following errors:

  - `sec_rgy_object_not_found`
    No item with the specified UUID could be found.

  - `sec_rgy_server_unavailable`
    The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_pgo_id_to_unix_num` API returns the UNIX number for the principal, group, or organization item having the specified UUID.

Related Information

**Routines**

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_name_to_id`
- `sec_rgy_pgo_unix_num_to_id`
sec_rgy_pgo_is_member

Purpose
Checks group or organization membership.

Format
#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 person_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 PGO item identified by the given name. The valid values are:
    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.
person_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.

Output
status A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
    sec_rgy_object_not_found The named group or organization was not found.
    sec_rgy_server_unavailable The DCE Registry Server is unavailable.
If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_pgo_is_member API tests whether the specified principal is a member of the named group or organization.

Required Permissions
The sec_rgy_pgo_is_member API 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.

Related Information
sec_rgy_pgo_is_member

Routines

sec_rgy_pgo_add_member  sec_rgy_pgo_get_members
sec_rgy_pgo_name_to_id

Purpose
Returns the UUID for a named PGO item.

Format

```c
#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 PGO item identified by the given name. The valid values are:
  - `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 of the resulting PGO item.
- `status` - A pointer to the completion status. On successful completion, the routine returns `error_status_ok`. Otherwise, it returns one of the following errors:
  - `sec_rgy_object_not_found` - The specified PGO item could not be found.
  - `sec_rgy_server_unavailable` - The DCE Registry Server is unavailable.

Usage

The `sec_rgy_pgo_name_to_id` API returns the UUID associated with the named PGO item.

Related Information

Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_unix_num_to_id`
sec_rgy_pgo_name_to_unix_num

Purpose
Returns the UNIX number for a PGO item identified by its name.

Format

```c
#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 PGO item identified by the given name. The valid values are:
  - `sec_rgy_domain_person`: `item_id` identifies a principal.
  - `sec_rgy_domain_group`: `item_id` identifies a group.
  - `sec_rgy_domain_org`: `item_id` 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 one of the following errors:
  - `sec_rgy_object_not_found`: No item with the specified UUID could be found.
  - `sec_rgy_server_unavailable`: The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_pgo_name_to_unix_num` API returns the UNIX number for the principal, group, or organization (PGO) item having the specified name.

Related Information

Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_unix_num_to_id`
sec_rgy_pgo_rename

Purpose
Changes the name of a PGO item in the registry database.

Format
```
#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 PGO item identified by the given name. The valid values are:
  - `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 one of the following errors:
  - `sec_rgy_not_authorized` The client program is not authorized to change the name of the specified PGO item.
  - `sec_rgy_object_not_found` No item with the specified UUID could be found.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.

Usage

The `sec_rgy_pgo_rename` API renames a principal, group, or organization item in the registry database.

Required Permissions

If the `sec_rgy_pgo_rename` API 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:

- d (delete) permission on the parent directory that contains the PGO item.
- n (name) permission on the old name of the PGO item.
- i (insert) permission on the parent directory in which the PGO item is to be added under the new name.

Related Information
sec_rgy_pgo_rename

Routines

sec_rgy_pgo_add  sec_rgy_pgo_replace
sec_rgy_pgo_replace

Purpose
Replaces the data in an existing PGO item.

Format

#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 PGO item identified by the given name. The valid values are:

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 one of the following errors:

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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_rgy_pgo_replace API replaces the data associated with a principal, group, or organization (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.
Required Permissions

The `sec_rgy_pgo_replace` API requires at least one of the following permissions:

- `m` (mgmt_info) permission on the PGO item, if quota or flags is being set.
- `f` (fullname) permission on the PGO item, if fullname is being set.

Related Information

Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_delete`
- `sec_rgy_pgo_rename`
sec_rgy_pgo_unix_num_to_id

Purpose
Returns the UUID for a PGO item identified by its UNIX number.

Format
#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 PGO item identified by the given name. The valid values are:
    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 of the resulting PGO item.
status A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
    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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_pgo_unix_num_to_id routine returns the (UUID) for a PGO item that has the specified UNIX number.

Related Information
Routines
sec_rgy_pgo_add  sec_rgy_pgo_get_by_unix_num  sec_rgy_pgo_id_to_unix_num
sec_rgy_pgo_get_by_id  sec_rgy_pgo_id_to_name  sec_rgy_pgo_name_to_id
sec_rgy_pgo_get_by_name

Chapter 6. Security and Related Services 6-461
sec_rgy_pgo_unix_num_to_name

Purpose
Returns the name for a PGO item identified by its UNIX number.

Format
#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 PGO item identified by item_unix_id. The valid values are:
    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 one of the following errors:
    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.

Usage
The sec_rgy_pgo_unix_num_to_name API returns the name for a PGO item that has the specified UNIX number.

Required Permissions
The sec_rgy_pgo_unix_num_to_name API requires at least one permission of any kind on the PGO item identified by item_unix_id.

Related Information
Routines

- `sec_rgy_pgo_add`
- `sec_rgy_pgo_get_by_id`
- `sec_rgy_pgo_get_by_name`
- `sec_rgy_pgo_get_by_unix_num`
- `sec_rgy_pgo_id_to_name`
- `sec_rgy_pgo_id_to_unix_num`
- `sec_rgy_pgo_name_to_id`
- `sec_rgy_pgo_unix_num_to_name`
sec_rgy_plcy_get_effective

Purpose
Returns the effective policy for an organization.

Format

```c
#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 one of the following errors:
  - `sec_rgy_object_not_found` The registry server could not find the specified organization.
  - `sec_rgy_server_unavailable` The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_plcy_get_effective` API 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 password expiration date. This is the date on which account passwords will expire. An expiration date value of 0 means there is no expiration date, and the password is valid indefinitely.
- 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.

If no organization is specified, the routine returns the registry’s policy data. To return the effective policy, you must specify an organization. The routine compares the registry’s policy data with that of the organization to determine which is more restrictive.
Required Permissions

The sec_ry_plcy_get_effective API 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.

Related Information

Routines

sec_ry_plcy_get_info  sec_ry_plcy_set_info
sec_rgy_plcy_get_info

Purpose
Returns the policy for an organization.

Format
#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 one of the following errors:

  - sec_rgy_object_not_found: The registry server could not find the specified organization.
  - sec_rgy_server_unavailable: The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_plcy_get_info API 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 password expiration date. This is the date on which account passwords will expire. An expiration date value of 0 means there is no expiration date, and the password is valid indefinitely.
- 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.

The returned policy may not be in effect if the overriding registry authorization policy is more restrictive. For information on the authentication policy, see [“sec_rgy_auth_plcy_get_effective” on page 6-415](#).
Required Permissions

The \texttt{sec\_rgy\_plcy\_get\_info} API requires the \texttt{r} (read) permission on the policy object or organization from which the data is to be returned.

Related Information

Routines

\texttt{sec\_rgy\_plcy\_get\_effective\_info} \hspace{1cm} \texttt{sec\_rgy\_plcy\_set\_info}
sec_rgy_plcy_set_info

Purpose
Sets the policy for an organization.

Format
#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 one of the following errors:

- sec_rgy_not_authorized: 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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_plcy_set_info API 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.

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, because the effective policy is the most restrictive combination of the organization and registry authentication policies. For information on the authentication policy, see sec_rgy_auth_plcy_get_effective on page 6-415.

Required Permissions
The sec_rgy_plcy_set_info API requires the m (mgmt_info) permission on the policy object or organization for which the data is to be set.

Related Information
Routines

sec_rgy_plcy_get_effective  sec_rgy_plcy_get_info
sec_rgy_properties_get_info

Purpose
Returns registry properties.

Format
#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 one of the following errors:

- sec_rgy_server_unavailable The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_properties_get_info API returns a list of the registry properties.

The property information consists of:

- 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. 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 indicating whether:
The `sec_rgy_properties_get_info` API requires the `r` (read) permission on the policy object from which the property information is to be returned.

### Related Information

#### Routines

- `sec_rgy_properties_set_info`
sec_rgy_properties_set_info

Purpose
Sets registry properties.

Format
#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 one of the following errors:

                sec_rgy_not_authorized        User is not authorized to change the registry
                                            properties.

                sec_rgy_server_unavailable     The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always
pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_properties_set_info API sets the registry properties.
The property information consists of:

- **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. 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 indicating whether:
  - **sec_rgy_prop_readonly**: If TRUE, the registry is a read-only site.
  - **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. The UNIX ID of any registry object cannot be changed, because UUID's cannot be changed.

**Required Permissions**

The **sec_rgy_properties_set_info** API requires the **m** (mgmt_info) permission on the policy object for which the property information is to be set.

**Related Information**

**Routines**

- **sec_rgy_properties_get_info**
sec_rgy_site_bind

Purpose
Binds to a registry site.

Format
#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) containing the name of the registry site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, /.../r_d.com or / for the local cell).
- To bind to a specific site in a specific cell, specify either the site’s global name (for example, /.../r_d.com/subsys/dce/sec/rs_server_250_2) or the site’s network address (for example, ncdag_ip_udp:15.22.144.248).

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. For information on setting a server binding handle, see "rpc_binding_set_auth_info" on page 2-79.

Output

context
A pointer to a sec_rgy_handle_t variable. On 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 one of the following errors:

- sec_rgy_server_unavailable: The DCE Registry Server is unavailable.

Usage

The sec_rgy_site_bind API 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 API, 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.

Related Information
Routines

sec_rgy_site_open sec_rgy_cell_bind
sec_rgy_site_bind_query

sec_rgy_site_bind_query

Purpose
Binds to a registry query site.

Format
#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) containing the name of the registry site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, /.../r_d.com or /.: for the local cell).
- To bind to a specific site in a specific cell, specify either the site's global name (for example, /.../r_d.com/subsys/dce/sec/rs_server_250_2) or the site's network address (for example, ncadg_ip_udp:15.22.144.248).

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 "rpc_binding_set_auth_info" on page 2-79.

Output
context
A pointer to a sec_rgy_handle_t variable. On 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 one of the following errors:

- sec_rgy_server_unavailable  The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage

The sec_rgy_site_bind_query API 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, you must change a registry update site, which then 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. Like the sec_rgy_bind_open API, 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.

Related Information
<table>
<thead>
<tr>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sec_rgy_site_open</code></td>
</tr>
<tr>
<td><code>sec_rgy_site_bind</code></td>
</tr>
</tbody>
</table>
sec_rgy_site_bind_update

Purpose
Binds to the registry update site.

Format
#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 registry site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, /.../r_d.com or /.: for the local cell).
- To bind to a specific site in a specific cell, specify either the site’s global name (for example, /.../r_d.com/subsys/dce/sec/rs_server_250_2) or the site’s network address (for example, ncadg_ip_udp:15.22.144.248).

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 "rpc_binding_set_auth_info" on page 2-79

Output
context A pointer to a sec_rgy_handle_t variable. On 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 one of the following errors:

- sec_rgy_server_unavailable The DCE Registry Server is unavailable.

Usage
The sec_rgy_site_bind_update API binds to the registry update site. The registry update site is a master server that may control several satellite (query) servers. To change the registry database, you must change a registry update site, which then 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 registry database (for example, the sec_rgy_pgo_add or sec_rgy_acct_lookup call).

Related Information
Routines

sec_rgy_site_open   sec_rgy_site_bind
sec_rgy_site_binding_get_info

Purpose
Returns information from the registry.

Format
#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 the 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. For information on
setting a binding handle, see "rpc_binding_set_auth_info" on page 2-79.

status
A pointer to the completion status. On successful completion, the routine returns
error_status_ok. Otherwise, it returns one of the following errors:

  sec_rgy_server_unavailable
  The DCE Registry Server is unavailable.

Usage
The sec_rgy_site_binding_get_info API 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.

Storage is allocated for the string returned in the cell_name, server_name, and string_binding parameters. The application
calls the rpc_string_free API to deallocate that storage.

Related Information
Routines

sec_rgy_site_open  sec_rgy_site_bind
sec_rgy_site_close

Purpose
Frees the binding handle for a registry server.

Format
#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.

Note: If an invalid context is passed, error_status_ok is returned.

Output
status
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

sec_rgy_server_unavailable The requested registry server is not available.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_site_close API frees the storage occupied by the specified handle and destroys its binding with the registry server.

A handle cannot be used after it is freed.

Related Information

Routines
sec_rgy_site_get sec_rgy_site_open sec_rgy_site_open_update
sec_rgy_site_is_readonly sec_rgy_site_open_query
sec_rgy_site_get

Purpose
Returns the string representation for a bound registry site.

Format

```c
#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 one of the following errors:
  
  - `sec_rgy_server_unavailable`: The requested registry server is not available.
  
  If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_site_get` API 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.

Storage is allocated for the string returned in the `site_name` parameter. The application calls the `rpc_string_free` API to deallocate that storage.

Note: To obtain binding information, the use of the `sec_rgy_binding_get_info` API is recommended in place of `sec_rgy_site_get`.

Notes

To obtain binding information, the use of the `sec_rgy_site_binding_get_info` API is recommended instead of this API.

Related Information
sec_rgy_site_get

Routines
sec_rgy_site_open
sec_rgy_site_is_readonly

Purpose
Checks whether a registry site is read only.

Format
#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.

Usage
The sec_rgy_site_is_readonly API 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; and FALSE if it is an update site.

Related Information
Routines
sec_rgy_site_open sec_rgy_site_open_query
sec_rgy_site_open

Purpose
Binds to a registry site.

Format
#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) containing the name of the registry site to bind to. Supply this name in any of the following forms:
- To randomly choose a site to bind to in the named cell, specify a cell name (for example, /.../r_d.com or /.: for the local cell).
- To bind to a specific site in a specific cell, specify either the site’s global name (for example, /.../r_d.com/subsys/dce/sec/rs_server_250_2) or the site’s network address (for example, ncadg_ip_udp:15.22.144.248).

Output
context A pointer to a sec_rgy_handle_t variable. On 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 one of the following errors:
- sec_rgy_server_unavailable The requested registry server is not available.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_site_open API 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. Like sec_rgy_site_open_query API, 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.

Note: To bind to a registry site, the use of the sec_rgy_site_bind API is recommended in place of sec_rgy_site_open.

Related Information
Routines
sec_rgy_site_close sec_rgy_site_open_query sec_rgy_site_open_update
sec_rgy_site_is_readonly
sec_rgy_site_open_query

Purpose
Binds to a registry query site.

Format
```c
#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`) containing the name of the registry site to bind to.  
Supply this name in any of the following forms:
- To randomly choose a site to bind to in the named cell, specify a cell name (for example, `././.d.com` or `./.` for the local cell).
- To bind to a specific site in a specific cell, specify either the site’s global name (for example, `././.d.com/subsys/dce/sec/rs_server_250_2`) or the site’s network address (for example, `ncadg_ip_udp:15.22.144.248`).

Output

*context*  
A pointer to a `sec_rgy_handle_t` variable. On 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 one of the following errors:
- `sec_rgy_server_unavailable`  
The DCE Registry Server is unavailable.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated `error_status_t` variable where required.

Usage

The `sec_rgy_site_open_query` API 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, you must change a registry update site, which then 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 of the associated registry server is returned in `context`.

Like `sec_rgy_site_open` API, 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`.

Note: To bind to a registry site, the use of the `sec_rgy_site_bind_query` API is recommended in place of `sec_rgy_site_open_query`.

Related Information
sec_rgy_site_open_query

Routines

<table>
<thead>
<tr>
<th>sec_rgy_site_close</th>
<th>sec_rgy_site_is_readonly</th>
<th>sec_rgy_site_open_update</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_rgy_site_get</td>
<td>sec_rgy_site_open</td>
<td></td>
</tr>
</tbody>
</table>
sec_rgy_site_open_update

Purpose
Binds to the registry update site.

Format
```
#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) containing the name of the registry site to bind to. Supply this name in any of the following forms:
- To randomly choose a site to bind to in the named cell, specify a cell name (for example, /.../r_d.com or .: for the local cell).
- To bind to a specific site in a specific cell, specify either the site’s global name (for example, /.../r_d.com/subsys/dce/sec/rs_server_250_2) or the site’s network address (for example, ncadg_ip_udp:15.22.144.248).

Output
context A pointer to a sec_rgy_handle_t variable. On 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 one of the following errors:
- sec_rgy_server_unavailable The DCE Registry Server is unavailable.
If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_site_open_update API binds to the registry update site. The registry update site is the master server that may control several satellite (query) servers. To change the registry database, you must change the registry update site, which then 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 registry database (for example, the sec_rgy_pgo_add or sec_rgy_acct_lookup call).

Note: To bind to the registry update site use of the sec_rgy_site_bind_update API is recommended in place of sec_rgy_site_open_update.

Related Information
<table>
<thead>
<tr>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec_rgy_site_close</td>
</tr>
<tr>
<td>sec_rgy_site_get</td>
</tr>
</tbody>
</table>
sec_rgy_unix_getgrgid

Purpose
Returns a UNIX style group entry for the account matching the specified group.

Format
#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
class 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. The first element in the member_list[] array is zero (0), therefore to have a maximum of 10 members returned by this call, you must specify 9 as the size of the array.

Input/Output
item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The sec_rgy_unix_getgrgid API 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
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_unix_getgrgid API returns the next UNIX group structure that matches the input UNIX group ID. The structure is in the following form:
sec_rgy_unix_getgrgid

typedef struct {
    sec_rgy_name_t name;
    signed32 gid;
    sec_rgy_member_buf_t members;
} sec_rgy_unix_group_t;

The structure includes:
- 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.
sec_rgy_unix_getgrnam

Purpose
Returns a UNIX style group entry for the account matching the specified group name.

Format
#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_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
A signed 32-bit integer specifying the length of name in characters.

max_num_members
The maximum number of member 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_pgo_get_next API 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. Note that this field always accesses group entries starting from the first entry in the registry database for this API only.

Output

group_entry
A UNIX style group entry structure returned with information about the account matching name.

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
A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.
Usage

The `sec_rgy_unix_getgrnam` API 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:

```c
typedef struct {
    sec_rgy_name_t             name;
    signed32                   gid;
    sec_rgy_member_buf_t       members;
} sec_rgy_unix_group_t;
```

The structure includes:

- 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 supplied in source code form.
sec_rgy_unix_getpwnam

Purpose
Returns a UNIX style password entry for the account matching the specified name.

Format
#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) specifying the name of the principal whose name entry is desired.
name_length A signed 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 API 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 name.
status A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
    sec_rgy_no_more_entries The end of the list of entries has been reached.
    sec_rgy_bad_data The name supplied as entered was too large.
If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_unix_getpwnam API returns the next UNIX password 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;
sec_rgy_unix_getpwnam

The structure includes:

- 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 supplied in source code form.
sec_rgy_unix_getpwuid

Purpose
Returns a UNIX style password entry for the account matching the specified UID.

Format
#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 UNIX ID.

Input/Output
item_cursor An opaque pointer indicating a specific PGO item entry in the registry database. The sec_rgy_unix_getpwuid API 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 A pointer to the completion status. On successful completion, the routine returns error_status_ok. Otherwise, it returns one of the following errors:
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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_unix_getpwuid API looks up the next password 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 account's login name
sec_rgy_unix_getpwuid

- 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.
sec_rgy_wait_until_consistent

Purpose
Blocks updates while extending prior updates to registry replicas.

Format
#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, the routine returns error_status_ok. Otherwise, it returns one of the following errors:

- sec_rgy_server_unavailable
  The DCE Registry Server 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.

If you pass a NULL status pointer to a security API, the function may return immediately. Always pass a pointer to an allocated error_status_t variable where required.

Usage
The sec_rgy_wait_until_consistent API blocks updates to the master registry until all prior updates have been extended to all active registry replicas.

Return Values
The routine returns TRUE when all replicas have received the prior updates. It returns FALSE if at least one replica did not receive the updates.

Related Information
Routines
Registry API
sec_rgy_wait_until_consistent
Chapter 7. Kerberos

This section describes the routines that make up the Kerberos Version 5 application programming interface. It is oriented towards programmers who already have a basic familiarity with Kerberos and are in the process of including Kerberos authentication as part of applications being developed.

The Kerberos model is based in part on Needham and Schroeder's trusted third-part authentication protocol and on modifications suggested by Denning and Sacco. Kerberos Version 4 is publicly available and has seen wide use across the Internet. Kerberos Version 5 has evolved from Version 4 based on new requirements for features not available in Version 4.

Kerberos Basics

Kerberos performs authentication as a trusted third-party authentication service by using conventional shared secret key cryptography. Kerberos provides a means of verifying the identities of principals, without relying on authentication by the host operating system, without basing trust on host addresses, without requiring physical security of all the hosts on the network, and under the assumption that packets traveling along the network can be read, modified, and inserted at will.

The two methods for obtaining credentials, the initial ticket exchange and the ticket granting ticket exchange, use slightly different protocols and require different API routines. The basic difference an application programmer will see is that the initial ticket exchange does not require a ticket-granting ticket (TGT) but does require the client's secret key. Usually, the initial ticket exchange is for a TGT, and TGT exchanges are used from then on. In a TGT exchange, the TGT is sent as part of the request for a ticket and the reply is encrypted in the session key obtained from the TGT. Thus, once a user's password is used to obtain the initial TGT, it is not required for subsequent TGT exchanges to obtain additional tickets.

A ticket-granting ticket contains the Kerberos server (krbtgt/realm) as the server name. A service ticket contains the application server as the server name. A ticket-granting ticket is used to obtain service tickets. In order to obtain a service ticket for a server in another realm, the application must first obtain a ticket-granting ticket to the Kerberos server for that realm.

The Kerberos server reply consists of a ticket and a session key, encrypted either in the user's secret key or the TGT session key. The combination of a ticket and a session key is known as a set of credentials. An application client can use these credentials to authenticate to the application server by sending the ticket and an authenticator to the server. The authenticator is encrypted in the session key of the ticket and contains the name of the client, the name of the server, and the time the authenticator was created.

In order to verify the authentication, the application server decrypts the ticket using its service key, which is known only by the application server and the Kerberos server. Inside the ticket, the Kerberos server has placed the name of the client, the name of the server, a session key associated with the ticket, and some additional information. The application server then uses the ticket session key to decrypt the authenticator and verifies that the information in the authenticator matches the information in the ticket. The server also verifies that the authenticator timestamp is recent to prevent replay attacks (the default is 5 minutes). Since the session key was generated randomly by the Kerberos server and delivered encrypted in the service key and a key known only by the user, the application server can be confident that the user really is who he claims to be, by virtue of the fact that the user was able to encrypt the authenticator in the correct key.

To provide detection of both replay attacks and message stream modification attacks, the integrity of all the messages exchanged between principals can also be guaranteed by generating and transmitting a collision-proof checksum of the client's message, keyed with the session key. Privacy and integrity of the message exchanged between principals can be secured by encrypting the data to be passed using the session key.

Purpose of Realms

The Kerberos protocol is designed to operate across organizational boundaries. Each organization wishing to run a Kerberos server establishes its own realm. The name of the realm in which a client is registered is part of the client's name and can be used by the application server to decide whether to honor a request.

By establishing inter-realm keys, the administrators of two realms can allow a client authenticated in one realm to use its credentials in the other realm. The exchange of inter-realm keys registers the ticket-granting service of each realm as a principal in the other realm. A client is then able to obtain a ticket-granting ticket for the remote realm's ticket-granting service.
from its local ticket-granting service. Tickets issued to a service in the remote realm will indicate that the client was authenticated from another realm.

This method can be repeated to authenticate throughout an organization across multiple realms. To build a valid authentication path to a distant realm, the local realm must share an inter-realm key with the target realm or with an intermediate realm which communicates with either the target realm or with another intermediate realm.

Realms are typically organized hierarchically. Each realm shares a key with its parent and a different key with each child. If an inter-realm key is not directly shared by two realms, the hierarchical organization allows an authentication path to be easily constructed. If a hierarchical organization is not used, it may be necessary to consult some database in order to construct an authentication path between realms.

Although realms are typically hierarchical, intermediate realms may be bypassed to achieve cross-realm authentication through alternate authentication paths. It is important for the end-service to know which realms were transited when deciding how much faith to place in the authentication process. To facilitate this decision, a field in each ticket contains the names of the realms that were involved in authenticating the client.

**Fundamental Assumptions About the Environment**

Kerberos has certain limitations that should be kept in mind when designing security measures:

- Kerberos does not address "denial of service" attacks. There are places in these protocols where an intruder can prevent an application from participating in the proper authentication steps. Detection and solution of such attacks (some of which can appear to be not-uncommon “normal” failure modes for the system) is usually best left to human administrators and users.

- Principals must keep their secret keys secret. If an intruder somehow steals a principal’s key, it will be able to masquerade as that principal or impersonate any server to the legitimate principal.

- "Password guessing" attacks are not solved by Kerberos. If a user chooses a poor password, it is possible for an attacker to successfully mount an offline dictionary attack by repeatedly attempting to decrypt messages which are encrypted under a key derived from the user's password.

**Kerberos Terminology**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
<td>Verifying the claimed identity of a principal.</td>
</tr>
<tr>
<td>Authentication header</td>
<td>A record containing a ticket and an authenticator to be presented to a server as part of the authentication process.</td>
</tr>
<tr>
<td>Authentication path</td>
<td>A sequence of intermediate realms transited in the authentication process when communicating from one realm to another.</td>
</tr>
<tr>
<td>Authenticator</td>
<td>A record containing information that can be shown to have been recently generated using the session key known only by the client and the server.</td>
</tr>
<tr>
<td>Authorization</td>
<td>The process of determining whether a client may use a service, which objects the client is allowed to access, and the type of access allowed for each.</td>
</tr>
<tr>
<td>Ciphertext</td>
<td>The output of an encryption function. Encryption transforms plaintext into ciphertext.</td>
</tr>
<tr>
<td>Client</td>
<td>A process that makes use of a network service on behalf of a user. Note that in many cases a server may itself be a client of some other server (for example, a print server may be a client of a file server).</td>
</tr>
<tr>
<td>Credentials</td>
<td>A ticket plus the secret session key necessary to successfully use that ticket in an authentication exchange.</td>
</tr>
<tr>
<td>KDC</td>
<td>Key Distribution Center, a network service that supplies tickets and temporary session keys; or an instance of that service or the host on which it runs. The KDC processes both initial ticket and ticket-granting ticket requests. The initial ticket portion is sometimes referred to as the Authentication Service (AS) while the ticket-granting portion is sometimes referred to as the Ticket Granting Service (TGS).</td>
</tr>
<tr>
<td>Kerberos</td>
<td>The name given to Project Athena's authentication service, the protocol used by that service, and the code used to implement the authentication service. (This is also the name of the three-headed dog guarding Hades.)</td>
</tr>
</tbody>
</table>
Plaintext The input to an encryption function or the output of a decryption function. Decryption transforms ciphertext into plaintext.

Principal A uniquely named client or server instance that participates in a network communication.

Principal identifier The name used to uniquely identify each different principal.

Seal To encipher a record containing several fields in such a way that the fields cannot be individually replaced without either knowledge of the encryption key or leaving evidence of tampering.

Secret key An encryption key shared by a principal and the KDC, distributed outside the bounds of the system, with a long lifetime. In the case of a human user's principal, the secret key is derived from a password.

Server A particular principal which provides a resource to network clients.

Service A resource provided to network clients; often provided by more than one server.

Session key A temporary encryption key used between two principals, with a lifetime limited to the duration of a single login session.

Sub-session key A temporary encryption key used between two principals, selected and exchanged by the principals using the session key, and with a lifetime limited to the duration of a single association.

Ticket A record that helps a client authenticate itself to a server; it contains the client's identity, a session key, a timestamp, and other information, all sealed using the server's secret key. It only serves to authenticate a client when presented along with a fresh authenticator.

Using Kerberos Files

The Kerberos runtime uses three types of files during its processing: credentials cache, replay cache, and key table. Each type of file has a set of API routines to manage and manipulate the file.

Credentials Cache

The credentials cache holds Kerberos credentials (tickets, session keys, and other identifying information) in a semi-permanent store. The Kerberos runtime reads credentials from the cache as they are needed and stores new credentials in the cache as they are obtained. The relieves the application of the responsibility for managing the credentials itself.

Kerberos supports two types of credentials caches: FILE and MEMORY. The FILE credentials cache is maintained in a file and can be shared between applications. The MEMORY credentials cache is maintained in storage and can be accessed only by the application which created it. In addition, a FILE credentials cache will persist until it is deleted while a MEMORY credentials cache will not exist once the application has ended.

Replay Cache

The replay cache is used to detect duplicate requests. Each time a request is processed by the Kerberos runtime, an entry is made in the replay cache. If a later request is processed which matches an entry already in the replay cache, an error will be returned to the application program. The replay cache is periodically purged to remove stale entries (a stale entry occurs when the lifetime of the associated request has expired).

Kerberos supports two types of replay caches: dfl and mem. The dfl replay cache is maintained in a file and will persist across application restarts. The mem replay cache is maintained in memory and will not exist once the application has ended. The replay cache should not be shared between applications since this could result in false replay errors caused by different requests with the same timestamp.
**Key Table**

The key table is used to store encryption keys. This is normally used by server applications to provide the encryption keys for use by the Kerberos runtime when it needs to decrypt a request received from a client application. Each key has an associated version number, and the version is incremented each time the key is changed. When a service ticket is encrypted by the KDC, it uses the latest encryption key stored in the Kerberos database and records the key version number in the ticket. Then, when the ticket is presented to the server, the key version number is used to retrieve the proper key from the key table. This allows the server to change its key without invalidating existing tickets.

Kerberos supports two types of key tables: **FILE** and **WRFILE**. Both of these key table types refer to the same file-based key table. The difference is that a key table opened as **FILE** is read-only while a key table opened as **WRFILE** can be read and written. The key table can be shared by multiple applications.

**Using Kerberos Services**

The `krb5_context` opaque data type represents the current Kerberos context. Each application must have at least one Kerberos context. The Kerberos context contains configuration data obtained from the Kerberos configuration file as well as override values which have been set by the application. A single Kerberos context may be shared by multiple threads in the same process but may not be shared between processes. The `krb5_init_context()` API is used to create a Kerberos context.

The `krb5_auth_context` opaque data type represents a Kerberos authentication context. The Kerberos authentication context is used by message service routines. Each client-server connection must have its own authentication context since sequence numbers, encryption keys, checksums, and authenticators are stored in the context. If an authentication context is shared between threads, the application is responsible for providing concurrency control so that the context is not accessed by more than one thread at a time. The `krb5_auth_con_init()` routine is used to create a Kerberos authentication context.

Applications which want to use Kerberos services must include the `<krb5/krb5.h>` include file and must link with the libkrb5.a archive library. If the `<dce/dcekrbmsg.h>` include file is also included, it must appear after the `<krb5/krb5.h>` include file. This is necessary because the error codes returned by the Kerberos API routines will be the values defined in the `<krb5/krb5.h>` include file and not the values defined in the `<dce/dcekrbmsg.h>` include file. In addition, since the Kerberos runtime is packaged as a DLL, the application must be linked with the -Wl,DLL option and the `/usr/lib/EUVFDLL.x` sidefile must be included to define the imported entry points.

In order to properly handle code pages other than IBM-1047, the `setlocale()` routine must be called before any Kerberos API routines are called in order to set the proper code page.

**Configuration Files**

The default Kerberos configuration file is `<krb5/krb5.conf>`. For compatibility with DCE, the Kerberos runtime also supports the `<krb5/krb.conf>` configuration file. Entries in `<krb5/krb5.conf>` will override entries in `<krb5/krb.conf>`. The DCE configuration program creates and maintains the `<krb5/krb.conf>` file. The user is responsible for creating and maintaining the `<krb5/krb5.conf>` file if it is needed.

**<krb5/krb.conf>**

The `<krb5/krb.conf>` configuration file consists of two or more lines. The first line contains a single token which specifies the name of the local Kerberos realm. All lines after the first line contain two tokens: a realm name and the name of the host containing the KDC for the realm. For example, suppose we have two realms: dceprod.endicott.ibm.com and ends390.endicott.ibm.com. Then the `<krb5/krb.conf>` file for realm dceprod.endicott.ibm.com might look like the following:

```plaintext
    dceprod.endicott.ibm.com
    dceprod.endicott.ibm.com gandalf.endicott.ibm.com
    ends390.endicott.ibm.com
```

This says that the KDC for realm dceprod.endicott.ibm.com is located on gandalf.endicott.ibm.com while the KDC for realm ends390.endicott.ibm.com is located on allanon.endicott.ibm.com.
The `<krb5/krb5.conf>` configuration file is considerably more powerful than the `<krb5/krb.conf>` configuration file. The file is divided into sections. Each section contains one or more name/value pairs with one pair per line. The name and value are separated by an equal sign. The value may be either a character string or a group of name/value pairs. If a character string is specified, it consists of all characters starting with the first nonblank character following the equal sign and continuing until the last nonblank character on the line. The maximum length of a single line in the configuration file is 2046 bytes. Comment lines are denoted by a semi-colon in the first position of the line and blank lines are ignored.

A section name is enclosed in brackets and must appear on a line by itself. Group values are enclosed in braces with one group per line. The opening brace for a group may follow the equal sign or may be on a line by itself. The closing brace must be on a line by itself so that it will not be treated as part of the value string.

The default configuration file is `<krb5/krb5.conf>`. This can be changed by defining the `KRB5_CONFIG` environment variable. Multiple configuration files can be specified for the `KRB5_CONFIG` variable by separating the names with colons. If a named entry can have just one value, then the first occurrence of the name will be used. Otherwise, all of the entries for the same name will be grouped together in the order they are encountered.

The configuration file must be in code page 1047. To support other code pages, the following trigraphs can be used:

- ??( = left bracket
- ??) = right bracket
- ??< = left brace
- ??> = right brace

The following sections are supported:

- `[libdefaults]`
  This section provides defaults for the Kerberos runtime routines.

- `[realms]`
  This section defines each of the realms which can be reached from the local realm. For each realm, one or more KDC hosts must be defined. If this section is not defined, the realm information will be obtained from the `/krb5/krb.conf` configuration file.

- `[domain_realm]`
  This section defines the mapping between DNS names and Kerberos realm names.

- `[capath]`
  This section defines connection paths between realms. This section is not required if the Kerberos realms are arranged in a hierarchical configuration. Even in a hierarchical configuration, this section should be defined if there are direct connections between realms.

### [libdefaults]

clockskew

Specifies the maximum clock difference in seconds. The default is 300 (5 minutes). A Kerberos request will be rejected if the difference between the server time and the request timestamp exceeds the clock skew value.

kdc_req_checksum_type

Specifies the default checksum type for a KDC request as follows:

- `crc32`
- `rsa-md4`
- `rsa-md4-des`
- `descbc`
- `rsa-md5`
- `rsa-md5-des`

The checksum type must be `rsa-md4` in order to interoperate with earlier levels of the DCE security server. The default is `rsa-md5`.

ap_reqChecksumType

Specifies the default checksum type for an application request. The checksum type must be `rsa-md4` in order to interoperate with earlier levels of the DCE security server. The default is `rsa-md5`.  

Chapter 7. Kerberos 7-5
safe_checksum_type

Specifies the default checksum type for a safe request. The default is rsa-md5-des.

kdc_default_options

Specifies the default options used when requesting an initial ticket from the KDC as follows:

- $0x00000010 = KDC\_OPT\_RENEWABLE\_OK$ (DCE default)
- $0x10000000 = KDC\_OPT\_PROXIABLE$
- $0x40000000 = KDC\_OPT\_FORWARDABLE$

Multiple options may be specified by ORing the values together. The default is $0x00000010$.

kdc_timesync

Specifies whether the local time is to be synchronized with the KDC time. Specify 1 to synchronize the time and 0 to not synchronize the time. Do not specify 1 if the local system is running a time daemon which synchronizes the clock (for example, the dsd daemon provided with DCE). The default is 0.

cache_type

Specifies the format of the credentials cache file as an integer value between 1 and 4. Specify type 2 to share credentials cache files with DCE. The default is 2.

default_tkt_enctypes

Specifies one or more initial ticket encryption types separated by commas. The first encryption type specified will be used when generating random keys, so it must be an encryption type which is supported by all KDC servers which might be accessed by applications on the local system. The following encryption types are supported:

- des-cbc-crc
- des-cbc-md5
- des-cbc-raw

The first encryption type must be des-cbc-crc in order to interoperate with earlier levels of the DCE security server. The default is des-cbc-crc,des-cbc-md5.

default_tgs_enctypes

Specifies one or more ticket-granting ticket encryption types separated by commas. The first encryption type specified will be used when generating random keys, so it must be an encryption type which is supported by all KDC servers which might be accessed by applications on the local system. The first encryption type must be des-cbc-crc in order to interoperate with earlier levels of the DCE security server. The default is des-cbc-crc,des-cbc-md5.

rsa-md4-des-compat

Specifies whether to generate MD4 DES checksums in compatibility mode (1) or in strict mode (0). You must specify compatibility mode in order to interoperate with earlier levels of Kerberos V5.

rsa-md5-des-compat

Specifies whether to generate MD5 DES checksums in compatibility mode (1) or in strict mode (0). You must specify compatibility mode in order to interoperate with earlier levels of Kerberos V5.

default_keytab_name

Specifies the default key table name. The default is /krb5/v5srvtab.

default_realm

Specifies the default realm. If not specified, the default realm name is obtained from the /krb5/krb.conf file.

[realms]

realm

The realm value is a Kerberos realm name. The value is a group definition which defines the KDC servers for the realm. Each realm that can be contacted by applications on the local system must have an entry in the [realms] section of the configuration file. The group entry consists of one or more occurrences of the kdc name. The value for each kdc name entry is the host name and the port, separated by a colon. If the port is omitted, it will default to 88.

[domain_realm]

hostname

The hostname value is a DNS host name. The value is the name of the Kerberos realm which contains the specified host system.

.suffix

The .suffix value is the domain portion of a DNS host name. The value is the name of the Kerberos realm which contains host systems in the specified domain. A specific host name definition will take precedence over the domain specification.
realm

Each realm value is a Kerberos realm name and represents the starting point for a request. If the configuration file is not shared between systems, then the only realm that needs to be specified is the local realm. Otherwise, there needs to be a realm definition for each system sharing the configuration file. The value is a group definition which defines the target realms. If multiple hops are required to reach the target realm, there will be multiple entries defining each of the hops from the local realm to the target realm. If there is a direct connection between the local realm and the target realm, specify the hop as a period.

Sample <krb5/krb5.conf> Configuration File

; Numeric values can be specified as follows:
; $ddddddd = decimal number
; 0$ddddddd = octal number
; 0$ddddddd = hexadecimal number

; Checksum types
;  crc32
; rsa-md4 (required for DCE interoperability)
; rsa-md4-des
; descrc
; rsa-md5 (Kerberos V5 default)
; rsa-md5-des

; Encryption types
;  des-cbc-crc (required for DCE interoperability)
;  des-cbc-md5 (Kerberos V5 default)
;  des-cbc-raw

; KDC option codes
;  0x00000010 = KDC_OPT_RENEWABLE_OK (DCE default)
;  0x10000000 = KDC_OPT_PROXIABLE
;  0x40000000 = KDC_OPT_FORWARDABLE

[libdefaults]

; Maximum clock skew in seconds
;  DCE always uses 5 minutes for the clock skew
clockskew = 300

; Checksum used for KDC requests
kdc_req_checksum_type = rsa-md4

; Checksum used for application requests
ap_req_checksum_type = rsa-md4

; Checksum used for safe requests
safe_checksum_type = rsa-md5-des

; Kerberos V5 Beta 1 through Beta 5 computed the checksum
; incorrectly for rsa-md4-des and rsa-md5-des. Setting the
; compatibility mode to 1 will cause the old algorithm to be used
; when generating a checksum. Set the compatibility mode to 0
; to use the new algorithm.
rsa_md4_des_compat = 1
rsa_md5_des_compat = 1

; Default KDC options
kdc_default_options = 0x00000010

; Synchronize Kerberos library time with KDC server
kdc_timesync = 0
; Credentials cache file version
; Specify ccache_type=2 to share credentials cache files
; with older levels of DCE
ccache_type = 2

; Ticket encryption types (listed in priority order)
; Must include des-cbc-crc for interoperability with DCE
default_tkt_enctypes = des-cbc-crc,des-cbc-md5
default_tgs_enctypes = des-cbc-crc,des-cbc-md5

; The default_realm value will be obtained from the /krb5/krb.conf
; file if it is not specified here.
default_realm = dcesec4.endicott.ibm.com

; The default key table name. The KRB5_KTNAME environment variable
; will override this specification.
default_keytab_name = FILE:/krb5/v5srvtab

[realms]
; Realm definitions are not used in the DCE environment but are
; required in the stand-alone Kerberos environment. The KDC relation
; is repeated for each KDC in the realm. The realm definitions will
; be obtained from the /krb5/krb.conf file if they are not specified
; here.
dcesec4.endicott.ibm.com = {
    kdc = dcesec4.endicott.ibm.com:88
    kdc = dcecell5.endicott.ibm.com:88
}
ends39qzerodot.endicott.ibm.com = {
    kdc = allanon.endicott.ibm.com:88
}

[domain_realm]
; Convert host names to realm names. Individual host names may be
; specified. Domain suffixes may be specified with a leading period
; and will apply to all host names ending in that suffix.
dcesec4.endicott.ibm.com = dcesec4.endicott.ibm.com
dcecell5.endicott.ibm.com = dcesec4.endicott.ibm.com
dcetape.endicott.ibm.com = dceprod.endicott.ibm.com
dcedfs.endicott.ibm.com = dceprod.endicott.ibm.com
decnsv.endicott.ibm.com = ends390.endicott.ibm.com

[capaths]
; Configurable authentication paths which define the trust relationships
; between client and servers. Each entry represents a client realm
; and consists of the trust relationships for each server which can
; be accessed from that realm. A server may be listed multiple times
; if there are multiple trust relationships involved. Specify '.' for
; a direct connection.
;
; In this example, we have the following trust connections:
; dcesec4 is connected to ends390
; dceprod is connected to ends390
; pokgate is connected to ends390
; pokfvt is connected to pokgate
dcesec4.endicott.ibm.com = {
    dceprod.endicott.ibm.com = ends390.endicott.ibm.com
    pokgate.pok.ibm.com = ends390.endicott.ibm.com
    pokfvt.pok.ibm.com = ends390.endicott.ibm.com
    pokfvt.pok.ibm.com = pokgate.pok.ibm.com
}
Environment Variables

The following environment variables are used by the Kerberos runtime:

**KR5.CONFIG**  
One or more configuration file names separated by colons. The default configuration file is /krb5/krb5.conf.

**KR5CCNAME**  
Default name for the credentials cache file and is specified as type:name. The supported types are FILE and MEMORY.

**KR5_KTNAME**  
Default key table name. If not specified, the file specified by the default_keytab_name configuration entry will be used. If the configuration entry is not specified, the default is /krb5/v5srvtab.

**KR5_RCACHETYPE**  
Default replay cache type and defaults to dfl.

**KR5_RCACHENAME**  
Default replay cache name. If not specified, the Kerberos runtime will generate a name.

**KR5_RCACHEDIR**  
Default replay cache directory and defaults to /tmp.

**_EUV_SEC_KR5CCNAME_FILE**  
Specifies the name of the credentials cache pointer file. This file will contain the name of the default credentials cache. Use this environment variable instead of the KR5CCNAME environment variable if you are sharing the credentials cache file with the DCE runtime.
krb5_address_compare

Purpose
Compares two Kerberos addresses.

Format
#include <krb5/krb5.h>

krb5_boolean krb5_address_compare (
    krb5_context context,
    const krb5_address *addr1,
    const krb5_address *addr2)

Parameters
Input
context Specifies the Kerberos context.
addr1 Specifies the first address.
addr2 Specifies the second address.

Usage
The krb5_address_compare() routine compares two addresses and returns TRUE if they are the same and FALSE otherwise.
krb5_address_search

Purpose
Determines if an address is present in an address list.

Format
#include <krb5/krb5.h>

krb5_boolean krb5_address_search (  
    krb5_context context,  
    const krb5_address *addr,  
    krb5_address *const *addrlist)

Parameters
Input
context Specifies the Kerberos context.
addr Specifies the search address.
addrlist Specifies the address list as an array of addresses. The last entry in the array must be a NULL pointer. Specify NULL for this parameter if no address list is present.

Usage
The krb5_address_search() routine will determine if an address is present in an address list.

Return Values
The function return value will be TRUE if the address is found in the address list or if no address list was provided. The function return value will be FALSE otherwise.
**Purpose**
Releases an authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_free (  
    krb5_context context,  
    krb5_auth_context auth_context)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Usage**
The `krb5_auth_con_free()` routine releases an authentication context.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Generates local and remote network addresses.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_genaddrs (
    krb5_context context,
    krb5_auth_context auth_context,
    int fd, 
    int flags)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.
- `fd` Specifies the socket descriptor to be used.
- `flags` Specifies the address generation flags as follows:
  - `KRB5_AUTH_CONTEXT_GENERATE_LOCAL_ADDR` - Generate the local network address.
  - `KRB5_AUTH_CONTEXT_GENERATE_LOCAL_FULL_ADDR` - Generate the local network address and the local port.
  - `KRB5_AUTH_CONTEXT_GENERATE_REMOTE_ADDR` - Generate the remote network address.
  - `KRB5_AUTH_CONTEXT_GENERATE_REMOTE_FULL_ADDR` - Generate the remote network address and the remote port.

**Usage**
The `krb5_auth_con_genaddrs()` routine will generate the local and remote network addresses represented by a socket connection. These addresses will be stored in the authentication context and can be retrieved by the application by calling the appropriate API routine. The socket must have been created using the `AF_INET` address family. The socket must be in the connected state if the remote network address is to be generated.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_getaddrs

Purpose
Returns the local and remote network addresses stored in the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getaddrs (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_address **local_addr,  
    krb5_address **remote_addr)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.

Output
local_addr Returns the local network address. Specify NULL for this parameter if the local network address is not required. The return value will be NULL if the local network address has not been set. The krb5_free_address() routine should be called to release the address when it is no longer needed.
remote_addr Returns the remote network address. Specify NULL for this parameter if the remote network address is not required. The return value will be NULL if the remote network address has not been set. The krb5_free_address() routine should be called to release the address when it is no longer needed.

Usage
The krb5_auth_con_getaddrs() routine returns the local and remote network addresses stored in the authentication context.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Returns the authenticator from the authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getauthenticator (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_authenticator **authent)
```

**Parameters**
- **Input**
  - `context` Specifies the Kerberos context.
  - `auth_context` Specifies the authentication context.
- **Output**
  - `authent` Returns the authenticator. The `krb5_free_authenticator()` routine should be called to release the authenticator when it is no longer needed.

**Usage**
The `krb5_auth_con_getauthenticator()` routine returns the authenticator used during mutual authentication.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_getflags**

**Purpose**
Returns the current authentication context flags.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getflags (  
  krb5_context context,  
  krb5_auth_context auth_context,  
  krb5_int32 *flags)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Output**
- `flags` Returns the current flags. The following symbolic definitions are provided for the flag bits:
  - `KRB5_AUTH_CONTEXT_DO_TIME` - Use timestamps in messages.
  - `KRB5_AUTH_CONTEXT_RET_TIME` - Return timestamps to application.
  - `KRB5_AUTH_CONTEXT_DO_SEQUENCE` - Use sequence numbers in messages.
  - `KRB5_AUTH_CONTEXT_RET_SEQUENCE` - Return sequence numbers to application.

**Usage**
The `krb5_auth_con_getflags()` routine returns the current authentication context flags.

**Return Values**
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
### krb5_auth_con_getivector

**Purpose**
Returns the address of the initial vector in the authentication context.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getivector (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_pointer *ivec)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Output**
- `ivec` Returns the address of the initial vector. The authentication context still points to this vector, so any changes made to the vector will affect future data encryption operations performed using the authentication context.

**Usage**
The `krb5_auth_con_getivector()` routine returns the address of the initial vector used by the specified authentication context. The application can then use this address to change the contents of the initial vector.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_getkey

Purpose
Retrieves the encryption key stored in the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getkey (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_keyblock **keyblock)

Parameters

Input
context    Specifies the Kerberos context.
auth_context    Specifies the authentication context.

Output
keyblock    Returns a keyblock containing the encryption key. The krb5_free_keyblock() routine should be called to release the keyblock when it is no longer needed.

Usage
The krb5_auth_con_getkey() routine returns the current encryption key stored in the authentication context. This is normally the session key which was obtained from an application request message.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_getlocalseqnumber**

**Purpose**
Returns the local message sequence number from the authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getlocalseqnumber (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_int32 *seqnum)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Output**
- `seqnum` Returns the message sequence number.

**Usage**
The `krb5_auth_con_getlocalseqnumber()` routine returns the local message sequence number. Sequence numbers are used when generating messages if the `KRB5_AUTH_CONTEXT_DO_SEQUENCE` flag has been set in the authentication context.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_getlocalsubkey

**Purpose**
Returns the local subsession key from the authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getlocalsubkey (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_keyblock **keyblock)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Output**
- `keyblock` Returns the subsession key. The `krb5_free_keyblock()` routine should be called to release the keyblock when it is no longer needed.

**Usage**
The `krb5_auth_con_getlocalsubkey()` routine returns the local subsession key from the authentication context.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_getrcache**

**Purpose**
Retruns the replay cache for the authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getrcache (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_rcache *rcache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.

**Output**
- `rcache` Returns the replay cache handle.

**Usage**
The `krb5_auth_con_getrcache()` function returns the replay cache for the authentication context. A replay cache is used when processing a message in order to detect message replay. A replay cache must be set in the authentication context if message timestamps are being used.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_getremoteseqnumber

Purpose
Returns the remote message sequence number from the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getremoteseqnumber (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_int32 *seqnum)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.

Output
seqnum Returns the message sequence number.

Usage
The krb5_auth_con_getremoteseqnumber() routine returns the remote message sequence number. Sequence numbers are used when generating messages if the KRB5_AUTH_CONTEXT_DO_SEQUENCE flag has been set in the authentication context.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_getremotesubkey

Purpose
Returns the remote subsession key from the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getremotesubkey (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_keyblock **keyblock)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.

Output
keyblock Returns the subsession key. The krb5_free_keyblock() routine should be called to release the keyblock when it is no longer needed.

Usage
The krb5_auth_con_getremotesubkey() routine returns the remote subsession key from the authentication context.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Returns the local and remote network ports stored in the authentication context.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_getports (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_address **local_port,  
    krb5_address **remote_port)
```

**Parameters**

**Input**
- **context**
  Specifies the Kerberos context.
- **auth_context**
  Specifies the authentication context.

**Output**
- **local_port**
  Returns the local network port. Specify **NULL** for this parameter if the local network port is not required. The return value will be **NULL** if the local network port has not been set. The **krb5_free_address()** routine should be called to release the address when it is no longer needed.
- **remote_port**
  Returns the remote network port. Specify **NULL** for this parameter if the remote network port is not required. The return value will be **NULL** if the remote network port has not been set. The **krb5_free_address()** routine should be called to release the address when it is no longer needed.

**Usage**
The **krb5_auth_con_getports()** routine returns the local and remote network ports stored in the authentication context.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_init**

**Purpose**
Creates an authentication context.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_init (
    krb5_context context,
    krb5_auth_context *auth_context)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.

**Output**

- `auth_context` Returns the authentication context created by this call. The `krb5_auth_con_free()` routine should be called to release the authentication context when it is no longer needed.

**Usage**

The `krb5_auth_con_init()` routine creates an authentication context. An authentication context contains information relating to a single connection between two applications. The context is initialized to enable the use of the replay cache (KRB5_AUTH_CONTEXT_DO_TIME) but to disable the use of message sequence numbers. The `krb5_auth_con_setflags()` routine can be used to change these defaults.

The `krb5_auth_con_free()` routine should be used to release the authentication context when it is no longer needed.

The Kerberos runtime provides no concurrency control for the authentication context. If the application wants to use the same authentication context in multiple threads, it is the responsibility of the application to serialize access to the authentication context so that just a single thread is accessing the authentication context at any time. Because message sequence numbers are contained in the authentication context, this serialization will need to be extended to encompass the message exchange between the two applications. Otherwise, message sequence errors are liable to occur if the messages are delivered out of sequence.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_initivector

Purpose
Allocates and zeros the initial vector in the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_initivector (  
    krb5_context context,  
    krb5_auth_context auth_context)

Parameters
Input
context  
    Specifies the Kerberos context.

auth_context  
    Specifies the authentication context.

Usage
The krb5_auth_con_initivector() routine allocates and zeros the initial vector in the authentication context. The authentication context must already contain an encryption key which defines the type of encryption which will be used. The initial vector is used to initialize the encryption sequence each time a message is encrypted. This serves to generate different encrypted results for the same message contents and encryption key.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_set_req_cksumtype**

**Purpose**
Sets the checksum type used to generate an application request message.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_set_req_cksumtype(
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_cksumtype cksumtype)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `auth_context`: Specifies the authentication context.
- `cksumtype`: Specifies the checksum type as follows:
  - `CKSUMTYPE_CRC32`: DES CRC checksum
  - `CKSUMTYPE_DESCBC`: DES CBC checksum
  - `CKSUMTYPE_RSA_MD4`: MD4 checksum
  - `CKSUMTYPE_RSA_MD4_DES`: DES MD4 checksum
  - `CKSUMTYPE_RSA_MD5`: MD5 checksum
  - `CKSUMTYPE_RSA_MD5_DES`: DES MD5 checksum

**Usage**
The `krb5_auth_con_set_req_cksumtype()` routine sets the checksum type which will be used by the `krb5_mk_req()` routine. This overrides the default value set by the `ap_req_checksum_type` entry in the Kerberos configuration file.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_set_safe_cksumtype

Purpose
Sets the checksum type used to generate a signed application message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_set_safe_cksumtype (  
  krb5_context context,  
  krb5_auth_context auth_context,  
  krb5_cksumtype cksumtype)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.
cksumtype Specifies the checksum type as follows:
  - CKSUMTYPE_CRC32 - DES CRC checksum
  - CKSUMTYPE_DESCBC - DES CBC checksum
  - CKSUMTYPE_RSA_MD4 - MD4 checksum
  - CKSUMTYPE_RSA_MD4_DES - DES MD4 checksum
  - CKSUMTYPE_RSA_MD5 - MD5 checksum
  - CKSUMTYPE_RSA_MD5_DES - DES MD5 checksum

Usage
The krb5_auth_con_set_safe_cksumtype() routine sets the checksum type which will be used by the krb5_mk_safe() routine. This overrides the default value set by the ap_safe_checksum_type entry in the Kerberos configuration file.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Sets the local and remote address values in the authentication context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setaddrs (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_address *local_addr,  
    krb5_address *remote_addr)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `auth_context`: Specifies the authentication context.
- `local_addr`: Specifies the local network address. Specify `NULL` for this parameter if the local network address is not to be changed.
- `remote_addr`: Specifies the remote network address. Specify `NULL` for this parameter if the remote network address is not to be changed.

**Usage**
The `krb5_auth_con_setaddrs()` routine sets the local and remote network address values in the authentication context. These values are used when obtaining tickets and constructing authenticators.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_setflags

Purpose
Sets the authentication context flags.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setflags (  
    krb5Context context,  
    krb5AuthContext auth_context,  
    krb5Int32 flags)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.
flags Specifies the context flags. The following symbolic definitions are provided for the flag bits:
- KRB5_AUTH_CONTEXT_DO_TIME - Use timestamps in messages.
- KRB5_AUTH_CONTEXT_RET_TIME - Return timestamps to application.
- KRB5_AUTH_CONTEXT_DO_SEQUENCE - Use sequence numbers in messages.
- KRB5_AUTH_CONTEXT_RET_SEQUENCE - Return sequence numbers to application.

Usage
The krb5_auth_con_setflags() routine sets the authentication context flags.

Return Values
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_setivector

Purpose
Sets the initial vector in the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setivector (  
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_pointer ivec)

Parameters
Input
context            Specifies the Kerberos context.
auth_context        Specifies the authentication context.
ivec                Specifies the initial vector.

Usage
The krb5_auth_con_setivector() routine sets the initial vector in the authentication context. A copy is not made of the initial vector, so the application must not change or free the buffer specified by the ivec parameter until either a new initial vector is set or the authentication context is released. The initial vector is used to initialize the encryption sequence each time a message is encrypted. This serves to generate different encrypted results for the same message contents and encryption key.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_con_setports

Purpose
Sets the local and remote network ports in the authentication context.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setports (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    krb5_address *local_port,  
    krb5_address *remote_port)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.
local_port Specifies the local network port. Specify NULL for this parameter if the local network port is not to be changed.
remote_port Specifies the remote network port. Specify NULL for this parameter if the remote network port is not to be changed.

Usage
The krb5_auth_con_setports() routine sets the local and remote network ports in the authentication context.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Sets the replay cache for the authentication context.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setrcache (
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_rcache rcache)
```

**Parameters**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>Specifies the Kerberos context.</td>
</tr>
<tr>
<td>auth_context</td>
<td>Specifies the authentication context.</td>
</tr>
<tr>
<td>rcache</td>
<td>Specifies the replay cache handle.</td>
</tr>
</tbody>
</table>

**Usage**

The `krb5_auth_con_setrcache()` function sets the replay cache for the authentication context. A replay cache is used when processing a message in order to detect message replay. A replay cache must be set in the authentication context if message timestamps are being used. The `krb5_rc_default()` and `krb5_rc_resolve()` routines can be used to obtain a replay cache handle.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_auth_con_setuserkey**

**Purpose**
Sets the user key in the authentication context.

**Format**

```
#include <krb5/krb5.h>

krb5_error_code krb5_auth_con_setuserkey ( 
    krb5_context context, 
    krb5_auth_context auth_context, 
    krb5_keyblock *keyblock)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.
- `keyblock` Specifies the user key.

**Usage**
The `krb5_auth_con_setuserkey()` routine sets the user key in the authentication context. This is only useful prior to calling the `krb5_rd_req()` routine for user-to-user authentication where the server has the key and needs to use it to decrypt the incoming request. Once the request has been decrypted, this key is no longer necessary and will be replaced in the authentication context with the session key obtained from the decoded request.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_auth_to_rep

Purpose
Converts Kerberos authenticator to replay entry.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_auth_to_rep (  
    krb5_context context,  
    krb5_tkt_authent *authent,  
    krb5_donot_replay *replay)

Parameters
Input
context Specifies the Kerberos context.
authent Specifies the Kerberos authenticator.

Output
replay Returns the replay entry data. The krb5_rc_free_entry_contents() routine should be called to release the entry data when it is no longer needed.

Usage
The krb5_auth_to_rep() routine extracts information from ticket authentication data and builds a replay cache entry. This entry can then be used to check for ticket replay by calling the krb5_rc_store() routine to save the entry in the replay cache.

Return Values
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_build_principal

Purpose
Builds a Kerberos principal from component strings.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_build_principal (  
    krb5_context context,  
    krb5_principal *ret_princ,  
    int realm_len,  
    krb5_const char *realm,  
    char *name1, name2, ...)

Parameters
Input
context Specifies the Kerberos context.
realm_len Specifies the length of the name name.
realm Specifies the realm name.
name1, name2 ... One or more name components. The end of the components is indicated by specifying NULL for the parameter.

Output
ret_princ Returns the Kerberos principal. The krb5_free_principal() routine should be called to release the principal when it is no longer needed.

Usage
The krb5_build_principal() routine creates a Kerberos principal from its component strings.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.

For example, to create the principal bambi/admin@forest, make the following call:

retval = krb5_build_principal(context, &princ, 6, "forest",  
                                "bambi", "admin", NULL);
krb5_build_principal_ext

Purpose
Builds a Kerberos principal from component strings.

Format

```c
#include <krb5/krb5.h>

krb5_error_code krb5_build_principal_ext(
    krb5_context context,
    krb5_principal *ret_princ,
    int realm_len,
    krb5_const char *realm,
    int name1_len,
    char *name1,
    int name2_len,
    char *name2, ...
);
```

Parameters

**Input**
- `context` Specifies the Kerberos context.
- `realm_len` Specifies the length of the name name.
- `realm` Specifies the realm name.
- `name1_len name1 ...` Specifies one or more name components. Each component consists of its length followed by its value. The end of the components is indicated by specifying a length of zero.

**Output**
- `ret_princ` Returns the Kerberos principal. The `krb5_free_principal()` routine should be called to release the principal when it is no longer needed.

Usage

The `krb5_build_principal_ext()` routine creates a Kerberos principal from its component strings. This routine is similar to the `krb5_build_principal()` routine except that the name component lengths are explicitly specified on the function call.

Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.

For example, to create the principal `bambi/admin@forest`, make the following call:

```c
retval = krb5_build_principal_ext(context, &princ, 6, "forest",
    5, "bambi", 5, "admin", 0);
```
**krb5_build_principal_ext_va**

**Purpose**
Builds a Kerberos principal from component strings.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_build_principal_ext_va (  
    krb5_context context,  
    krb5_principal *ret_princ,  
    int realm_len,  
    krb5_const char *realm,  
    va_list ap)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `realm_len` Specifies the length of the name name.
- `realm` Specifies the realm name.
- `ap` Specifies a variable argument list consisting of name lengths and character pointers which specify one or more name components. The end of the components is indicated by specifying a name length of zero.

**Output**
- `ret_princ` Returns the Kerberos principal. The **krb5_free_principal()** routine should be called to release the principal when it is no longer needed.

**Usage**
The **krb5_build_principal_ext_va()** routine creates a Kerberos principal from its component strings. It is similar to the **krb5_build_principal_ext()** routine except the name components are specified as a variable argument list instead of as discrete parameters on the function call.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.

For example, assume we have a function **my_func** which is called with a list of names. It could generate a Kerberos principal from these names as follows:
#include <stdarg.h>
#include <krb5/krb5.h>

krb5_error_code my_func(int realm_len, char *realm, ...) {
    va_list ap;
    krb5_error_code retval;

    va_start(ap, realm);
    retval = krb5_build_principal_ext_va(context, &princ,
        realm_len, realm, ap);
    va_end(ap);
    return retval;
}

int main(int argc, char *argv[]) {
    my_func(6, "forest", 5, "bambi", 5, "admin", 0);
    return 0;
}
krb5_build_principal_va

Purpose
Builds a Kerberos principal from component strings.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_build_principal_va (  
    krb5_context  context,  
    krb5_principal *ret_princ,  
    int  realm_len,  
    krb5_const char *realm,  
    va_list  ap)

Parameters
Input
context   Specifies the Kerberos context.
realm_len  Specifies the length of the name name.
realm  Specifies the realm name.
ap  Specifies a variable argument list consisting of character pointers which specify one or more name components. The end of the components is indicated by specifying NULL for the parameter.

Output
ret_princ  Returns the Kerberos principal. The krb5_free_principal() routine should be called to release the principal when it is no longer needed.

Usage
The krb5_build_principal_va() routine creates a Kerberos principal from its component strings. It is similar to the krb5_build_principal() routine except the name components are specified as a variable argument list instead of as discrete parameters on the function call.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.

For example, assume we have a function my_func which is called with a list of names. It could generate a Kerberos principal from these names as follows:
#include <stdarg.h>
#include <krb5/krb5.h>

krb5_error_code my_func(char *realm, ...) {
    va_list ap;
    krb5_error_code retval;

    va_start(ap, realm);
    retval = krb5_build_principal_va(context, &princ, strlen(realm), realm, ap);
    va_end(ap);
    return retval;
}

int main(int argc, char *argv[]) {
    my_func("forest", "bambi", "admin", NULL);
    return 0;
}
krb5_cc_close

krb5_cc_close

Purpose
Closes a credentials cache.

Format
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_close (  
    krb5_context context,  
    krb5_ccache ccache)
```

Parameters

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>context</code></td>
<td>Specifies the Kerberos context.</td>
</tr>
<tr>
<td><code>ccache</code></td>
<td>Specifies the credentials cache handle.</td>
</tr>
</tbody>
</table>

Usage
The `krb5_cc_close()` routine closes a credentials cache. The cache handle may not be used once this routine completes.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_cc_default**

**Purpose**
Resolves the default credentials cache.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_cc_default (
    krb5_context context,
    krb5_ccache *ccache)
```

**Parameters**

**Input**

*context* 
Specifies the Kerberos context.

**Output**

*ccache* 
Returns the credentials cache handle.

**Usage**
The `krb5_cc_default()` routine resolves the default credentials cache and returns a handle which can be used to access the cache. This is equivalent to calling the `krb5_cc_resolve()` routine with the name returned by the `krb5_cc_default_name()` routine.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_default_name

Purpose
Returns the default credentials cache name.

Format
#include <krb5/krb5.h>

char *krb5_cc_default_name (  
    krb5_context context)

Parameters
Input
context Specifies the Kerberos context.

Usage
The krb5_cc_default_name() routine returns the name of the default credentials cache for the current user. If the KRB5CCNAME environment variable is set, this will be the name of the default cache. Otherwise, the name will be obtained from the krb5ccname file. If this file does not exist or if there is no default credentials cache name set in the file, a new credentials cache name will be generated.

Return Values
The function return value will be NULL if an error occurred. Otherwise, it will be the address of the default credentials cache name. This will be a pointer to read-only storage and must not be freed by the application.
**krb5_cc_destroy**

**Purpose**
Deletes a credentials cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_destroy (  
    krb5_context context,  
    krb5_ccache ccache)
```

**Parameters**

**Input**
- *context*  
  Specifies the Kerberos context.
- *ccache*  
  Specifies the credentials cache handle.

**Usage**
The `krb5_cc_destroy()` routine closes and deletes a credentials cache. The cache handle may not be used after this routine completes.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_end_seq_get

**Purpose**
Ends the sequential reading of the credentials cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_end_seq_get (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_cc_cursor *cursor)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.

**Input/Output**
- `cursor` Specifies the cursor created by the `krb5_cc_start_seq_get()` routine.

**Usage**
The `krb5_cc_end_seq_get()` routine unlocks the credentials cache and releases the cursor. The cursor may not be used once `krb5_cc_end_seq_get()` has completed.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_cc_generate_new**

**Purpose**
Generates a new credentials cache.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_generate_new (
    krb5_context context,
    const char *type,
    krb5_ccache *ccache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `type` Specifies the credentials cache type (for example, `FILE`).

**Output**
- `ccache` Returns the credentials cache handle. The `krb5_cc_close()` or `krb5_cc_destroy()` routine should be called to release the handle when it is no longer needed.

**Usage**
The `krb5_cc_generate_new()` routine will create a new credentials cache with a unique name. The `krb5_cc_initialize()` function must be called to set the cache principal before storing any credentials in the cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_get_name

Purpose
Returns the credentials cache name.

Format
#include <krb5/krb5.h>

char *krb5_cc_get_name (  
    krb5_context context,  
    krb5_ccache ccache)  

Parameters
Input
context Specifies the Kerberos context.
cache Specifies the credentials cache handle.

Usage
The krb5_cc_get_name() routine returns the name of the credentials cache. The returned name will not include the credentials cache type prefix.

Return Values
The function return value will be the address of the credentials cache name. This is a read-only value and must not be freed by the application.
**krb5_cc_get_principal**

**Purpose**
Returns the principal associated with the credentials cache.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_get_principal (  
    krb5_context context,       
    krb5_ccache ccache,         
    krb5_principal *principal)  
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.

**Output**
- `principal` Returns the principal. The `krb5_free_principal()` routine should be called to release the principal when it is no longer needed.

**Usage**
The `krb5_cc_get_principal()` routine returns the principal associated with the credentials cache. The principal name is set by the `krb5_cc_initialize()` routine. This is the default client principal for tickets stored in the credentials cache.

**Return Values**
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_get_type

Purpose
Returns the credentials cache type.

Format
#include <krb5/krb5.h>

char *krb5_cc_get_type (  
    krb5_context context,  
    krb5_ccache ccache)  

Parameters

Input
context Specifies the Kerberos context.
ccache Specifies the credentials cache handle.

Usage
The krb5_cc_get_type() routine returns the credentials cache type.

Return Values
The function return value will be the address of the credentials cache type. This is a read-only value and must not be freed by the application.
**krb5_cc_initialize**

**Purpose**
Initializes a credentials cache.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_initialize (
    krb5_context context,
    krb5_ccache ccache,
    krb5_principal principal)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.
- `principal` Specifies the default principal for the cache.

**Usage**
The `krb5_cc_initialize()` routine will initialize a credentials cache. Any existing credentials will be discarded and the principal name for the cache will be set to the value specified. The principal name is the default client name for tickets which will be placed into the cache. A new cache must be initialized before tickets can be stored in the cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_next_cred

Purpose
Returns the next entry from the credentials cache.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_cc_next_cred (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_cc_cursor *cursor,  
    krb5_creds *creds)

Parameters

Input
context Specifies the Kerberos context.
ccache Specifies the credentials cache handle.

Input/Output
cursor Specifies the cursor created by the krb5_cc_start_seq_get() routine.  
The cursor will be updated upon successful completion of this routine.

Output
creds Returns the contents of the cache entry. The krb5_free_cred_contents() routine should be called to release the credentials contents when they are no longer needed.

Usage
The krb5_cc_next_cred() routine reads the next entry from the credentials cache and returns it to the application. The krb5_cc_start_seq_get() routine must be called to begin the sequential read operation. The krb5_cc_next_cred() routine is then called repeatedly to read cache entries. Finally, the krb5_cc_end_seq_get() routine is called when no more entries are to be read.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_cc_register**

**Purpose**
Defines a new credentials cache type.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_register (  
    krb5_context context,  
    krb5_cc_ops *ops,  
    krb5_boolean override)
```

**Parameters**

**Input**
- `context` specifies the Kerberos context.
- `ops` specifies the credentials cache operations vector. This vector defines the routines which will be called to perform the various credentials cache operations for the new cache type.
- `override` specifies whether to override an existing definition for the same type. An error will be returned if the type is already registered and `FALSE` is specified for this parameter.

**Usage**
The `krb5_cc_register()` routine registers a new credentials cache type. Once the new type is registered, it can be used by any thread in the current process. The type is not known outside the current process and will no longer be registered when the application ends.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_cc_remove_cred

Purpose
Removes an entry from the credentials cache.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_cc_remove_cred (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_flags flags,  
    krb5_creds *mcreds)

Parameters

Input
context Specifies the Kerberos context.
ccache Specifies the credentials cache handle.
flags Specifies the search flags which are used to determine whether a particular cache entry should be removed. The following symbolic definitions are provided for the various flags and should be ORed together to set the desired search flags:

- **KRB5_TC_MATCH_TIMES** - The `renew_till` and `endtime` values in the cache entry must be greater than the values in the match credentials. A time value will be ignored if it is zero.
- **KRB5_TC_MATCH_IS_SKEY** - The `is_skey` flag in the cache entry must be the same as the `is_skey` flag in the match credentials.
- **KRB5_TC_MATCH_FLAGS** - All of the flags set in the match credentials must also be set in the cache entry.
- **KRB5_TC_MATCH_TIMES_EXACT** - The time fields in the cache entry must exactly match the time fields in the match credentials.
- **KRB5_TC_MATCH_FLAGS_EXACT** - The flags in the cache entry must exactly match the flags in the match credentials.
- **KRB5_TC_MATCH_AUTHDATA** - The authorization data in the cache entry must be identical to the authorization data in the match credentials.
- **KRB5_TC_MATCH_SRV_NAMEONLY** - Only the name portion of the server principal in the cache entry needs to match the server principal in the match credentials. The realm values may be different. If this flag is not set, the complete principal name must match.
- **KRB5_TC_MATCH_2ND_TKT** - The second ticket in the cache entry must exactly match the second ticket in the match credentials.
- **KRB5_TC_MATCH_KTYPE** - The encryption key type in the cache entry must match the encryption key type in the match credentials.

mcreds Specifies the match credentials. Fields from these credentials will be matched with fields in the cache entries based upon the search flags. The client and server principals must always be set in the match credentials no matter what search flags are specified.

Usage
The krb5_cc_remove_cred() routine removes matching entries from the credentials cache. The client principal must always match. The **KRB5_TC_MATCH_SRV_NAMEONLY** flag controls how much of the server principal must match.
Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.

*Note:* The `krb5_cc_remove_cred()` routine is not supported for the `FILE` and `MEMORY` cache types and will return an error code of `KRB5_CC_OP_NOT_SUPPORTED`. 
**krb5_cc_resolve**

**Purpose**
Resolves a credentials cache name.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_resolve (    krb5_context context,    char *cache_name,    krb5_ccache *ccache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `cache_name` Specifies the credentials cache name in the format `type:name`. The type must be a registered credentials cache type and the name must uniquely identify a particular credentials cache of the specified type.

**Output**
- `ccache` Returns the credentials cache handle.

**Usage**
The `krb5_cc_resolve()` routine resolves a credentials cache name and returns a handle which can be used to access the cache. The Kerberos runtime supports two credentials cache types: `FILE` and `MEMORY`. Additional credentials cache types can be registered by the application by calling the `krb5_cc_register()` routine. If no type is specified, the default is `FILE`.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_cc_retrieve_cred**

**Purpose**
Retrieves a set of credentials from the cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_retrieve_cred (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_flags flags,  
    krb5_creds *mcreds,  
    krb5_creds *creds)
```

**Parameters**

**Input**
- **context**
  Specifies the Kerberos context.
- **ccache**
  Specifies the credentials cache handle.
- **flags**
  Specifies the search flags which are used to determine whether a particular cache entry should be returned to the caller. The following symbolic definitions are provided for the various flags and should be ORed together to set the desired search flags:

  - **KRB5_TC_MATCH_TIMES** - The `renew_till` and `endtime` values in the cache entry must be greater than the values in the match credentials. A time value will be ignored if it is zero.
  - **KRB5_TC_MATCH_IS_SKEY** - The `is_skey` flag in the cache entry must be the same as the `is_skey` flag in the match credentials.
  - **KRB5_TC_MATCH_FLAGS** - All of the flags set in the match credentials must also be set in the cache entry.
  - **KRB5_TC_MATCH_TIMES_EXACT** - The time fields in the cache entry must exactly match the time fields in the match credentials.
  - **KRB5_TC_MATCH_FLAGS_EXACT** - The flags in the cache entry must exactly match the flags in the match credentials.
  - **KRB5_TC_MATCH_AUTHDATA** - The authorization data in the cache entry must be identical to the authorization data in the match credentials.
  - **KRB5_TC_MATCH_SRV_NAMEONLY** - Only the name portion of the server principal in the cache entry needs to match the server principal in the match credentials. The realm values may be different. If this flag is not set, the complete principal name must match.
  - **KRB5_TC_MATCH_2ND_TKT** - The second ticket in the cache entry must exactly match the second ticket in the match credentials.
  - **KRB5_TC_MATCH_KTYPE** - The encryption key type in the cache entry must match the encryption key type in the match credentials.

**Output**
- **creds**
  Returns the contents of the matched cache entry. The `krb5_free_cred_contents()` routine should be called to release the credentials contents when they are no longer needed.
**Usage**

The `krb5_cc_retrieve_cred()` routine searches the credentials cache and returns an entry which matches the credentials specified. The client principal must always match. The `KRB5_TC_MATCH_SRV_NAMEONLY` flag controls how much of the server principal must match.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**kb5_cc_set_flags**

**Purpose**
Sets processing flags for the credentials cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_set_flags (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_flags flags)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.
- `flags` Specifies the flags. The allowable flags will depend upon the cache type.

**Usage**
The `kb5_cc_set_flags()` routines sets the processing flags for a credentials cache. The interpretation of the flags is dependent upon the cache type.

The MEMORY cache type does not support the `kb5_cc_set_flags()` routine and will return `KRB5_CC_OP_NOT_SUPPORTED`.

The FILE cache type supports just the `KRB5_TC_OPENCLOSE` flag. If this flag is specified, the credentials cache file will be opened each time a credentials cache routine is called and then closed before returning to the caller (this is the default behavior if the `kb5_cc_set_flags()` routine is not called). If this flag is not specified, the credentials cache file is opened and remains open until the credentials cache is closed by the `kb5_cc_close()` or `kb5_cc_destroy()` routine. An exception is for the sequential read routines. Regardless of the `KRB5_TC_OPENCLOSE` flag setting, the credentials cache file is opened when the `kb5_cc_start_seq_get()` routine is called and remains open until the `kb5_cc_end_seq_get()` routine is called.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Starts sequentially retrieving entries from the credentials cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_start_seq_get (
    krb5_context context,
    krb5_ccache ccache,
    krb5_cc_cursor *cursor)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.

**Output**
- `cursor` Returns the cursor. The `krb5_cc_end_seq_get()` routine should be called to release the cursor at the completion of the sequential read operation.

**Usage**
The `krb5_cc_start_seq_get()` routine prepares for sequentially reading entries in the credentials cache. The `krb5_cc_next_cred()` routine is called repeatedly to retrieve each successive cache entry. The `krb5_cc_end_seq_get()` routine is called at the completion of the read operation.

The credentials cache is locked when the `krb5_cc_start_seq_get()` routine is called and remains locked until the `krb5_cc_end_seq_get()` routine is called. Write access to the cache by other processes and threads will be blocked until the cache is unlocked. After the `krb5_cc_start_seq_get()` routine has been called, the current thread may not call any other credentials cache functions except `krb5_cc_next_cred()` and `krb5_cc_end_seq_get()` for the specified cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_cc_store_cred**

**Purpose**
Stores a new set of credentials in the cache.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_cc_store_cred (
    krb5_context context,
    krb5_ccache ccache,
    krb5_creds *creds)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache handle.
- `creds` Specifies the Kerberos credentials.

**Usage**

The `krb5_cc_store_cred()` routine stores a new set of Kerberos credentials in the credentials cache. Existing credentials for the same client/server pair are not removed, even if they are expired. Credentials are stored first-in, first-out which means that newer credentials will be retrieved after older credentials.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_address**

**Purpose**
Copies a Kerberos address to a new structure.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_address (
    krb5_context context,
    const krb5_address *from_addr,
    krb5_address **to_addr)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `from_addr` Specifies the address to be copied.

**Output**

- `to_addr` Returns the new `krb5_address` structure. The `krb5_free_address()` routine should be called to release the address when it is no longer needed.

**Usage**
The `krb5_copy_address()` routine makes a copy of a Kerberos address structure.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_addresses**

**Purpose**
Copies an array of Kerberos addresses.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_addresses (
    krb5_context context,
    const krb5_address **from_addrs,
    krb5_address ***to_addrs)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `from_addrs` Specifies the array of addresses to be copied. The last array entry must be a NULL pointer.

**Output**
- `to_addrs` Returns the new `krb5_address` array. The `krb5_free_addresses()` routine should be called to release the address array when it is no longer needed.

**Usage**
The `krb5_copy_addresses()` routine makes a copy of an array of Kerberos address structures.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_authdata**

**Purpose**
Copies an array of authorization data structures.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_authdata (
    krb5_context context,
    krb5_authdata *const *from_authdata,
    krb5_authdata ***to_authdata)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `from_authdata`: Specifies the array of `krb5_authdata` structures. The last array entry must be a NULL pointer.

**Output**
- `to_authdata`: Returns the new array of `krb5_authdata` structures. The `krb5_free_authdata()` routine should be called to release the array when it is no longer needed.

**Usage**
The `krb5_copy_authdata()` routine copies an array of `krb5_authdata` structures.

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_authenticator**

**Purpose**
Copies a Kerberos authenticator.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_authenticator (  
    krb5_context context,  
    const krb5_authenticator *from_authent,  
    krb5_authenticator **to_authent)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `from_authent` Specifies the authenticator to be copied.

**Output**
- `to_authent` Returns the copied authenticator. The `krb5_free_authenticator()` routine should be called to release the authenticator when it is no longer needed.

**Usage**
The `krb5_copy_authenticator()` routine copies a Kerberos authenticator.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Copies a Kerberos checksum.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_checksum (
    krb5_context context,
    const krb5_checksum *from_cksum,
    krb5_checksum **to_cksum)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `from_cksum` Specifies the checksum to be copied.

**Output**
- `to_cksum` Returns the copied checksum. The `krb5_free_checksum()` routine should be called to release the checksum when it is no longer needed.

**Usage**
The `krb5_copy_checksum()` routine copies a Kerberos checksum.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_creds**

**Purpose**
Copies Kerberos credentials.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_copy_creds (
    krb5_context context,
    const krb5_creds *from_creds,
    krb5_creds **to_creds)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `from_creds` Specifies the credentials to be copied.

**Output**
- `to_creds` Returns the copied credentials. The `krb5_free_creds()` routine should be called to release the credentials are no longer needed.

**Usage**
The `krb5_copy_creds()` routine copies Kerberos credentials.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
Purpose
Copies a Kerberos data object.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_copy_data (
    krb5_context context,
    const krb5_data *from_data,
    krb5_data **to_data)

Parameters
Input
custom Specifies the Kerberos context.
from_data Specifies the data object to be copied.

Output
to_data Returns the copied data object. The krb5_free_data() routine should be called to release the data object when it is no longer needed.

Usage
The krb5_copy_data() routine copies a Kerberos data object which is represented by a krb5_data structure.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_copy_keyblock

Purpose
Copies a Kerberos keyblock.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_copy_keyblock (
    krb5_context context,
    const krb5_keyblock *from_keyblock,
    krb5_keyblock **to_keyblock)

Parameters
Input
context Specifies the Kerberos context.
from_keyblock Specifies the keyblock to be copied.

Output
to_keyblock Returns the copied keyblock. The krb5_free_keyblock() routine should be called to release the
keyblock when it is no longer needed.

Usage
The krb5_copy_keyblock() routine copies a Kerberos keyblock.

Return Values
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
**krb5_copy_keyblock_contents**

**Purpose**
Copies the contents of a Kerberos keyblock.

**Format**

```
#include <krb5/krb5.h>

krb5_error_code krb5_copy_keyblock_contents (
    krb5_context context,
    const krb5_keyblock *from_keyblock,
    krb5_keyblock *to_keyblock)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `from_keyblock`: Specifies the keyblock to be copied.

**Output**
- `to_keyblock`: Returns the contents of the input keyblock. The `krb5_free_keyblock_contents()` routine should be called to release the contents of the keyblock when it is no longer needed.

**Usage**
The `krb5_copy_keyblock_contents()` routine copies the contents of a Kerberos keyblock into an existing keyblock. The current contents of the output keyblock are not released before performing the copy.

**Return Values**
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_copy_principal

Purpose
Copies a Kerberos principal.

Format
```c
#include <krb5/krb5.h>

krb5_error_code krb5_copy_principal (  
    krb5_context context,  
    krb5_const_principal from_princ,  
    krb5_principal *to_princ)
```

Parameters

**Input**
- `context` Specifies the Kerberos context.
- `from_princ` Specifies the principal to be copied.

**Output**
- `to_princ` Returns the copied principal. The `krb5_free_principal()` routine should be called to release the principal when it is no longer needed.

Usage
The krb5_copy_principal() routine copies a Kerberos principal.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_copy_ticket

Purpose
Copies a Kerberos ticket.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_copy_ticket ( 
    krb5_context context, 
    const krb5_ticket *from_ticket, 
    krb5_ticket **to_ticket)

Parameters

Input
context Specifies the Kerberos context.
from_ticket Specifies the ticket to be copied.

Output
to_ticket Returns the copied ticket. The krb5_free_ticket() routine should be called to release the ticket when it is no longer needed.

Usage
The krb5_copy_ticket() routine copies a Kerberos ticket.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_free_address**

**Purpose**
Releases the storage assigned to a Kerberos address.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_address (  
    krb5_context context,  
    krb5_address *addr)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `addr` Specifies the `krb5_address` to be released.

**Usage**
The `krb5_free_address()` routine will release the storage assigned to the contents of a `krb5_address` structure and then it will release the `krb5_address` structure itself.
krb5_free_addresses

Purpose
Releases the storage assigned to an array of Kerberos addresses.

Format
#include <krb5/krb5.h>

void krb5_free_addresses (  
    krb5_context context,  
    krb5_address **addrs)

Parameters
Input
context  Specifies the Kerberos context.
addrs   Specifies the array to be released. The last entry in the array must be a NULL pointer.

Usage
The krb5_free_addresses() routine releases the storage assigned to an array of krb5_address structures. Each krb5_address structure will be released and then the pointer array itself will be released.
**Purpose**
Releases the storage assigned to the decrypted portion of an AP_REP message.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_ap_rep_enc_part (
    krb5_context context,
    krb5_ap_rep_enc_part enc_part)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `enc_part` Specifies the reply to be released.

**Usage**
The `krb5_free_ap_rep_enc_part()` routine releases the storage assigned to the decrypted reply returned by the `krb5_rd_rep()` routine.
**krb5_free_authdata**

**Purpose**
Releases the storage assigned to an array of authentication data.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_authdata (
    krb5_context context,
    krb5_authdata **authdata)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `authdata` Specifies the array to be released. The last entry in the array must be a NULL pointer.

**Usage**
The `krb5_free_authdata()` routine releases the storage assigned to an array of `krb5_authdata` structures. Each `krb5_authdata` structure will be released and then the pointer array itself will be released.
**Purpose**
Releases the storage assigned to an authenticator.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_authenticator (  
    krb5_context context,  
    krb5_authenticator *authent)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `authent` Specifies the `krb5_authenticator` to be released.

**Usage**
The `krb5_free_authenticator()` routine releases the storage assigned to the contents of a `krb5_authenticator` structure and then it releases the `krb5_authenticator` structure itself.
**Purpose**
Releases the storage assigned to the contents of an authenticator.

**Format**

```c
#include <krb5/krb5.h>

void krb5_free_authenticator_contents (  
    krb5_context context,  
    krb5_authenticator *authent)
```

**Parameters**

**Input**

- `context`: Specifies the Kerberos context.
- `authent`: Specifies the `krb5_authenticator` to be released.

**Usage**
The `krb5_free_authenticator_contents()` routine releases the storage assigned to the contents of a `krb5_authenticator` structure. Unlike the `krb5_free_authenticator()` routine, the `krb5_free_authenticator_contents()` routine does not free the `krb5_authenticator` structure.
**krb5_free_checksum**

**Purpose**
Releases the storage assigned to a checksum.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_checksum (
    krb5_context context,
    krb5_checksum *cksum)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `cksum` Specifies the `krb5_checksum` to be released.

**Usage**
The `krb5_free_checksum()` routine releases the storage assigned to a `krb5_checksum` structure and then releases the `krb5_checksum` structure itself.
**krb5_free_context**

**Purpose**
Releases a Kerberos context.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_context (
    krb5_context context
);
```

**Parameters**

**Input**
- `context` Specifies the context to be released.

**Usage**
The `krb5_free_context()` routine is used to release a context which was created by the `krb5_init_context()` routine.

**Return Values**
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_free_cred_contents

Purpose
Releases the storage assigned to contents of a credential.

Format
#include <krb5/krb5.h>

void krb5_free_cred_contents (
   krb5_context context,
   krb5_creds *creds)

Parameters

Input
context Specifies the Kerberos context.
creds Specifies the credentials.

Usage
The krb5_free_cred_contents() routine releases the storage assigned to the contents of a krb5_creds structure. Unlike the krb5_free_creds() routine, the krb5_free_cred_contents() routine does not release the krb5_creds structure.
krb5_free_creds

Purpose
Releases the storage assigned to a credential.

Format
#include <krb5/krb5.h>

void krb5_free_creds (
    krb5_context context,
    krb5_creds *creds)

Parameters

Input
context Specifies the Kerberos context.
creds Specifies the credentials.

Usage
The krb5_free_creds() routine releases the storage assigned to the contents of a krb5_creds structure and then releases the krb5_creds structure itself.
**Purpose**
Releases the storage assigned to a Kerberos data object.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_data (  
    krb5_context context,  
    krb5_data *data)  
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `data` Specifies the data object.

**Usage**
The `krb5_free_data()` routine releases the storage assigned to a Kerberos data object represented by a `krb5_data` structure.
Purpose
Releases the storage assigned to an encrypted ticket part.

Format
#include <krb5/krb5.h>

void krb5_free_enc_tkt_part (  
    krb5_context context,  
    krb5_enc_tkt_part *enc_tkt)

Parameters
Input

context  
    Specifies the Kerberos context.

enc_tkt  
    Specifies the krb5_enc_tkt_part structure to be released.

Usage
The krb5_free_enc_tkt_part() routine releases the storage assigned to the krb5_enc_tkt_part structure and then releases the krb5_enc_tkt_part structure itself. The krb5_enc_tkt_part structure is created when a ticket is decrypted and decoded.
krb5_free_error

Purpose
Releases the storage assigned to a Kerberos error message.

Format
```
#include <krb5/krb5.h>

void krb5_free_error (
    krb5_context context,
    krb5_error *error)
```

Parameters

Input
- `context` Specifies the Kerberos context.
- `error` Specifies the `krb5_error` structure to be released.

Usage
The `krb5_free_error()` routine releases the storage assigned to the `krb5_error` structure and then releases the `krb5_error` structure itself. The `krb5_error` structure is created when a Kerberos error message is processed by the `krb5_rd_error()` routine.
krb5_free_host_realm

Purpose
Releases the storage assigned to a realm list.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_free_host_realm (  
    krb5_context context,  
    char *const *realm_list)

Parameters

Input
context Specifies the Kerberos context.
realm_list Specifies the realm list to be released.

Usage
The krb5_free_host_realm() routine releases the storage assigned to a realm list.

Return Values
The function return value will always be zero.
**krb5_free_kdc_rep**

**Purpose**
Releases the storage assigned to a KDC reply.

**Format**
```
#include <krb5/krb5.h>

void krb5_free_kdc_rep (  
    krb5_context context,  
    krb5_kdc_rep *reply)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `reply` Specifies the KDC reply to be released.

**Usage**
The `krb5_free_kdc_rep()` routine will release the contents of the `krb5_kdc_rep` structure and then it will release the `krb5_kdc_rep` structure itself.
**Purpose**
Releases the storage assigned to a keyblock.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_keyblock (
    krb5_context context,
    krb5_keyblock *keyblock)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `keyblock` Specifies the keyblock to be released.

**Usage**
The `krb5_free_keyblock()` routine will release the contents of the `krb5_keyblock` structure and then it will release the `krb5_keyblock` structure itself.
**Purpose**
Releases the storage assigned to the contents of a keyblock.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_keyblock_contents(
    krb5_context context,
    krb5_keyblock *keyblock)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `keyblock`: Specifies the keyblock to be released.

**Usage**
The `krb5_free_keyblock_contents()` routine will release the contents of the `krb5_keyblock` structure. Unlike the `krb5_free_keyblock()` routine, the `krb5_free_keyblock_contents()` routine will not release the `krb5_keyblock` structure.
**Purpose**
Releases the storage assigned to a host list.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_free_krbhst(
    krb5_context context,
    char *const *host_list)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `host_list` Specifies the host list to be released.

**Usage**
The `krb5_free_krbhst()` routine releases the storage assigned to a host list.

**Return Values**
The function return value will always be zero.
**krb5_free_principal**

**Purpose**
Releases the storage assigned to a principal.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_principal (  
    krb5_context context,  
    krb5_principal principal)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `principal` Specifies the `krb5_principal` to be released.

**Usage**
The `krb5_free_principal()` routine releases storage assigned to a `krb5_principal`. 
**Purpose**
Releases the storage assigned to an array of credentials.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_tgt_creds (
    krb5_context context,
    krb5_creds **creds)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `creds` Specifies the credentials array to be released. The last entry in the array must be a NULL pointer.

**Usage**
The `krb5_free_tgt_creds()` routine releases the storage assigned to an array of `krb5_creds` structures. Each `krb5_creds` structure is released and then the pointer array itself is released.
**krb5_free_ticket**

**Purpose**
Releases the storage assigned to a ticket.

**Format**
```
#include <krb5/krb5.h>

void krb5_free_ticket (  
    krb5_context context,  
    krb5_ticket *ticket)  
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ticket` Specifies the `krb5_ticket` to be released.

**Usage**
The `krb5_free_ticket()` routine releases the storage assigned to a `krb5_ticket` structure and then releases the `krb5_ticket` structure itself.
### krb5_free_tickets

**Purpose**
Releases the storage assigned to an array of tickets.

**Format**
```c
#include <krb5/krb5.h>

void krb5_free_tickets (  
    krb5_context context,  
    krb5_ticket **tickets)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `tickets` Specifies the array to be released. The last entry in the array must be a NULL pointer.

**Usage**
The `krb5_free_tickets()` routine releases the storage assigned to an array of `krb5_ticket` structures. Each `krb5_ticket` structure is released and then the pointer array itself is released.
**krb5_gen_replay_name**

**Purpose**
Generates a replay cache name.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_gen_replay_name (
    krb5_context context,
    const krb5_address *inaddr,
    const char *unique,
    char **string)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `inaddr`: Specifies the address to be incorporated into the cache name.
- `unique`: Specifies the unique portion of the replay cache name.

**Output**
- `string`: Returns the generated replay cache name. This string should be freed by the application when it is no longer needed.

**Usage**
The `krb5_gen_replay_name()` routine will generate a unique replay cache name based upon the Kerberos address supplied by the caller. The unique parameter is used to differentiate this replay cache from others currently in use on the system. The generated cache name will consist of the unique portion concatenated with the hexadecimal representation of the Kerberos address.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_generate_seq_number

Purpose
Generates a random sequence number

Format
#include <krb5/krb5.h>

krb5_error_code krb5_generate_seq_number (  
    krb5_context context,  
    const krb5_keyblock *key,  
    krb5_int32 *seqno)

Parameters

Input
context Specifications the Kerberos context.
key Specifies the key which will be used to generate the random sequence number.

Output
seqno Returns the random sequence number.

Usage
The krb5_generate_seq_number() generates a random sequence number based upon the supplied key.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Generates a subsession key.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_generate_subkey (  
    krb5_context context,  
    const krb5_keyblock *key,  
    krb5_keyblock **subkey)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `key` Specifies the session key.

**Output**
- `subkey` Returns the generated subsession key. The `krb5_free_keyblock()` routine should be called to release the key when it is no longer needed.

**Usage**
The `krb5_generate_subkey()` generates a random subsession key which is based upon the supplied session key.

**Return Values**
The function return value is zero if no errors occurred. Otherwise, it is a Kerberos error code.
krb5_get_cred_from_kdc

Purpose
Obtains a service ticket from the Kerberos KDC server.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_get_cred_from_kdc (  
    krb5_context context,  
    krb5_ccache ccache,  
    krb5_creds *in_cred,  
    krb5_creds **out_cred,  
    krb5_creds ***tgts)

Parameters

Input
context
Specifies the Kerberos context.

ccache
Specifies the credentials cache. The initial TGT for the local realm must already be in the cache. The Kerberos runtime will obtain additional ticket-granting tickets as needed if the target server is not in the local realm.

in_cred
Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The second_ticket field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

Output
out_cred
Returns the service ticket. The krb5_free_creds() routine should be called to release the credentials when they are no longer needed.

tgts
Returns any new ticket-granting tickets which were obtained while getting the service target from the KDC in the target realm. There may be ticket-granting tickets returned for this parameter even if the Kerberos runtime was ultimately unable to obtain a service ticket from the target KDC. The krb5_free_tgt_creds() routine should be called to release the TGT array when it is no longer needed.

Usage
The krb5_get_cred_from_kdc() routine obtains a service ticket from the Kerberos KDC server. The credentials are not stored in the credentials cache (the application should store them in the cache if appropriate). The application should not call krb5_get_cred_from_kdc() if the requested service ticket is already in the credentials cache.

The krb5_get_cred_from_kdc() routine will obtain any necessary ticket-granting tickets for intermediate realms between the client realm and the server realm. It will then call the krb5_get_cred_via_tkt() routine to obtain the actual service ticket. The KDC options will be the same as the TGT ticket options. The KDC_OPT_ENC_TKT_IN_SKEY flag will be set if the in_cred parameter provided a second ticket.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
** krb5_get_cred_from_kdc_renew **

**Purpose**
Renews a service ticket obtained from the Kerberos KDC server.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_get_cred_from_kdc_renew (
    krb5_context context,
    krb5_ccache ccache,
    krb5_creds *in_cred,
    krb5_creds ***tgts)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache. The initial TGT for the local realm must already be in the cache. The Kerberos runtime will obtain additional ticket-granting tickets as needed if the target server is not in the local realm.
- `in_cred` Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The `second_ticket` field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

**Output**
- `out_cred` Returns the renewed service ticket. The `krb5_free_creds()` routine should be called to release the credentials when they are no longer needed.
- `tgts` Returns any new ticket-granting tickets which were obtained while getting the service target from the KDC in the target realm. There may be ticket-granting tickets returned for this parameter even if the Kerberos runtime was ultimately unable to obtain a service ticket from the target KDC.

The `krb5_free_tgt_creds()` routine should be called to release the TGT array when it is no longer needed.

**Usage**
The `krb5_get_cred_from_kdc_renew()` routine renews a service ticket obtained from the Kerberos KDC server. The credentials are not stored in the credentials cache (the application should store them in the cache if appropriate). The application should call `krb5_get_cred_from_kdc_renew()` to renew a renewable ticket before the ticket end time is reached. Note that a renewable ticket may not be renewed after its end time even if its `renew_till` time has not been reached yet.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Validates a service ticket obtained from the Kerberos KDC server.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_cred_from_kdc_validate (
    krb5_context context,
    krb5_ccache ccache,
    krb5_creds *in_cred,
    krb5_creds **out_cred,
    krb5_creds ***tgts)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `ccache` Specifies the credentials cache. The initial TGT for the local realm must already be in the cache. The Kerberos runtime will obtain additional ticket-granting tickets as needed if the target server is not in the local realm.
- `in_cred` Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The `second_ticket` field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

**Output**

- `out_cred` Returns the validated service ticket. The `krb5_free_creds()` routine should be called to release the credentials when they are no longer needed.
- `tgts` Returns any new ticket-granting tickets which were obtained while getting the service target from the KDC in the target realm. There may be ticket-granting tickets returned for this parameter even if the Kerberos runtime was ultimately unable to obtain a service ticket from the target KDC. The `krb5_free_tgt_creds()` routine should be called to release the TGT array when it is no longer needed.

**Usage**

The `krb5_get_cred_from_kdc_validate()` routine validates a service ticket obtained from the Kerberos KDC server. The credentials are not stored in the credentials cache (the application should store them in the cache if appropriate). The application should call `krb5_get_cred_from_kdc_validate()` to validate a postdated ticket once the ticket start time has been reached.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_cred_via_tkt**

**Purpose**
Obtains a service ticket from the Kerberos KDC server.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_cred_via_tkt (  
    krb5_context context,  
    krb5_creds *tkt,  
    const krb5_flags kdc_options,  
    krb5_address **address,  
    krb5_creds *in_cred,  
    krb5_creds **out_cred)
```

**Parameters**

**Input**
- **context**
  Specifies the Kerberos context.
- **tkt**
  Specifies the ticket-granting ticket for the realm containing the target server for the service ticket. The client in the TGT must be the same as the client in the request credentials.
- **kdc_options**
  Specifies KDC options for the service ticket as follows:
  - **KDC_OPT_FORWARDABLE** - Obtain a forwardable ticket.
  - **KDC_OPT_PROXIAIBLE** - Obtain a proxiable ticket.
  - **KDC_OPT_ALLOW_POSTDATE** - Allow postdated tickets.
  - **KDC_OPT_RENEWABLE** - Obtain a renewable ticket. The *renew-till* time must be set in the request.
  - **KDC_OPT_RENEWABLE_OK** - A renewable ticket is acceptable if the KDC policy does not allow a ticket to be generated with the requested endtime.
  - **KDC_OPT_ENC_TKT_IN_SKEY** - Encrypt the service ticket in the session key of the second ticket.
- **address**
  Specifies the addresses to be placed in the ticket. The ticket addresses determine which host systems can generate requests which use the ticket.
- **in_cred**
  Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The *second_ticket* field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

**Output**
- **out_cred**
  Returns the service ticket. The *krb5_free_creds()* routine should be called to release the credentials when they are no longer needed.

**Usage**
The **krb5_get_cred_via_tkt()** routine uses the supplied ticket-granting ticket to obtain a service ticket to the requested server for the requested client.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Obtains a service ticket.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_get_credentials (  
    krb5_context context,  
    const krb5_flags options,  
    krb5_ccache ccache,  
    krb5_creds *in_cred,  
    krb5_creds **out_cred)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `options` Specifies the option flags as follows:
  - `KRB5_GC_USER_USER` - Obtain a user-to-user ticket.
  - `KRB5_GC_CACHED` - Do not obtain a service ticket if one is not found in the credentials cache.
- `ccache` Specifies the credentials cache to be used. The initial TGT must already be in the cache.
- `in_cred` Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The `second_ticket` field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

**Output**
- `out_cred` Returns the service ticket. The `krb5_free_creds()` routine should be called to release the credentials when they are no longer needed.

**Usage**
The `krb5_get_credentials()` routine will obtain a service ticket for the requested server. This routine is the normal way for an application to obtain a service ticket. If the service ticket is already in the credentials cache, the `krb5_get_credentials()` routine will return the cached ticket. Otherwise, the `krb5_get_credentials()` routine will call the `krb5_get_cred_from_kdc()` routine to obtain a service ticket from the KDC.

The `krb5_get_credentials()` routine will store any tickets obtained during its processing in the credentials cache. This includes the requested service ticket as well as any ticket-granting tickets required to obtain the service ticket.

If `KRB5_GC_CACHED` is specified, the `krb5_get_credentials()` routine will only search the credentials cache for a service ticket.

If `KRB5_GC_USER_USER` is specified, the `krb5_get_credentials()` routine will get credentials for user-to-user authentication. In user-to-user authentication, the secret key for the server is the session key from the server's ticket-granting ticket (TGT). The TGT is passed from the server to the client over the network (this is safe since the TGT is encrypted in a key known only by the Kerberos server). The client must then pass this TGT to `krb5_get_credentials()` as the second ticket in the request credentials. The Kerberos server will use this TGT to construct a user-to-user ticket which can be verified by the server using the session key from its TGT.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_get_credentials_renew

Purpose
Renews a ticket.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_get_credentials_renew (  
    krb5_context context,  
    const krb5_flags options,  
    krb5_ccache ccache,  
    krb5_creds *in_cred,  
    krb5_creds **out_cred)

Parameters

Input
context Specifies the Kerberos context.
options Specifies the option flags as follows:
  • KRB5_GC_USER_USER - Obtain a user-to-user ticket.
ccache Specifies the credentials cache to be used.
in_cred Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The second_ticket field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

Output
out_cred Returns the service ticket. The krb5_free_creds() routine should be called to release the credentials when they are no longer needed.

Usage
The krb5_get_credentials_renew() routine will renew a service ticket for the requested service. Upon successful completion, the credentials cache will be re-initialized and the service ticket will be stored in the cache.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_get_credentials_validate

Purpose
Validates a ticket.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_get_credentials_validate (
    krb5_context context,
    const krb5_flags options,
    krb5_ccache ccache,
    krb5_creds *in_cred,
    krb5_creds **out_cred)

Parameters
Input
context  Specifies the Kerberos context.
options  Specifies the option flags as follows:
    • KRB5_GC_USER_USER - Obtain a user-to-user ticket.
cache  Specifies the credentials cache to be used.
in_cred  Specifies the request credentials. The client and server fields must be set to the desired values for the service ticket. The second_ticket field must be set if the service ticket is to be encrypted in a session key. The ticket expiration time can be set to override the default expiration time.

Output
out_cred  Returns the service ticket. The krb5_free_creds() routine should be called to release the credentials when they are no longer needed.

Usage
The krb5_get_credentials_validate() routine will validate a service ticket for the requested service. Upon successful completion, the credentials cache will be re-initialized and the service ticket will be stored in the cache.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_default_in_tkt_ktypes**

**Purpose**
Returns the default encryption types that are used when requesting an initial ticket from the KDC.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_default_in_tkt_ktypes (
    krb5_context context,
    krb5_enctype **ktypes)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.

**Output**
- `ktypes` Returns an array of encryption types. The last entry in the array will be `ENCTYPE_NULL`. The caller is responsible for freeing the array returned for this parameter when it is no longer needed.

**Usage**
The `krb5_get_default_in_tkt_ktypes()` routine returns the default encryption types which will be used when requesting the initial ticket from the KDC. The values are set by the `krb5_set_default_in_tkt_ktypes()` routine or obtained from the Kerberos configuration file.

**Return Values**
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
Purpose
Returns the default realm for the local system.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_get_default_realm (
    krb5_context context,
    char ***realm)

Parameters
Input
context Specifies the Kerberos context.

Output
realm Returns the realm name. The application should free the name when it is no longer needed.

Usage
The krb5_get_default_realm() routine returns the default realm for the local system. The default realm is set by the krb5_set_default_realm() routine. If the default realm has not been set, it is obtained from the default_realm entry in the [libdefaults] section of the Kerberos configuration file. If this file does not exist, the default realm is obtained from the /krb5/krb.conf file.

Return Values
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Returns the default encryption types which will be used when requesting a service ticket from the KDC.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_default_tgs_ktypes (
    krb5_context context,
    krb5_enctype **ktypes)
```

**Parameters**
**Input**
`context` Specifies the Kerberos context.

**Output**
`ktypes` Returns an array of encryption types. The last entry in the array will be `ENCTYPE_NULL`. The caller is responsible for freeing the array returned for this parameter when it is no longer needed.

**Usage**
The `krb5_get_default_tgs_ktypes()` routine returns the default encryption types which will be used when requesting a service ticket from the KDC. The values are set by the `krb5_set_default_tgs_ktypes()` routine or obtained from the Kerberos configuration file.

**Return Values**
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_get_host_realm

Purpose
Gets the Kerberos realm name for a host name.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_get_host_realm ( 
    krb5_context context, 
    const char *host, 
    char ***realm_list)

Parameters
Input
context Specifies the Kerberos context.
host Specifies the host name. The local host name is used if NULL is specified for this parameter.

Output
realm_list Returns an array of realm names. The last entry in the array will be a NULL pointer. The krb5_free_host_realm() routine should be called to release the realm list when it is no longer needed.

Usage
The krb5_get_host_realm() routine returns a list of Kerberos realm names for the specified host name. The entries in the [domain_realm] section of the Kerberos configuration file are used. A direct match will take precedence over a suffix match. The current implementation of this routine will return a single realm name. If no realm name is found, the uppercased host domain will be returned as the realm name.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_in_tkt_with_keytab**

**Purpose**
Gets an initial ticket using a key table.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_in_tkt_with_keytab (  
    krb5_context context,  
    const krb5_flags *options,  
    const krb5_address *addrs,  
    krb5_enctype *enctypes,  
    krb5_preauthtype *pre_auth_types,  
    krb5_keytab *keytab,  
    krb5_ccache *ccache,  
    krb5_creds *creds,  
    krb5_kdc_rep **ret_as_reply)
```

**Parameters**

**Input**

- **context**
  Specifies the Kerberos context.

- **options**
  Specifies KDC options as follows:
  - **KDC_OPT_FORWARDABLE** - Obtain a forwardable ticket.
  - **KDC_OPT_PROXIABLE** - Obtain a proxiable ticket.
  - **KDC_OPT_ALLOW_POSTDATE** - Allow postdated tickets.
  - **KDC_OPT_RENEWABLE** - Obtain a renewable ticket. The `renew-till` time must be set in the request.
  - **KDC_OPT_RENEWABLE_OK** - A renewable ticket is acceptable if the KDC policy does not allow a ticket to be generated with the requested endtime.

- **addrs**
  Specifies the addresses to be placed in the ticket. If NULL is specified for this parameter, the local system addresses will be used. The ticket addresses determine which host systems can generate requests which use the ticket.

- **enctypes**
  Specifies an array of encryption types to be used. The last entry in the array must be **ENCTYPE_NULL**. If NULL is specified for this parameter, the default encryption types will be used. The following encryption types may be specified:
  - **ENCTYPE_DES_CBC_CRC** - 32-bit CRC checksum with DES encryption. This encryption type should be used for interoperability with older levels of Kerberos V5.
  - **ENCTYPE_DES_CBC_MD5** - MD5 checksum with DES encryption.

- **pre_auth_types**
  Specifies an array of preauthentication types to be used. The last entry in the array must be **KR55_PADATA_NONE**. If NULL is specified for this parameter, no preauthentication will be done unless required by KDC policy. If multiple preauthentication types are specified, the KDC is supposed to accept the request as long as it recognizes at least one of the preauthentication types. Unfortunately, early implementations of the KDC did not follow this rule and will fail the request if the first preauthentication type is not recognized. The following preauthentication types may be specified:
  - **KR55_PADATA_ENC_TIMESTAMP** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with a Kerberos KDC.
  - **KR55_PADATA_ENC_UNIX_TIME** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with an older DCE KDC.
**krb5_get_in_tkt_with_keytab**

**keytab**
Specifies the key table containing the key for the client principal. The entry with the highest key version number will be used. The default key table will be used if `NULL` is specified for this parameter.

**ccache**
Specifies the credentials cache handle. The initial ticket will be stored in the credentials cache for later use by the application. The credentials will not be stored if `NULL` is specified for this parameter.

**Input/Output**

**creds**
Specifies the credentials which will be used to obtain the initial ticket. The client and server fields must be set. The `endtime` field may be set to explicitly specify the ticket lifetime or it may be set to zero to use the default ticket lifetime. The `renew_till` field must be set if a renewable ticket is being requested. The `starttime` field must be set if a postdated ticket is being requested.

Upon completion of the request, the `creds` will be updated with the initial ticket and the session key. The `krb5_free_cred_contents()` or `krb5_free_creds()` routine should be called to release the credentials when they are no longer needed.

**Output**

**ret_as_reply**
Returns the KDC reply. Specify `NULL` for this parameter if the KDC reply is not needed. The `krb5_free_kdc_rep()` routine should be called to release the reply when it is no longer needed.

**Usage**
The `krb5_get_in_tkt_with_keytab()` routine is called to obtain an initial ticket using a key table. This initial ticket can then be used to obtain service tickets.

The client must be in the same realm as the KDC in order to be able to obtain an initial ticket from the KDC. The initial ticket can be used to obtain tickets in the same realm or in different realms as long as the proper inter-realm trust relationships have been established.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_in_tkt_with_password**

**Purpose**
Gets an initial ticket using a text password.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_get_in_tkt_with_password (   
    krb5_context context,   
    const krb5_flags options,   
    krb5_address *addr,   
    krb5_enctype *enctypes,   
    krb5_preauthtype *pre_auth_types,   
    const char *password,   
    krb5_ccache ccache,   
    krb5_creds *creds,   
    krb5_kdc_rep **ret_as_reply)
```

**Parameters**

**Input**

*context* Specifies the Kerberos context.

*options* Specifies KDC options as follows:

- **KDC_OPT_FORWARDABLE** - Obtain a forwardable ticket.
- **KDC_OPT_PROXIABLE** - Obtain a proxiable ticket.
- **KDC_OPT_ALLOW_POSTDATE** - Allow postdated tickets.
- **KDC_OPT_RENEWABLE** - Obtain a renewable ticket. The *renew-till* time must be set in the request.
- **KDC_OPT_RENEWABLE_OK** - A renewable ticket is acceptable if the KDC policy does not allow a ticket to be generated with the requested endtime.

*addr* Specifies the addresses to be placed in the ticket. If NULL is specified for this parameter, the local system addresses will be used. The ticket addresses determine which host systems can generate requests which use the ticket.

*enctypes* Specifies an array of encryption types to be used. The last entry in the array must be **ENCTYPE_NULL**. If NULL is specified for this parameter, the default encryption types will be used. The following encryption types may be specified:

- **ENCTYPE_DES_CBC_CRC** - 32-bit CRC checksum with DES encryption. This encryption type should be used for interoperability with older levels of Kerberos V5.

- **ENCTYPE_DES_CBC_MD5** - MD5 checksum with DES encryption.

*pre_auth_types* Specifies an array of preauthentication types to be used. The last entry in the array must be **KRBS5_PADATA_NONE**. If NULL is specified for this parameter, no preauthentication will be done unless required by KDC policy. If multiple preauthentication types are specified, the KDC is supposed to accept the request as long as it recognizes at least one of the preauthentication types. Unfortunately, early implementations of the KDC did not follow this rule and will fail the request if the first preauthentication type is not recognized. The following preauthentication types may be specified:

- **KRBS5_PADATA_ENC_TIMESTAMP** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with a Kerberos KDC.

- **KRBS5_PADATA_ENC_UNIX_TIME** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with an older DCE KDC.
**krb5_get_in_tkt_with_password**

**password**
Specifies the password string. This string will be converted to a Kerberos key value using the rules for the first encryption type specified by the enctypes parameter.

**ccache**
Specifies the credentials cache handle. The initial ticket will be stored in the credentials cache for later use by the application. The credentials will not be stored if NULL is specified for this parameter.

**Input/Output**

**creds**
Specifies the credentials which will be used to obtain the initial ticket. The client and server fields must be set. The *endtime* field may be set to explicitly specify the ticket lifetime or it may be set to zero to use the default ticket lifetime.

The *renew_till* field must be set if a renewable ticket is being requested.

The *starttime* field must be set if a postdated ticket is being requested.

Upon completion of the request, the *creds* will be updated with the initial ticket and the session key. The *krb5_free_cred_contents()* or *krb5_free_creds()* routine should be called to release the credentials when they are no longer needed.

**Output**

**ret_as_reply**
Returns the KDC reply. Specify NULL for this parameter if the KDC reply is not needed. The *krb5_free_kdc_rep()* routine should be called to release the reply when it is no longer needed.

**Usage**
The *krb5_get_in_tkt_with_password()* routine is called to obtain an initial ticket using a text password. This initial ticket can then be used to obtain service tickets. The client must be in the same realm as the KDC in order to be able to obtain an initial ticket from the KDC. The initial ticket can be used to obtain tickets in the same realm or in different realms as long as the proper inter-realm trust relationships have been established.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_in_tkt_with_skey**

**Purpose**
Gets an initial ticket using a session key.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_get_in_tkt_with_skey (
  krb5_context context,
  const krb5_flags options,
  krb5_address *const *addr,
  krb5_enctype *enctypes,
  krb5_preauthtype *pre_auth_types,
  const krb5_keyblock *key,
  krb5_ccache ccache,
  krb5_creds *creds,
  krb5_kdc_rep **ret_as_reply)
```

**Parameters**

**Input**
- **context** Specifies the Kerberos context.
- **options** Specifies KDC options as follows:
  - **KDC_OPT_FORWARDABLE** - Obtain a forwardable ticket.
  - **KDC_OPT_PROXIABLE** - Obtain a proxiable ticket.
  - **KDC_OPT_ALLOW_POSTDATE** - Allow postdated tickets.
  - **KDC_OPT_RENEWABLE** - Obtain a renewable ticket. The renew-till time must be set in the request.
  - **KDC_OPT_RENEWABLE_OK** - A renewable ticket is acceptable if the KDC policy does not allow a ticket to be generated with the requested endtime.
- **addr** Specifies the addresses to be placed in the ticket. If **NULL** is specified for this parameter, the local system addresses will be used. The ticket addresses determine which host systems can generate requests which use the ticket.
- **enctypes** Specifies an array of encryption types to be used. The last entry in the array must be **ENCTYPE_NULL**. If **NULL** is specified for this parameter, the default encryption types will be used. The following encryption types may be specified:
  - **ENCTYPE_DES_CBC_CRC** - 32-bit CRC checksum with DES encryption. This encryption type should be used for interoperability with older levels of Kerberos V5.
  - **ENCTYPE_DES_CBC_MD5** - MD5 checksum with DES encryption.
- **pre_auth_types** Specifies an array of preauthentication types to be used. The last entry in the array must be **KRBS5_PADATA_NONE**. If **NULL** is specified for this parameter, no preauthentication will be done unless required by KDC policy. If multiple preauthentication types are specified, the KDC is supposed to accept the request as long as it recognizes at least one of the preauthentication types. Unfortunately, early implementations of the KDC did not follow this rule and will fail the request if the first preauthentication type is not recognized. The following preauthentication types may be specified:
  - **KRBS5_PADATA_ENC_TIMESTAMP** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with a Kerberos KDC.
  - **KRBS5_PADATA_ENC_UNIX_TIME** - Encrypted timestamp preauthentication. This preauthentication type should be used for interoperability with an older DCE KDC.
krb5_get_in_tkt_with_skey

**key**
Specifies the key to be used. The default key table will be used if **NULL** is specified for this parameter.

**ccache**
Specifies the credentials cache handle. The initial ticket will be stored in the credentials cache for later use by the application. The credentials will not be stored if **NULL** is specified for this parameter.

**Input/Output**

**creds**
Specifies the credentials which will be used to obtain the initial ticket. The client and server fields must be set. The *endtime* field may be set to explicitly specify the ticket lifetime or it may be set to zero to use the default ticket lifetime. The *renew_till* field must be set if a renewable ticket is being requested. The *starttime* field must be set if a postdated ticket is being requested. Upon completion of the request, the *creds* will be updated with the initial ticket and the session key. The krb5_free_credential_contents() or krb5_free_creds() routine should be called to release the credentials when they are no longer needed.

**Output**

**ret_as_reply**
Returns the KDC reply. Specify **NULL** for this parameter if the KDC reply is not needed. The krb5_free_kdc_rep() routine should be called to release the reply when it is no longer needed.

**Usage**
The krb5_get_in_tkt_with_skey() routine is called to obtain an initial ticket using a session key. This initial ticket can then be used to obtain service tickets. The client must be in the same realm as the KDC in order to be able to obtain an initial ticket from the KDC. The initial ticket can be used to obtain tickets in the same realm or in different realms as long as the proper inter-realm trust relationships have been established.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_get_krbhst**

**Purpose**
Returns a list of KDC hosts for a Kerberos realm.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_get_krbhst (
    krb5_context context,
    const krb5_data *realm,
    char ***hostlist)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `realm` Specifies the Kerberos realm.

**Output**
- `hostlist` Returns the KDC host list. The last entry in the list will be a NULL pointer. The `krb5_free_krbhst()` routine should be called to release the host list when it is no longer needed.

**Usage**
The `krb5_get_krbhst()` routine returns a list of hosts in the specified realm which are running Kerberos servers. The list is obtained from the [realms] section of the Kerberos configuration file or from the `/krb5/krb.conf` file if an entry is not found in the Kerberos configuration file.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_get_server_rcache

Purpose
Generates a replay cache for server use.

Format

```c
#include <krb5/krb5.h>

kerb5_error_code krb5_get_server_rcache (
    krb5_context context,
    const krb5_data *piece,
    krb5_rcache *ret_rcache)
```

Parameters

**Input**
- `context` Specifies the Kerberos context.
- `piece` Specifies the unique portion of the replay cache name.

**Output**
- `ret_rcache` Returns the replay cache handle. The `krb5_rc_close()` routine should be called to close the replay cache when it is no longer needed.

Usage
The `krb5_get_server_rcache()` routine will generate a unique replay cache name and then open the replay cache. The piece parameter is used to differentiate this replay cache from others currently in use on the system by the same user. The generated cache name will be in the form `rc_piece_uid` and will use the default replay cache type. The replay cache will be initialized if it can not be recovered. The clock skew value will be obtained from the Kerberos context if it is necessary to initialize the cache.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_init_context**

**Purpose**
Creates a Kerberos context.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_init_context (  
    krb5_context *context)
```

**Parameters**

**Output**

- `context`: Returns the handle for the Kerberos context.

**Usage**

The `krb5_init_context()` routine creates a new Kerberos context and initializes it with default values obtained from the Kerberos configuration file. Each application needs at least one Kerberos context. A context may be shared by multiple threads within the same process. Use the `krb5_free_context()` routine to release the context when it is no longer needed.

**Return Values**

The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Adds a new entry to a key table.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_add_entry (  
    krb5_context context,  
    krb5_keytab ktid,  
    krb5_keytab_entry *entry)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `ktid`: Specifies the key table handle.
- `entry`: Specifies the entry to be added to the key table.

**Usage**
The `krb5_kt_add_entry()` routine adds a new entry to a key table. No checking is done for duplicate entries. The key table type must support write operations.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
### krb5_kt_close

#### Purpose
Closes a key table.

#### Format
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_close (
    krb5_context context,
    krb5_keytab ktid)
```

#### Parameters
**Input**
- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.

#### Usage
The `krb5_kt_close()` routine closes a key table. The key table handle may not be used once this routine completes.

#### Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_kt_default

Purpose
Resolves the default key table.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_kt_default (  
    krb5_context context,  
    krb5_keytab *ktid)

Parameters

Input
context Specifies the Kerberos context.

Output
ktid Returns the key table handle.

Usage
The krb5_kt_default() routine resolves the default key table and returns a handle which can be used to access the table. This is equivalent to calling the krb5_kt_resolve() routine with the name returned by the krb5_kt_default_name() routine.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Returns the default key table name.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_kt_default_name(
    krb5_context context,
    char *name,
    int name_size)
```

**Parameters**

**Input**

- `context`: Specifies the Kerberos context.
- `name_size`: Specifies the size of the buffer pointed to by the name parameter. The size must be large enough to contain the key table name and the trailing delimiter. One way to do this is to allocate the buffer to be `MAX_KEYTAB_NAME_LENGTH+1` bytes.

**Output**

- `name`: Returns the key table name.

**Usage**

The `krb5_kt_default_name()` routine returns the name of the default key table for the current user. If the `KRB5_KTNAME` environment variable is set, this will be the name of the default key table. Otherwise, the key table name will be obtained from the `default_keytab_name` entry in the `[libdefaults]` section of the Kerberos configuration file. If this entry is not defined, the default key table name will be `/krb5/v5srvtab`.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Ends the sequential reading of the key table.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_kt_end_seq_get (
    krb5_context context, 
    krb5_keytab ktid, 
    krb5_kt_cursor *cursor)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.

**Input/Output**

- `cursor` Specifies the cursor created by the `krb5_kt_start_seq_get()` routine.

**Usage**

The `krb5_kt_end_seq_get()` routine unlocks the key table and releases the cursor. The cursor may not be used once `krb5_kt_end_seq_get()` has completed.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_kt_free_entry**

**Purpose**
Releases the storage assigned to a key table entry.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_free_entry (  
    krb5_context context,  
    krb5_keytab_entry *entry)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `entry` Specifies the key table entry.

**Usage**
The `krb5_kt_free_entry()` routine will release the contents of a key table entry. It will not free the `krb5_keytab_entry` structure itself.

**Return Values**
The function return value will always be zero.
**krb5_kt_get_entry**

**Purpose**
Returns an entry from the key table.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_get_entry (
    krb5_context context,
    krb5_keytab ktid,
    krb5_principal principal,
    krb5_kvno vno,
    krb5_enctype enctype,
    krb5_keytab_entry *entry)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.
- `principal` Specifies the principal.
- `vno` Specifies the key version number for the key to be retrieved. Specify a version number of zero to retrieve the key with the highest version number.
- `enctype` Specifies the key encryption type. Specify an encryption type of zero if the encryption type does not matter.

**Output**
- `entry` Returns the contents of the key table entry. The `krb5_kt_free_entry()` routine should be called to release the entry contents when they are no longer needed.

**Usage**
The `krb5_kt_get_entry()` routine returns an entry from the key table for the specified principal.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Returns the key table name.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_get_name (  
    krb5_context context,  
    krb5_keytab ktid,  
    char *name,  
    int name_size)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.
- `name_size` Specifies the size of the buffer pointed to by the name parameter. The size must be large enough to contain the key table name and the trailing delimiter. One way to do this is to allocate the buffer to be `MAX_KEYTAB_NAME_LENGTH+1` bytes.

**Output**
- `name` Returns the key table name.

**Usage**
The `krb5_kt_get_name()` routine returns the name of the key table. The returned name will not include the key table type prefix.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_kt_get_type**

**Purpose**
Returns the key table type.

**Format**
```
#include <krb5/krb5.h>

char *krb5_kt_get_type (  
    krb5_context context,  
    krb5_keytab ktid)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.

**Usage**
The `krb5_kt_get_type()` routine returns the key table type.

**Return Values**
The function return value will be the address of the key table type. This is a read-only value and must not be freed by the application.
**krb5_kt_next_entry**

**Purpose**
Returns the next entry from the key table.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_kt_next_entry (  
    krb5_context context,  
    krb5_keytab ktid,  
    krb5_keytab_entry *entry,  
    krb5_kt_cursor *cursor)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ktid` Specifies the key table handle.

**Input/Output**
- `cursor` Specifies the cursor created by the `krb5_kt_start_seq_get()` routine. The cursor will be updated upon successful completion of this routine.

**Output**
- `entry` Returns the contents of the table entry. The `krb5_kt_free_entry()` routine should be called to release the entry contents when they are no longer needed.

**Usage**
The `krb5_kt_next_entry()` reads the next entry from the key table and returns it to the application. The `krb5_kt_start_seq_get()` routine must be called to begin the sequential read operation. The `krb5_kt_next_entry()` routine is then called repeatedly to read table entries. Finally, the `krb5_kt_end_seq_get()` routine is called when no more entries are to be read.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Retrieves the service key from the key table.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_kt_read_service_key (  
    krb5_context context,  
    krb5_pointer keytab_name  
    krb5_principal principal,  
    krb5_kvno vno,  
    krb5_enctype enctype,  
    krb5_keyblock **key)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `keytab_name`: Specifies the key table name. If a NULL address is specified, the default key table will be used.
- `principal`: Specifies the service principal.
- `vno`: Specifies the key version number for the key to be retrieved. Specify a version number of zero to retrieve the key with the highest version number.
- `enctype`: Specifies the key encryption type. Specify an encryption type of zero if the encryption type does not matter.

**Output**
- `key`: Returns the retrieved key. The `krb5_free_keyblock()` routine should be called to release the key when it is no longer needed.

**Usage**
The `krb5_kt_read_service_key()` routine retrieves the key for a service principal from a key table.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_kt_register**

**Purpose**
Defines a new key table type.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_kt_register (  
    krb5_context context,  
    krb5_kt_ops *ops)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `ops` Specifies the key table operations vector. This vector defines the routines which will be called to perform the various key table operations for the new type.

**Usage**
The `krb5_kt_register()` routine registers a new key table type. An error will be returned if the key table type has already been registered. Once the new type is registered, it can be used by any thread in the current process. The type is not known outside the current process and will no longer be registered when the application ends.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_kt_remove_entry

Purpose
Removes an entry from a key table.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_kt_remove_entry (  
            krb5_context context,  
            krb5_keytab ktid,  
            krb5_keytab_entry *entry)

Parameters
Input
context     Specifies the Kerberos context.
ktid        Specifies the key table handle.
entry       Specifies the entry to be removed from the key table.

Usage
The krb5_kt_remove_entry() routine removes an entry from a key table. The key table type must support write operations.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_kt_resolve

Purpose
Resolves a key table name.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_kt_resolve (  
    krb5_context context,  
    krb5_const char *keytab_name,  
    krb5_keytab *ktid)

Parameters
Input
context Specifies the Kerberos context.
keytab_name Specifies the key table name in the format type:name. The type must be a registered key table type and the name must uniquely identify a particular key table of the specified type.

Output
ktid Returns the key table handle.

Usage
The krb5_kt_resolve() routine resolves a key table name and returns a handle which can be used to access the table. The Kerberos runtime supports two key table types: FILE and WRFILE. Additional key table types can be registered by the application by calling the krb5_kt_register() routine. If no type is specified, the default is FILE.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
Purpose
Starts sequentially retrieving entries from the key table.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_kt_start_seq_get (  
    krb5_context context,  
    krb5_keytab ktid,  
    krb5_kt_cursor *cursor)

Parameters
Input
context     Specifies the Kerberos context.
ktid        Specifies the key table handle.

Output
cursor      Returns the cursor. The krb5_kt_end_seq_get() routine should be called to release the cursor at the completion of the sequential read operation.

Usage
The krb5_kt_start_seq_get() routine prepares for sequentially reading entries in the key table. The krb5_kt_next_entry() routine is called repeatedly to retrieve each successive table entry. The krb5_kt_end_seq_get() routine is called at the completion of the read operation.

The key table is locked when the krb5_kt_start_seq_get() routine is called and remains locked until the krb5_kt_end_seq_get() routine is called. Write access to the key table by other processes and threads will be blocked until the table is unlocked. After the krb5_kt_start_seq_get() routine has been called, the current thread may not call any other key table functions except krb5_kt_next_entry() and krb5_kt_end_seq_get() for the specified table.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_md4_crypto_compat_ctl**

**Purpose**
Sets the compatibility mode for MD4 checksum generation.

**Format**

```c
#include <krb5/krb5.h>

void krb5_md4_crypto_compat_ctl (krb5_boolean compat_mode);
```

**Parameters**

**Input**

`compat_mode` Specifies the compatibility mode as **TRUE** or **FALSE**.

**Usage**

The `krb5_md4_crypto_compat_ctl()` routine sets the compatibility mode for MD4 DES checksum generation. Early beta levels of Kerberos Version 5 computed the MD4 DES checksum incorrectly. Enabling compatibility mode will cause the Kerberos runtime to generate the MD4 DES checksum in the same way while disabling compatibility mode will cause the Kerberos runtime to generate the checksum correctly.

MD4 compatibility mode is set for the entire process by this routine and overrides the compatibility mode set by the `rsa_md4_des_compat` entry in the Kerberos configuration file.
Purpose
Sets the compatibility mode for MD5 checksum generation.

Format
#include <krb5/krb5.h>

void krb5_md5_crypto_compat_ctl (krb5_boolean compat_mode)

Parameters
Input
compat_mode Specifies the compatibility mode as TRUE or FALSE.

Usage
The krb5_md5_crypto_compat_ctl() routine sets the compatibility mode for MD5 DES checksum generation. Early beta levels of Kerberos Version 5 computed the MD5 DES checksum incorrectly. Enabling compatibility mode will cause the Kerberos runtime to generate the MD5 DES checksum in the same way while disabling compatibility mode will cause the Kerberos runtime to generate the checksum correctly.

MD5 compatibility mode is set for the entire process by this routine and overrides the compatibility mode set by the rsa_md5_des_compat entry in the Kerberos configuration file.
**krb5_mk_error**

**Purpose**
Creates a Kerberos KRB_ERROR message.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_mk_error (  
    krb5_context context,  
    const krb5_error *dec_err,  
    krb5_data *enc_err)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `dec_err`: Specifies the krb5_error structure which is to be encoded.

**Output**
- `enc_err`: Returns the encoded krb5_error structure as a byte stream. The storage pointed to by the data field of the krb5_data structure should be freed by the application when it is no longer needed.

**Usage**
The `krb5_mk_error()` routine creates a Kerberos KRB_ERROR message. This message is then sent to the remote partner instead of sending a reply message. For example, if an error is detected while processing an AP_REQ message, the application would return a KRB_ERROR message instead of an AP_REP message.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
Purpose
Creates a Kerberos KRB_PRIV message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_mk_priv(
    krb5_context context,
    krb5_auth_context auth_context,
    const krb5_data *userdata,
    krb5_data *out_data,
    krb5_replay_data *replay_data)

Parameters
Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.
userdata Specifies the application data for the KRB_PRIV message.

Output
out_data Returns the KRB_PRIV message. The storage pointed to by the data field of the returned parameter should be freed by the application when it is no longer needed.
replay_data Returns replay information to the caller. This parameter is required if the KR55_AUTH_CONTEXT_RET_TIME or KR55_AUTH_CONTEXT_RET_SEQUENCE flag is set in the authentication context. Otherwise, NULL may be specified for this parameter.

Usage
The krb5_mk_priv() routine creates a KRB_PRIV message using data supplied by the application. This is similar to the krb5_mk_safe() routine but the message is encrypted and integrity-protected rather than just integrity-protected. The krb5_rd_priv() routine will decrypt and validate the message integrity. The authentication context specifies the checksum type, the data encryption type, the keyblock used to seed the checksum, the addresses of the sender and receiver, and the replay cache. The local address in the authentication context is used to create the KRB_PRIV message and must be present. The remote address is optional. The authentication context flags determine whether sequence numbers or timestamps should be used to identify the message.

The encryption type is taken from the keyblock in the authentication context. If the initial vector has been set in the authentication context, it will be used as the initialization vector for the encryption (if the encryption type supports initialization) and its contents will be replaced with the last block of encrypted data upon return.

If timestamps are used (KR55_AUTH_CONTEXT_DO_TIME is set), an entry describing the message will be entered in the replay cache so that the caller may detect if this message is sent back to him by an attacker. An error will be returned if the authentication context does not specify a replay cache.

If sequence numbers are used (KR55_AUTH_CONTEXT_DO_SEQUENCE or KR55_AUTH_CONTEXT_RET_SEQUENCE is set), then the local sequence number in the authentication context will be placed in the protected message as its sequence number.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_mk_rep

Purpose
Creates a Kerberos AP_REP message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_mk_rep (  
    krb5_context context,
    krb5_auth_context auth_context,
    krb5_data *out_data)

Parameters

Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.

Output
out_data Returns the AP_REP message. The storage pointed to by the data field of the krb5_data structure should be freed by the application when it is no longer needed.

Usage
The krb5_mk_rep() routine creates an AP_REP message using information in the authentication context. An AP_REP message is returned to the partner application after processing an AP_REQ message received from the partner application. The information in the authentication context is set by the krb5_rd_req() routine when it processes the AP_REQ message.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_mk_req

Purpose
Creates a Kerberos AP_REQ message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_mk_req (  
  krb5_context context,  
  krb5_auth_context *auth_context,  
  const krb5_flags ap_req_options,  
  char *service,  
  char *hostname,  
  krb5_data *in_data,  
  krb5_ccache ccache,  
  krb5_data *out_data)

Parameters

Input
context Specifies the Kerberos context.
ap_req_options Specifies request options as follows:

- **AP_OPTS_USE_SESSION_KEY** - Use session key instead of server key. The credentials must include a ticket which is encrypted in the session key.
- **AP_OPTS_MUTUAL_REQUIRED** - Mutual authentication required.
- **AP_OPTS_USE_SUBKEY** - Generate a subsession key from the current session key obtained from the credentials.
service Specifies the name of the service.
hostname Specifies the host name which identifies the desired service instance.
in_data Specifies the application data whose checksum is to be included in the authenticator. Specify NULL for this parameter if no checksum is to be included in the authenticator.
ccache Specifies the credentials cache which is to be used to obtain credentials to the desired service.

Input/Output
auth_context Specifies the authentication context. A new authentication context will be created and returned in this parameter if the value is NULL.

Output
out_data Returns the generated AP_REQ message. The storage pointed to by the data field in the returned krb5_data structure should be freed by the application when it is no longer needed.

Usage
The krb5_mk_req() routine generates an AP_REQ message. The checksum of the input data is included in the authenticator which is part of the AP_REQ message. This message is then sent to the partner application, which calls the krb5_rd_req() routine to extract the application data after validating the authenticity of the message. The checksum method set in the authentication context is used to generate the checksum.

The krb5_sname_to_principal() routine is called to convert the service and hostname parameters to a Kerberos principal. The krb5_get_host_realm() routine is called to convert the hostname parameter to a Kerberos realm. If the credentials cache does not already contain a service ticket for the target server, the Kerberos runtime will issue a default TGS request to obtain the credentials and store them in the cache.
Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_mk_req_extended**

**Purpose**
Creates a Kerberos AP_REQ message.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_mk_req_extended (  
    krb5_context context,  
    krb5_auth_context *auth_context,  
    const krb5_flags ap_req_options,  
    krb5_data *in_data,  
    krb5_creds *in_creds,  
    krb5_data *out_data)
```

**Parameters**

**Input**
- **context**
  Specifies the Kerberos context.
- **ap_req_options**
  Specifies request options as follows:
  - **AP_OPTS_USE_SESSION_KEY** - Use session key instead of server key. The credentials must include a ticket which is encrypted in the session key.
  - **AP_OPTS_MUTUAL_REQUIRED** - Mutual authentication required.
  - **AP_OPTS_USE_SUBKEY** - Generate a subsession key from the current session key obtained from the credentials.

- **in_data**
  Specifies the application data whose checksum is to be included in the authenticator.
- **in_creds**
  Specifies the credentials for the specified service.

**Input/Output**
- **auth_context**
  Specifies the authentication context. A new authentication context will be created and returned in this parameter if the value is NULL.

**Output**
- **out_data**
  Returns the generated AP_REQ message. The storage pointed to by the data field in the returned krb5_data structure should be freed by the application when it is no longer needed.

**Usage**
The krb5_mk_req_extended() routine is similar to the krb5_mk_req() routine but the caller passes the actual credentials as a parameter instead of letting the Kerberos runtime construct the credentials.

The krb5_mk_req_extended() routine generates an AP_REQ message. The checksum of the input data is included in the authenticator which is part of the AP_REQ message. This message is then sent to the partner application, which calls the krb5_rd_req() routine to extract the application data after validating the authenticity of the message. The checksum method set in the authentication context is used to generate the checksum.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_mk_safe

Purpose
Creates a Kerberos KRB_SAFE message.

Format
```c
#include <krb5/krb5.h>

krb5_error_code krb5_mk_safe (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    const krb5_data *userdata,  
    krb5_data *out_data,  
    krb5_replay_data *replay_data)
```

Parameters

Input
- `context` Specifies the Kerberos context.
- `auth_context` Specifies the authentication context.
- `userdata` Specifies the application data for the KRB_SAFE message.

Output
- `out_data` Returns the KRB_SAFE message. The storage pointed to by the data field of the returned parameter should be freed by the application when it is no longer needed.
- `replay_data` Returns replay information to the caller. This parameter is required if the `KRB5_AUTH_CONTEXT_RET_TIME` or `KRB5_AUTH_CONTEXT_RET_SEQUENCE` flag is set in the authentication context. Otherwise, `NULL` may be specified for this parameter.

Usage
The `krb5_mk_safe()` routine creates a KRB_SAFE message using data supplied by the application. Message created by the `krb5_mk_safe()` routine are integrity-protected. The `krb5_rd_safe()` routine will return an error if the message has been modified. The authentication context specifies the checksum type, the keyblock used to seed the checksum, the addresses of the sender and receiver, and the replay cache. The local address in the authentication context is used to create the KRB_SAFE message and must be present. The remote address is optional. The authentication context flags determine whether sequence numbers or timestamps should be used to identify the message.

If timestamps are used (KRB5_AUTH_CONTEXT_DO_TIME is set), an entry describing the message will be entered in the replay cache so that the caller may detect if this message is sent back to him by an attacker. An error will be returned if the authentication context does not specify a replay cache.

If sequence numbers are used (KRB5_AUTH_CONTEXT_DO_SEQUENCE or KRB5_AUTH_CONTEXT_RET_SEQUENCE is set), then the local sequence number in the authentication context will be placed in the protected message as its sequence number.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_os_hostaddr

Purpose
Returns the network addresses used by a specific host system.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_os_hostaddr (
    krb5_context context,
    const char *host,
    krb5_address ***addrs)

Parameters
Input
context Specifies the Kerberos context.
host Specifies the name of the host system. The name must be acceptable for use with the gethostbyname() system function.

Output
addrs Returns an array of krb5_address pointers. The last entry in the array will be a NULL pointer. The krb5_free_addresses() routine should be called to release the address array when it is no longer needed.

Usage
The krb5_os_hostaddr() routine returns the network addresses which are available on the specified host system. At the present time, only the AF_INET address family is supported and the gethostbyname() system function will be used to lookup the addresses assigned to the specified host.

Return Values
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_os_localaddr**

**Purpose**
Returns the network addresses used by the local system.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_os_localaddr (krb5_context context, krb5_address ***addrs)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.

**Output**
- `addrs` Returns an array of `krb5_address` pointers. The last entry in the array will be a NULL pointer. The `krb5_free_addresses()` routine should be called to release the address array when it is no longer needed.

**Usage**
The `krb5_os_localaddr()` routine returns the network addresses which are available on the local system. At the present time, only the `AF_INET` address family is supported.

**Return Values**
The return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_parse_name

Purpose
Creates a Kerberos principal from a text string.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_parse_name (  
    krb5_context context,  
    const char *name,  
    krb5_principal *principal)

Parameters
Input
context  
    Specifies the Kerberos context.
name  
    Specifies the string to be parsed. The string must be in the format name@realm.

Output
principal  
    Returns the Kerberos principal. The krb5_free_principal() routine should be called to release
    the principal when it is no longer needed.

Usage
The krb5_parse_name() routine will convert a text string into a Kerberos principal. The string must be in the format
name@realm. If the realm is not specified, the default realm will be used. Each forward slash in the name will start a new
name component unless it is escaped by preceding the forward slash with a backslash. Forward slashes in the realm are not
treated as component separators and are copied unchanged.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Comparates two Kerberos principals.

**Format**
```
#include <krb5/krb5.h>

krb5_boolean krb5_principal_compare(
    krb5_context context,
    krb5_const_principal princ1,
    krb5_const_principal princ2)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `princ1` Specifies the first principal to be compared.
- `princ2` Specifies the second principal to be compared.

**Usage**
The `krb5_principal_compare()` routine compares two Kerberos principals.

**Return Values**
The function return value will be **TRUE** if the principals are the same and **FALSE** if they are not the same.
**krb5_random_confounder**

**Purpose**
Creates a random confounder.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_random_confounder (
    krb5_context context,
    int buffer_size,
    krb5_pointer output_buffer)
```

**Parameters**

**Input**
- `context`: Specifies the Kerberos context.
- `buffer_size`: Specifies the size of the output buffer.

**Output**
- `output_buffer`: Specifies the buffer to receive the confounder.

**Usage**
The `krb5_random_confounder()` routine creates a random value which can be used as a confounder when encrypting data. A confounder is used to initialize the encryption block chaining value so that the encrypted result is different each time a data value is encrypted even when the data value and encryption key are not changed.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_close**

**Purpose**
Closes a replay cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_close (
    krb5_context context,
    krb5_rcache rcache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Usage**
The `krb5_rc_close()` routine closes a replay cache. The cache handle may not be used once this routine completes.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Resolves the default replay cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_default (  
    krb5_context context,  
    krb5_rcache *rcache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.

**Output**
- `rcache` Returns the replay cache handle.

**Usage**
The `krb5_rc_default()` routine resolves the default replay cache and returns a handle which can be used to access the table. This is equivalent to calling the `krb5_rc_resolve()` routine with the name returned by the `krb5_rc_default_name()` routine.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_default_name**

**Purpose**
Returns the default replay cache name.

**Format**
```c
#include <krb5/krb5.h>

char *krb5_rc_default_name (krb5_context context)
```

**Parameters**
- **Input**
  - `context` Specifies the Kerberos context.

**Usage**
The `krb5_rc_default_name()` routine returns the name of the default replay cache for the current user. The `KRB5RCACHENAME` environment variable defines the default replay cache name.

**Return Values**
The function return value will be the default replay cache name or NULL if the default name has not been set. The return value will be the address of a read-only string and must not be freed by the application.
**krb5_rc_destroy**

**Purpose**
Deletes a replay cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_destroy (  
    krb5_context context,  
    krb5_rcache rcache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Usage**
The `krb5_rc_destroy()` routine closes and deletes a replay cache. The cache handle may not be used after this routine completes.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_expunge**

**Purpose**
Deletes expired entries from the replay cache.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_rc_expunge (  
    krb5_context context,  
    krb5_rcache rcache)  
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Usage**
The `krb5_rc_expunge()` routine will delete expired entries from the replay cache. The entry lifespan is set by the `krb5_rc_initialize()` routine. This routine should be called periodically to clean up the replay cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_free_entry_contents**

**Purpose**
Releases the storage associated with a replay cache entry.

**Format**
```c
#include <krb5/krb5.h>

void krb5_rc_free_entry_contents (
    krb5_context context,
    krb5_donot_replay *entry)
```

**Parameters**

**Input**
- *context* Specifies the Kerberos context.
- *entry* Specifies the entry to be released.

**Usage**
The `krb5_rc_free_entry_contents()` will release the contents of a replay entry. The `krb5_donot_replay` structure itself will not be released.
**krb5_rc_get_lifespan**

**Purpose**
Returns the authenticator lifespan for entries in the replay cache.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_get_lifespan(
    krb5_context context,
    krb5_rcache rcache,
    krb5_deltat *span)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Output**
- `span` Returns the authenticator lifespan in seconds.

**Usage**

The `krb5_rc_get_lifespan()` routine returns the authenticator lifespan that was set by the `krb5_rc_initialize()` routine.

**Return Values**

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Returns the replay cache name.

**Format**

```c
#include <krb5/krb5.h>

char *krb5_rc_get_name (
    krb5_context context,
    krb5_rcache rcache)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Usage**

The `krb5_rc_get_name()` routine returns the name of the replay cache. The returned name will not include the replay cache type prefix.

**Return Values**

The function return value will be the address of the replay cache name. This is a read-only value and must not be freed by the application.
krb5_rc_get_type

Purpose
Returns the replay cache type.

Format
```
#include <krb5/krb5.h>

char *krb5_rc_get_type (  
    krb5_context context,  
    krb5_rcache rcache)  
```

Parameters

Input
- `context`: Specifies the Kerberos context.
- `rcache`: Specifies the replay cache handle.

Usage
The `krb5_rc_get_type()` routine returns the replay cache type.

Return Values
The function return value will be the address of the replay cache type. This is a read-only value and must not be freed by the application.
Purpose
Initializes the replay cache.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rc_initialize (
    krb5_context context,
    krb5_rcache rcache,
    krb5_deltat span)

Parameters

Input
context Specifies the Kerberos context.
rcache Specifies the replay cache handle.
span Specifies the authenticator lifespan in seconds.

Usage
The krb5_rc_initialize() routine will initialize a replay cache. Any existing cache entries will be deleted. The authenticator lifespan indicates how long an authenticator remains valid. Once an authenticator has expired, its replay cache entry can be deleted by calling the krb5_rc_expunge() routine.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_recover**

**Purpose**
Recovers the replay cache.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_rc_recover (
    krb5_context context,
    krb5_rcache rcache)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.

**Usage**
The `krb5_rc_recover()` routine reads a replay cache into storage after the application has been restarted. Either `krb5_rc_recover()` or `krb5_rc_initialize()` must be called before any replay entries can be added to the replay cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
Purpose
Defines a new replay cache type.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rc_register_type (  
    krb5_context context,  
    krb5_rc_ops *ops)

Parameters

Input
context       Specifies the Kerberos context.
ops           Specifies the replay cache operations vector. This vector defines the routines which will be
called to perform the various replay cache operations for the new type.

Usage
The krb5_rc_register_type() routine registers a new replay cache type. An error will be returned if the replay cache type has
already been registered. Once the new type is registered, it can be used by any thread in the current process. The type is not
known outside the current process and will no longer be registered when the application ends.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_resolve**

**Purpose**
Resolves a replay cache name.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_resolve (  
    krb5_context context,  
    krb5_rcache *rcache,  
    char *name)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `name` Specifies the replay cache name in the format `type:name`. The type must be a registered replay cache type and the name must uniquely identify a particular replay cache of the specified type.

**Output**
- `rcache` Returns the replay cache handle.

**Usage**
The `krb5_rc_resolve()` routine resolves a replay cache name and returns a handle which can be used to access the cache. The Kerberos runtime supports two replay cache types: `dfl` and `mem`. Additional replay cache types can be registered by the application by calling the `krb5_rc_register_type()` routine. If no type is specified, the default is `dfl`.

After successfully calling `krb5_rc_resolve()`, the application should call either the `krb5_rc_recover()` or the `krb5_rc_initialize()` routine.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_rc_store**

**Purpose**
Stores a new entry in the replay cache.

**Format**
```c
#include <krb5/krb5.h>

krb5_error_code krb5_rc_store (  
    krb5_context context,  
    krb5_rcache rcache,  
    krb5_donot_replay *replay)
```

**Parameters**
**Input**
- `context` Specifies the Kerberos context.
- `rcache` Specifies the replay cache handle.
- `replay` Specifies the replay entry.

**Usage**
The `krb5_rc_store()` routine stores a new entry in the replay cache after verifying that the entry is not already in the cache. The `krb5_auth_to_rep()` routine can be used to create a replay entry from a Kerberos authenticator. The `krb5_rc_expunge()` routine should be called periodically to purge expired entries from the replay cache.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_rd_error

Purpose
Processes a Kerberos KRB_ERROR message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rd_error (  
   krb5_context context,  
   const krb5_data *enc_err,  
   krb5_error **dec_err)  

Parameters
Input
context Specifies the Kerberos context.
en_enc_err Specifies the error message created by the krb5_mk_error() routine.

Output
dec_err Returns the decoded error message. The krb5_free_error() routine should be called to release the krb5_error structure when it is no longer needed.

Usage
The krb5_rd_error() routine processes a KRB_ERROR message created by the krb5_mk_error() routine and returns a krb5_error structure.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_rd_priv

Purpose
Processes a Kerberos KRB_PRIV message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rd_priv (
    krb5_context context,
    krb5_auth_context auth_context,
    const krb5_data *in_data,
    krb5_data *out_data,
    krb5_replay_data *replay_data)

Parameters

Input
context Specifies the Kerberos context.
auth_context Specifies the authentication context.
in_data Specifies the buffer containing the KRB_PRIV message

Output
out_data Returns the application data. The application should free the storage pointed to by the data field of the returned parameter when it is no longer needed.
replay_data Returns replay information to the caller. This parameter is required if the
    KR5_AUTH_CONTEXT_RET_TIME or KR5_AUTH_CONTEXT_RET_SEQUENCE flag is set in the authentication context. Otherwise, NULL may be specified for this parameter.

Usage
The krb5_rd_priv() routine processes a KRB_PRIV message and extracts the application data after verifying its integrity. If timestamps are being used, the message will be stored in the replay cache associated with the authentication context.

The keyblock which is used for decrypting the data and for verifying message integrity is obtained from the authentication context. If the initialization vector in the authentication context has been set, it will be used to initialize the decryption (if the encryption type supports initialization) and its contents will be replaced with the last block of encrypted data in the message upon return.

The remote address in the authentication context must be present. It specifies the address of the sender and must be of type ADDRTYPE_ADDRPORT. An error will be returned if the address in the message does not match the remote address in the authentication context.

The local address in the authentication context is optional. If it is present, then it must match the receiver address in the message.

Otherwise, the receiver message in the message must match one of the local addresses returned by the krb5_os_localaddr() routine.

If message sequence numbers are being used (KR5_AUTH_CONTEXT_DO_SEQUENCE is set in the authentication context), the remote sequence number in the authentication context must match the sequence number in the message.

If timestamps are being used (KR5_AUTH_CONTEXT_DO_TIME is set in the authentication context), the timestamp in the message must be within the Kerberos clock skew for the current time. In addition, the message must not be found in the replay cache obtained from the authentication context.
Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_rd_rep

Purpose
Processes a Kerberos AP_REP message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rd_rep (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    const krb5_data *in_data,  
    krb5_ap_rep_enc_part **reply)

Parameters
Input
context Specifies the Kerberos context.
in_data Specifies the buffer containing the AP_REP message.

Input/Output
auth_context Specifies the authentication context.

Output
reply Returns the decrypted reply data. The krb5_free_ap_rep_enc_part() routine should be called to release the reply when it is no longer needed.

Usage
The krb5_rd_rep() routine processes an AP_REP message created by the krb5_mk_rep() routine. The authentication context is updated with sequencing information obtained from the reply message.

Return Values
The function return value is zero if no errors occurred. Otherwise, it is a Kerberos error code.
 krb5_rd_req

Purpose
Processes a Kerberos AP_REQ message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rd_req (  
    krb5_context context,  
    krb5_auth_context *auth_context,  
    const krb5_data *in_data,  
    krb5_const_principal server,  
    krb5_keytab keytab,  
    krb5_flags *ap_req_options,  
    krb5_ticket **ticket)

Parameters

Input
context Specifies the Kerberos context.
in_data Specifies the buffer containing the AP_REQ message.
server Specifies the server name. The server principal in the AP_REQ must be the same as the principal specified by this parameter. Specify NULL if any server principal is acceptable.
keytab Specifies the key table which contains the server key. The default key table will be used if NULL is specified for this parameter.

Input/Output
auth_context Specifies the authentication context. A new authentication context will be created and returned in this parameter if the value is NULL.

Output
ap_req_options Returns the options from the AP_REQ message. Specify NULL for this parameter if the options are not needed.
ticket Returns the ticket from the AP_REQ message. Specify NULL for this parameter if the ticket is not needed. The krb5_free_ticket() routine should be called to release the ticket when it is no longer needed.

Usage
The krb5_rd_req() routine processes an AP_REQ message generated by the partner application. The authenticator is extracted, validated, and stored in the authentication context. If the server parameter is not NULL and no replay cache is associated with the authentication context, the Kerberos runtime will create a replay cache and store the cache handle in the authentication context.

If the authentication context contains a keyblock, it will be used to decrypt the ticket in the AP_REQ message. This is useful for user-to-user authentication. If the authentication context does not contain a keyblock, the key table specified on the function call will be used to obtain the decryption key.

The client in the authenticator must match the client in the ticket. If the remote address has been set in the authentication context, the request must have come from that address. If a replay cache handle is stored in the authentication context, the new authenticator is stored in the cache after checking for replay.

If no errors are detected, the authenticator, subsession key, and remote sequence number are stored in the authentication context. If AP_OPTS_MUTUAL_REQUIRED is specified in the AP_REQ message, the local sequence number is XORed with the remote sequence number.
Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
Purpose
Processes a Kerberos KRB_SAFE message.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_rd_safe (  
    krb5_context context,  
    krb5_auth_context auth_context,  
    const krb5_data *in_data,  
    krb5_data *out_data,  
    krb5_replay_data *replay_data)

Parameters
Input
context    Specifies the Kerberos context.
auth_context    Specifies the authentication context.
in_data    Specifies the buffer containing the KRB_SAFE message
Output
out_data Returns the application data. The application should free the storage pointed to by the data field of the returned parameter when it is no longer needed.
replay_data Returns replay information to the caller. This parameter is required if the
    KRB5_AUTH_CONTEXT_RET_TIME or KRB5_AUTH_CONTEXT_RET_SEQUENCE flag is set in the authentication context. Otherwise, NULL may be specified for this parameter.

Usage
The krb5_rd_safe() routine processes a KRB_SAFE message and extracts the application data after verifying its integrity. If timestamps are being used, the message will be stored in the replay cache associated with the authentication context.

The keyblock which is used for verifying message integrity is obtained from the authentication context. The first non-NULL keyblock is used by checking the local_subkey, remote_subkey, or keyblock, in that order. The remote address in the authentication context must be present. It specifies the address of the sender and must be of type ADDRTYPE_ADDRPORT. An error will be returned if the address in the message does not match the remote address in the authentication context. The local address in the authentication context is optional. If it is present, then it must match the receiver address in the message. Otherwise, the receiver message in the message must match one of the local addresses returned by the krb5_os_localaddr() routine.

If message sequence numbers are being used (KRB5_AUTH_CONTEXT_DO_SEQUENCE is set in the authentication context), the remote sequence number in the authentication context must match the sequence number in the message. If timestamps are being used (KRB5_AUTH_CONTEXT_DO_TIME is set in the authentication context), the timestamp in the message must be within the Kerberos clock skew for the current time. In addition, the message must not be found in the replay cache obtained from the authentication context.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
krb5_realm_compare

Purpose
Compares the realms of two principals.

Format
#include <krb5/krb5.h>

krb5_boolean krb5_realm_compare (
    krb5_context context,
    krb5_const_principal princ1,
    krb5_const_principal princ2)

Parameters
Input
context Specifies the Kerberos context.
princ1 Specifies the first principal to be compared.
princ2 Specifies the second principal to be compared.

Usage
The krb5_realm_compare() routine will compare the realms for two principals.

Return Values
The function return value will be TRUE if the realms are the same and FALSE if they are not the same.
**Purpose**
Sets the files to be processed for Kerberos configuration requests.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_set_config_files (  
    krb5_context context,  
    const char **names)  
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `names` Specifies an array of file names. The last entry in the array must be a NULL pointer.

**Usage**
The `krb5_set_config_files()` routine specifies the names of the files which are to be processed to obtain the Kerberos configuration. This replaces the configuration files which were used to create the Kerberos context. Changing the configuration files will not affect context values which have already been set from the old configuration files.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Sets the default encryption types used when requesting an initial ticket from the KDC.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_set_default_in_tkt_ktypes (
    krb5_context context, 
    const krb5_enctype *ktypes)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `ktypes` Specifies an array of `krb5_enctype` values which will be used when requesting an initial ticket. The last element in the array must be set to `ENCTYPE_NULL`. The following symbolic definitions are provided for specifying the encryption types:
  - `ENCTYPE_DES_CBC_CRC` - DES encryption with a CRC checksum
  - `ENCTYPE_DES_CBC_MD4` - DES encryption with an MD4 checksum
  - `ENCTYPE_DES_CBC_MD5` - DES encryption with an MD5 checksum
  - `ENCTYPE_DES_CBC_RAW` - DES encryption with no checksum

**Usage**

The `krb5_set_default_in_tkt_ktypes()` routine sets the default encryption types used when requesting the initial ticket from the KDC. The first encryption type specified is used for generating random keys, so it must be an encryption type which is supported by the KDC. In order to interoperate with older Kerberos V5 servers, you should specify `ENCTYPE_DES_CBC_CRC` as the first encryption type.

The encryption types specified will override any values specified by the `default_tkt_enctypes` entry in the Kerberos configuration file.

**Return Values**

The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_set_default_realm

Purpose
Sets the default realm for the local system.

Format

```c
#include <krb5/krb5.h>

krb5_error_code krb5_set_default_realm (  
    krb5_context context,  
    char *realm)
```

Parameters

Input

- `context` Specifies the Kerberos context.
- `realm` Specifies the name for the default realm.

Usage

The `krb5_set_default_realm()` routine sets the default realm for the specified Kerberos context. This overrides the default realm set by the Kerberos configuration file. The realm set by `krb5_set_default_realm()` applies only to the Kerberos context specified by the context parameter.

Return Values

The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**krb5_set_default_tgs_ktypes**

**Purpose**
Sets the default encryption types used when requesting a service ticket from the KDC.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_set_default_tgs_ktypes(
    krb5_context context,
    const krb5_enctype *ktypes)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.
- `ktypes` Specifies an array of `krb5_enctype` values which will be used when requesting a service ticket. The last element in the array must be set to `ENCTYPE_NULL`. The following symbolic definitions are provided for specifying the encryption types:
  - `ENCTYPE_DES_CBC_CRC` - DES encryption with a CRC checksum
  - `ENCTYPE_DES_CBC_MD4` - DES encryption with an MD4 checksum
  - `ENCTYPE_DES_CBC_MD5` - DES encryption with an MD5 checksum
  - `ENCTYPE_DES_CBC_RAW` - DES encryption with no checksum

**Usage**
The `krb5_set_default_tgs_ktypes()` routine sets the default encryption types used when requesting a service ticket from the KDC. The first encryption type specified is used for generating random keys, so it must be an encryption type which is supported by the KDC. In order to interoperate with older Kerberos V5 servers, you should specify `ENCTYPE_DES_CBC_CRC` as the first encryption type.

The encryption types specified will override any values specified by the `default_tgs_enctypes` entry in the Kerberos configuration file.

**Return Values**
The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_sname_to_principal

Purpose
Converts a service name to a Kerberos principal.

Format
```
#include <krb5/krb5.h>

krb5_error_code krb5_sname_to_principal (
    krb5_context context,
    const char *hostname,
    const char *sname,
    host_int32 type,
    krb5_principal *ret_princ)
```

Parameters

Input
- `context` Specifies the Kerberos context.
- `hostname` Specifies the host containing the desired service instance. The local host will be used if `NULL` is specified for this parameter.
- `sname` Specifies the service name. The service name will be set to host if `NULL` is specified for this parameter.
- `type` Specifies the type of host name provided as follows:
  - `KRB5_NT_SRV_HST` - A DNS host name has been provided. The Kerberos runtime will lookup the address assigned to the host name and then do a reverse-lookup to get the primary host name for that address. The resulting host name will then be converted to lowercase.
  - `KRB5_NT_UNKNOWN` - The host name type is unknown. No translation will be performed on the specified host name and it will be used as-is.

Output
- `ret_princ` Returns the generated principal. The `krb5_free_principal()` routine should be called to release the principal when it is no longer needed.

Usage
The `krb5_sname_to_principal()` routine generates a Kerberos principal from a service name and a host name. The principal name will be in the format `sname/hostname@realm`. The realm name that corresponds to the host name is obtained by calling the `krb5_get_host_realm()` routine.

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Returns the Kerberos runtime release level.

**Format**

```c
#include <krb5/krb5.h>

void krb5_svc_get_level (
    krb5_int32 *version,
    krb5_int32 *release,
    krb5_int32 *level)
```

**Parameters**

**Output**

- **version**: Returns the version number.
- **release**: Returns the release number.
- **level**: Returns the modification level.

**Usage**

The `krb5_svc_get_level()` routine returns the current Kerberos runtime release level. This is Version 5, Release 1, Level 0.
**Purpose**

Returns a printable text message corresponding to a Kerberos error code.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_svc_get_msg (  
    krb5_ui_4 error_code,  
    char **msg_text)
```

**Parameters**

**Input**

- `error_code` Specifies the Kerberos error code.

**Output**

- `msg_text` Returns the character string describing the error code. The caller should free the character string returned by this parameter when it is no longer needed.

**Usage**

The `krb5_svc_get_msg()` routine returns a printable character string which describes the error represented by the supplied error code. This allows the application to log the error or display it to the user.

**Return Values**

The function return value will be zero if no error occurred. Otherwise, it will be a Kerberos error code.
krb5_timeofday

Purpose
Returns the current time of day in seconds since the epoch.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_timeofday (  
    krb5_context context,  
    krb5_int32 *seconds)

Parameters

Input
context  Specifies the Kerberos context.

Output
seconds  Returns the number of seconds since the epoch.

Usage
The krb5_timeofday() routine returns the number of seconds since the epoch (January 1, 1970). The returned time is not adjusted for local time differences.

Return Values
The function return value is zero if no error occurred. Otherwise, it is a Kerberos error code.
krb5_unparse_name

Purpose
Converts a Kerberos principal to a text string.

Format
#include <krb5/krb5.h>

krb5_error_code krb5_unparse_name (
    krb5_context context,
    krb5_const_principal principal,
    char **name)

Parameters
Input
context            Specifies the Kerberos context.
principal          Specifies the principal to be converted.

Output
name               Returns the text string for the principal in the format name@realm. The application should free the text string when it is no longer needed.

Usage
The krb5_unparse_name() routine will create a text string from a Kerberos principal. The string will be in the format name@realm with the name components separated by forward slashes. If a forward slash occurs within a name component, it will be escaped in the generated string by preceding the forward slash (/) with a backslash (\).

Return Values
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**
Converts a Kerberos principal to a text string.

**Format**
```
#include <krb5/krb5.h>

krb5_error_code krb5_unparse_name_ext (
    krb5_context context,
    krb5_const_principal principal,
    char **name,
    int *size)
```

**Parameters**

**Input**
- `context` Specifies the Kerberos context.
- `principal` Specifies the principal to be converted.

**Input/Output**
- `name` Returns the text string for the principal in the format `name@realm`. The application should free the text string when it is no longer needed. If the `name` parameter contains a NULL address upon entry, `krb5_unparse_name_ext()` will allocate a new buffer and return the address in the `name` parameter and the size in the `size` parameter. Otherwise, the `name` parameter must contain the address of an existing buffer and the `size` parameter must contain the size of this buffer. The `krb5_unparse_name_ext()` will reallocate the buffer if necessary and return the updated values in the `name` and `size` parameters.
- `size` Specifies the size of the buffer specified by the `name` parameter.

**Usage**
The `krb5_unparse_name_ext()` routine will create a text string from a Kerberos principal. The string will be in the format `name@realm` with the name components separated by forward slashes. If a forward slash occurs within a name component, it will be escaped in the generated string by preceding the forward slash with a backslash.

The `krb5_unparse_name_ext()` routine is similar to the `krb5_unparse_name()` routine but it allows the application to avoid the overhead of repeatedly allocating the output string when a large number of conversions need to be performed.

**Return Values**
The function return value will be zero if no errors occurred. Otherwise, it will be a Kerberos error code.
**Purpose**

Returns the current time of day in seconds and microseconds since the epoch.

**Format**

```c
#include <krb5/krb5.h>

krb5_error_code krb5_us_timeofday (
    krb5_context context,
    krb5_int32 *seconds,  
    krb5_int32 *useconds)
```

**Parameters**

**Input**

- `context` Specifies the Kerberos context.

**Output**

- `seconds` Returns the seconds portion of the result.
- `useconds` Returns the microseconds portion of the result.

**Usage**

The `krb5_us_timeofday()` routine returns the number of seconds and microseconds since the epoch (January 1, 1970). The returned time is not adjusted for local time differences.

**Return Values**

The function return value is zero if no error occurred. Otherwise, it is a Kerberos error code.
krb5_us_timeofday
Chapter 8. DCE Utility Routines

The DCE utility (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.

**DCE configuration routines**

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

**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 stand-alone programs that do not involve remote procedure calls (RPC).

**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.

**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.

**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 sets of routines above, see the introductory section that appears before each set.
The DCE Attribute Interface Routines

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 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 re-allocating.

- **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.

- **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.

- **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.
dce_attr_sch_bind

Purpose
Return an opaque handle to a schema object.

Format
#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.

Possible status codes are:

dce_attr_s_bad_name
DCE attribute names specified are not valid.

sec_login_s_no_current_context
There was no current context to retrieve.

rpc_s_entry_not_found
Inquiry context is not valid.

rpc_s_no_more_bindings
All compatible bindings have been exported.

dce_attr_s_unknown_auth_info_type
When dce_attr_xxx_bind was started, an unknown auth_info type was specified. Refer to dce_attr_base.h for all defined auth_info types.

dce_attr_s_no_memory
Not enough memory to perform the operation.

Usage

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.

Related Information
dce_attr_sch_bind

Routines:  dce_attr_sch_bind_free

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

Purpose
Releases an opaque handle of type dce_attr_sch_handle_t to a schema object.

Format
#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.

Usage
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.

Related Information
Routines:  dce_attr_sch_bind

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

Purpose
Create a schema entry in a schema bound to by a previous dce_attr_sch_bind() routine.

Format
```
#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.
  Possible status codes are:
  - dce_attr_s_bad_binding: The attribute binding handle specified was invalid.

Usage
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.

Related Information

Routines
dce_attr_sch_delete_entry dce_attr_sch_update

Files
/usr/include/dce/dce_attr_base.idl The idl file from which dce/dce_attr_base.h was derived.
### dce_attr_sch_cursor_alloc

#### Purpose
Allocates resources to a cursor used with the `dce_attr_sch_scan` call.

#### Format
```
#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.

Possible status codes are:
- `dce_attr_s_no_memory` Not enough memory to perform the operation.

#### Usage
The `dce_attr_sch_cursor_alloc()` call allocates resources to a cursor used with the `dce_attr_sch_scan` call. 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()` call requires appropriate permissions on the schema object. These permissions are managed by the target server.

#### Related Information

**Routines**
- `dce_attr_sch_cursor_init`
- `dce_attr_sch_cursor_release`
- `dce_attr_sch_scan`

**Files**
- `/usr/include/dce/dce_attr_base.idl` The idl file from which `dce/dce_attr_base.h` was derived.
**dce_attr_sch_cursor_init**

**Purpose**

Initialize and allocate a cursor used with the `dce_attr_sch_scan` call.

**Format**

```c
#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 schema.

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

**Usage**

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

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

**Permissions Required**

None.

**Related Information**

**Routines**

- `dce_attr_sch_cursor_release`
- `dce_attr_sch_scan`
- `dce_attr_sch_cursor_allocate`
Files

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

Purpose
Release states associated with a cursor that has been allocated with either dce_attr_sch_cursor_init() or dce_attr_sch_cursor_alloc().

Format
#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.

Usage
The dce_attr_sch_cursor_init() routine releases the resources allocated to a cursor that has been allocated by either a 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.

Related Information
Routines
dce_attr_sch_cursor_init
dce_attr_sch_cursor_reset
dce_attr_sch_scan
dce_attr_sch_cursor_alloc

Files
/usr/include/dce/dce_attr_base.idl The idl file from which dce/dce_attr_base.h was derived.
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()`.

Format
```c
#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.

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

Permissions Required
None.

Related Information

Routines
- `dce_attr_sch_cursor_init`
- `dce_attr_sch_cursor_alloc`
- `dce_attr_sch_scan`

Files
- `/usr/include/dce/dce_attr_sch.idl` The idl file from which `dce/dce_attr_sch.h` was derived.
Purpose
Delete a schema entry.

Format
#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.

Usage
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 appropriate permissions on the schema object. These permissions are managed by the target server.

Related Information
Routines
dce_attr_sch_create_entry dce_attr_sch_update_entry

Files
/usr/include/dce/dce_attr_base.idl Theidl file from which dce/dce_attr_base.h was derived.
dce_attr_sch_get_acl_mgrs

Purpose
Retrieve the manager types of the ACLs protecting the objects dominated by a named schema.

Format
#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_manager_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.

Usage
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. Then, using the dce_attr_sch_aclmgr_strings() routine, they can determine how to format for display the permissions supported by that ACL manager type.

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.

Related Information
dce_attr_sch_get_acl_mgrs

Routines:  dce_attr_sch_aclmgr_strings

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

**Purpose**
Read a schema entry identified by UUID.

**Format**
```c
#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.

Possible status codes are:
- `dce_attr_s_bad_binding` The attribute binding handle specified was invalid.

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

**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.

**Related Information**

**Routines**
- `dce_attr_sch_lookup_by_name`
- `dce_attr_sch_scan`

**Files**
- `/usr/include/dce/dce_attr_base.idl` The idl file from which `dce/dce_attr_base.h` was derived.
**dce_attr_sch_lookup_by_name**

**Purpose**
Read a schema entry identified by name.

**Format**
```c
#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.

Possible status codes are:
- `dce_attr_s_bad_binding` The attribute binding handle specified was invalid.

**Usage**
The `dce_attr_schLookup_by_name()` routine reads a schema entry identified by name. This routine is useful for use with an interactive editor.

**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.

**Related Information**

**Routines**
- `dce_attr_sch_lookup_by_id`
- `dce_attr_sch_scan`

**Files**
- `/usr/include/dce/dce_attr_base.idl` The idl file from which `dce/dce_attr_base.h` was derived.
dce_attr_sch_scan

Purpose
Read a specified number of schema entries.

Format
#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.

Possible status codes are:
dce_attr_s_bad_binding The attribute binding handle specified was invalid.
dce_attr_s_bad_cursor The attribute cursor specified was not valid.

Usage
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() call. 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 call returns the message no_more_entries.

This routine is useful as a browser.
dce_attr_sch_scan

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

Related Information

Routines
- `dce_attr_sch_cursor_init`
- `dce_attr_sch_cursor_alloc`
- `dce_attr_sch_cursor_release`

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

**Purpose**
Update a schema entry.

**Format**
```c
#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.

**Usage**
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 to make any changes to these components, the schema entry must be deleted (which deletes all attribute instances of that type) and recreated. These components are listed below:
- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- 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.

**Related Information**
Routines

dce_attr_sch_delete_entry
dce_attr_sch_create_entry

Files

/usr/include/dce/dce_attr_base.idl

The idl file from which dce/dce_attr_base.h was derived.
The DCE Configuration Routines

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 resides on each 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 find the principal name of the host and 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 default pathname is:

```
/usr/lib/nls/csr/code_set_registry.db
```

The Security Service component on each DCE machine must be able to find out, by strictly local means, its system host name, the principal name of the host system, and its cell's name. The DCE configuration routines exist primarily to enable Security components to find this information. 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 “hostname” as used throughout this section refers to the DCE hostname (that is, the machine's `l.*cellname/host_directory/hostname` entry in the CDS namespace), and not, for example, its DNS (Domain Name Service) hostname, which could be quite different from the DCE name.

**Notes:**

1. Cell names are case insensitive. Cell names returned by DCE APIs default to lowercase. If you are doing string comparisons on cell names returned by DCE APIs, you must convert your strings to lowercase before performing the comparisons.
2. The storage for a returned name string is allocated by `malloc`, and must be freed by the caller of `dce_cf_get_cell_name`.

The DCE configuration routines are:

```c

int dce_cf_binding_entry_from_host();
    Returns the host binding entry name.

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

int 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).

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

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

char *dce_cf_get_cell_name();
    Returns the primary cell name for the local cell.

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

char *dce_cf_get_host_name();
    Returns the hostname relative to a local cell.

char *dce_cf_prin_name_from_host();
    Returns the host's principal name.

int dce_cf_profile_entry_from_host
    Returns the host's profile entry.

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

DCE Configuration File

`/opt/dcelocal/dce_cf.db` is the name of a machine's local DCE configuration file.

The format of the configuration file follows these rules:

- Each entry 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 for the machine where the configuration file resides. A sample configuration file could have the following contents:

```
cellname /.qc@1>4osf.org
hostname hosts/brazil
```

which would identify the host **brazil** in the **osf.org** cell.

- Text characterized by the following is ignored:
  - Lines that do not conform to the format described above.
  - 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.

**Cautions:**

The z/OS DCE implementation of the DCE configuration routines will accept only lines (in the configuration file) whose length is less than 1024 characters.

If a tag occurs more than once in the input, the routines will recognize only the first occurrence.

**DCE Configuration API Output**

The DCE configuration routines return relative names (that is, those without global or cell-relative prefixes), such as:

```
host_directory/hostname
```

or:

```
principalname
```

where:

- *host_directory* is usually hosts.

However, the DCE NSI (Name Service Interface) routines require names passed to them to be expressed either in a cell-relative form, such as:
```
./host_directory/hostname
```

or as global names, with the global root prefix /.* and the cell name, such as:
```
./.*cellname/host_directory/hostname
```

Therefore, an application must add either the cell-relative (/.* or correct global (/.*cellname) prefix 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, 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.
dce_cf_binding_entry_from_host

Purpose
Returns the host binding entry name.

Format
#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 names are case-sensitive. If NULL, the configuration
                  file is searched for the host name; if that name is found, it 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 was completed successfully and, if not, the reason.

  Possible status codes and their meanings are:
  dce_cf_st_ok    Operation successfully completed.
  dce_cf_e_file_open    File open error.
  dce_cf_e_no_mem     No storage available.
  dce_cf_e_no_match    No match for hostname in DCE configuration file.
  dce_cf_e_invalid_parameter Parameter not valid.

Usage
The dce_cf_binding_entry_from_host API 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.

Return Values
None

Related Information
Routines
dce_cf_find_name_by_key    dce_cf_get_host_name    dce_cf_prin_name_from_host
dce_cf_get_cell_name
dce_cf_binding_entry_from_host

Files
/opt/dcelocal/dce_cf.db The local DCE configuration file of the machine.

Books
• [z/OS DCE Administration Guide](#)
**dce_cf_dced_entry_from_host**

**Purpose**

Returns the dced entry name on a host.

**Format**

```c
#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.

Possible status codes and their meanings are as follows:

- `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 hostname entry in the DCE configuration file.

**Usage**

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 dcep. For example, the string name `/.:/hosts/vineyard/config/srvrconf/dtsd` names the server configuration data for the DTS server on the host `vineyard`.

**Return Values**

None.
### Related Information

**Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dce_cf_binding_entry_from_host</code></td>
<td><code>dce_cf_get_cell_name</code></td>
<td><code>dce_cf_prin_name_from_host</code></td>
</tr>
<tr>
<td><code>dce_cf_find_name_by_key</code></td>
<td><code>dce_cf_get_host_name</code></td>
<td><code>dced_binding_create</code></td>
</tr>
</tbody>
</table>

**Files**

- `dcelocal/dce_cf.db`

  The machine's local DCE configuration file (where `dcelocal` is usually something like `/opt/dcelocal`).
dce_cf_find_name_by_key

Purpose
Returns a string tagged by a character string key.

Format
```c
#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 was completed successfully and, if not, the reason. Possible status codes and their meanings are:
  - `dce_cf_st_ok`: Operation successfully completed.
  - `dce_cf_e_no_mem`: No storage available.
  - `dce_cf_e_no_match`: No match for `key` in file.
  - `dce_cf_e_invalid_parameter`: Parameter not valid.

Usage

The `dce_cf_find_name_by_key` API searches a text file for the first occurrence of a string tag identical to the string passed in `key`. The tag string must be the first nonwhite space 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 include file `<dce_cf.h>`.

Caution: The storage for a returned name string is allocated by `malloc`, and must be freed by the original caller of the configuration API that called `dce_cf_find_name_by_key`.

This function is limited to processing lines of text whose length is less than 1024 characters.

Return Values

None

Related Information
dce_cf_find_name_by_key

Routines

dce_cf_binding_entry_from_host  dce_cf_get_host_name  dce_cf_prin_name_from_host
dce_cf_get_cell_name

Files
/opt/dcelocal/dce_cf.db  The local DCE configuration file of the machine.

Books

- [z/OS DCE Administration Guide](#)
**dce_cf_free_cell_aliases**

**Purpose**
Frees a list of cell name aliases for the local cell.

**Format**

```c
#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.

Possible status codes are:

- **dce_cf_st_ok**  
The configuration request was successful.

- **dce_cf_e_file_open**  
The DCE configuration routine is not able to open the file /opt/dcelocal/dce_cf.db.

- **dce_cf_e_no_mem**  
Not enough memory to perform the operation.

- **dce_cf_e_no_match**  
The object specified is not found in the DCE configuration file /opt/dcelocal/dce_cf.db.

**Usage**

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.

**Return Values**

None.

**Related Information**

**Routines**

- *dce_cf_get_cell_name*
- *dce_cf_get_host_name*
- *dce_cf_prin_name_from_host*
- *dce_cf_same_cell_name*
Purpose
Returns a list of aliases for the local cell.

Format
#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.

Possible status codes are:
dce_cf_st_ok Operation successfully completed.
dce_cf_e_file_open File open error.
dce_cf_e_no_mem No storage available.
dce_cf_e_no_match No match for hostname in DCE configuration file.

Usage
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.

Return Values
None.

Related Information
Routines
dce_cf_get_cell_name dce_cf_get_host_name dce_cf_same_cell_name
dce_cf_get_cell_name

Purpose
Returns the local cell name.

Format
#include <dce/dce_cf.h>

void dce_cf_get_cell_name(
    char **cellname,
    error_status_t *status);

Parameters
Input
None.

Input/Output
cellname A pointer to a string (char **) in which a string containing the cell name 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 was completed successfully and, if not, the reason.

Possible status codes and their meanings are:
dce_cf_st_ok Operation successfully completed.
dce_cf_e_file_open File open error.
dce_cf_e_no_mem No storage available.
dce_cf_e_no_match No match for cellname in DCE configuration file.
dce_cf_e_invalid_parameter Parameter not valid.

Usage
The dce_cf_get_cell_name API searches the DCE configuration file for the name of the local cell. If the name is found, dce_cf_get_cell_name allocates (by a call to malloc) a buffer for it, copies the name into the buffer, and returns the buffer's address in the cellname input parameter.

Caution: The storage for a returned name string is allocated by malloc, and must be freed by the caller of dce_cf_get_cell_name.

This function is limited to processing lines of text whose length is less than 1024 characters.

Return Values
None

Related Information

Chapter 8. DCE Utility Routines 8-31
**dce_cf_get_cell_name**

**Routines**

- dce_cf_binding_entry_from_host
- dce_cf_find_name_by_key
- dce_cf_free_cell_aliases_key
- dce_cf_get_host_name
- dce_cf_prin_name_from_host
- dce_cf_same_cell_name

**Files**

 `/opt/dcelocal/dce_cf.db`  
The local DCE configuration file of the machine.

**Books**

- [z/OS DCE Administration Guide](#)
dce_cf_get_csrgy_filename

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

Format
#include <dce/dce_cf.h>
void dce_cf_get_csrgy_filename(
    char **csrgy_filename,
    error_status_t *status);

Parameters
Input
None.

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.

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.

Possible status codes and their meanings are as follows:
- dce Cf_st_ok       Operation successfully completed.
- dce Cf_e_file_open File open error.
- dce Cf_e_no_mem    No memory available.

Usage
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.

Note: csrc is not supported in z/OS DCE. 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 the returned csrgy_filename string is allocated by malloc(), and must be freed by the caller of dce Cf_get_csrgy_filename().

Return Values
None.

Related Information
dce_cf_get_csrgy_filename

Routines

- dce_cf_find_name_by_key
- dce_cf_get_host_name
- dce_cf_prin_name_from_host
- dce_loc_to_rgy
- dce_rgy_to_loc
- rpc_rgy_get_codesets

Files

- dcelocal/dce_cf.db
  The machine’s local DCE configuration file (where dcelocal defaults to /opt/dcelocal).
dce_cf_get_host_name

Returns the host name relative to the local cell’s root.

Format

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

void dce_cf_get_host_name(
    char **hostname,
    error_status_t *status);
```

Parameters

Input

None.

Input/Output

`hostname` 
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 was completed successfully and, if not, the reason.

Possible status codes and their meanings are:

- `dce_cf_st_ok` 
  Operation successfully completed.
- `dce_cf_e_file_open` 
  File open error.
- `dce_cf_e_no_mem` 
  No storage available.
- `dce_cf_e_no_match` 
  No match for `hostname` in DCE configuration file.
- `dce_cf_e_invalid_parameter` 
  Parameter not valid.

Usage

The `dce_cf_get_host_name` API searches the DCE configuration file for the name of the local host 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 storage for a returned name string is allocated by malloc, and must be freed by the caller of `dce_cf_get_host_name`.

This DCE version of this function is limited to processing lines of text whose length is less than 1024 characters.

Return Values

None

Related Information

Routines

- `dce_cf_binding_entry_from_host`
- `dce_cf_get_cell_name`
- `dce_cf_prin_name_from_host`
- `dce_cf_find_name_by_key`
Files

`/opt/dcelocal/dce_cf.db`  The local DCE configuration file of the machine.

Books

- [z/OS DCE Administration Guide](#)
dce_cf_prin_name_from_host

Purpose
Returns the principal name of the host.

Format

```
#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; if it is found, it 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 was completed successfully and, if not, the reason.

Possible status codes and their meanings are:

- **dce_cf_st_ok** Operation successfully completed.
- **dce_cf_e_file_open** File open error.
- **dce_cf_e_no_mem** No storage available.
- **dce_cf_e_no_match** No match for *hostname* in DCE configuration file.
- **dce_cf_e_invalid_parameter** Parameter not valid.

Usage

The **dce_cf_prin_name_from_host** API 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**.

Return Values

None

Related Information

**Routines**

- **dce_cf_binding_entry_from_host**
- **dce_cf_find_name_by_key**
- **dce_cf_get_cell_name**
- **dce_cf_get_host_name**
dce_cf_prin_name_from_host

Files
/opt/dcelocal/dce_cf.db The machine’s local DCE configuration file machine.

Books
- z/OS DCE Administration Guide
dce_cf_profile_entry_from_host

Purpose
Returns the host profile name.

Format
#include <dce/dce_cf.h>

void dce_cf_profile_entry_from_host(
    char *hostname,
    char **prof_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; if it is found, it 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, the reason.

Possible status codes and their meanings are:

dce_cf_st_ok  Operation successfully completed.
dce_cf_e_file_open  File open error.
dce_cf_e_no_mem  No storage available.
dce_cf_e_no_match  No match for hostname in DCE configuration file.
dce_cf_e_invalid_parameter  Parameter not valid.

Usage
The dce_cf_profile_entry_from_host API returns the profile name associated with the hostname passed to it. If hostname is NULL, dce_cf_profile_entry_from_host returns the profile name associated with the name returned by dce_cf_get_host_name.

Note: Cell names are case insensitive. Cell names returned by DCE APIs default to lowercase. If you are doing string comparisons on cell names returned by DCE APIs, you must convert your strings to lowercase before performing the comparisons.

The storage for a returned name string is allocated by malloc, and must be freed by the caller of dce_cf_get_cell_name.

Return Values
None

Related Information

dce_cf_profile_entry_from_host

Routines

dce_cf_binding_entry_from_host  dce_cf_get_cell_name  dce_cf_get_host_name
dce_cf_find_name_by_key

Files

/opt/dcelocal/dce_cf.db  The machine’s local DCE configuration file machine.

Books

- [z/OS DCE Administration Guide](#)
dce_cf_same_cell_name

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

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

void dce_cf_same_cell_name(
    char *cell_name1,
    char *cell_name2,
    boolean32 *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.

Possible status codes are:
- **dce_cf_st_ok**: Operation successfully completed.
- **dce_cf_e_no_match**: No match for hostname in DCE configuration file.

Usage
The \texttt{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 cell. The \texttt{result} parameter is set to \texttt{TRUE} if they do, and to \texttt{FALSE} if they do not.

If only one cell name is specified as an input parameter, the \texttt{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 \texttt{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.

Return Values
None.

Related Information
dce_cf_same_cell_name

Routines

dce_cf_free_cell_aliases       dce_cf_get_cell_aliases       dce_cf_get_cell_name
The DCE Backing Store Interface Routines

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 stand-alone programs that do not involve remote procedure calls (RPC). 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 "Data Structures," below.

Encoding and Decoding in the Backing Store: When a Remote Procedure Call (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 z/OS DCE [Application Development Guide: Core Components](https://www.ibm.com/docs/en/zos). 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 (ES) 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, because 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 file name. 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.
\textbf{dce\_db\_store\_by\_uuid()}

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

\textbf{dce\_db\_unlock()}

Unlocks a backing store.

\textbf{Data Types and Structures:}

\textit{dce\_db\_handle\_t}

An opaque handle to a backing store. Use \textbf{dce\_db\_open()} acquire the handle.

\textit{dce\_db\_header\_t}

The data structure that defines a standard backing store header for data items. Use \textbf{dce\_db\_header\_fetch()} to retrieve it from a backing store and \textbf{dce\_db\_std\_header\_init()} to initialize it.

\textit{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 \textbf{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.

\textbf{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.

\textbf{Files:} database.idl
database.h
db.h
dbif.h
**dce_db_close**

**Purpose**
Closes an open backing store.

**Format**
```
#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.

**Usage**

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.

**Related Information**

Routines:  
`dce_db_open`
dce_db_delete

Purpose
Deletes an item from a backing store.

Format
#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 data type 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. Possible error codes and their meanings are:

- 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.

Usage
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.

Related Information
Routines
dce_db_delete_by_name        dce_db_delete_by_uuid        dce_db_open
**dce_db_delete_by_name**

**Purpose**
Deletes an item from a string-indexed backing store.

**Format**
```c
#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. Possible error codes and their meanings are:
  - `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.

**Usage**
The `dce_db_delete_by_name()` 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 name, as selected by the `db_c_index_by_name` bit in the `flags` parameter to `dce_db_open()` when the backing store was created.

**Related Information**

**Routines**
- `dce_db_delete_by_uuid`
- `dce_db_iter_start`
- `dce_db_open`
dce_db_delete_by_uuid

Purpose
Deletes an item from a UUID-indexed backing store.

Format
#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. Possible error codes and their meanings are:

- 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.

Usage
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.

Related Information

Routines
dce_db_delete_by_name         dce_db_iter_start         dce_db_open
dce_db_fetch

Purpose
Retrieves data from a backing store.

Format
#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 data type 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.

     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.

Usage
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.

Related Information
Routines

dce_db_fetch_by_name   dce_db_fetch_by_uuid

dce_db_free            dce_db_open
**dce_db_fetch_by_name**

**Purpose**
Retrieves data from a string-indexed backing store.

**Format**
```
#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. Possible error codes and their meanings are:

  - `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.

**Usage**

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);

Related Information

Routines

dce_db_fetch  dce_db_free  dce_db_open

dce_db_fetch_by_uuid
**dce_db_fetch_by_uuid**

**Purpose**
Retrieves data from a UUID-indexed backing store.

**Format**
```c
#include <dce/dce.h>
#include <dce/dbif.h>

void dce_db_fetch_by_uuid(
    dce_db_handle_t handle,
    uuid_t CcNHIX key,
    void CcNHIX data,
    error_status_t CcNHIX 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. Possible error codes and their meanings are:
  - `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.

**Usage**
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 stand-alone 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);

Related Information

Routines

- dce_db_fetch
- dce_db_fetch_by_name
- dce_db_free
- dce_db_open
Purpose
Releases the data supplied from a backing store.

Format
#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.

Usage
The dce_db_free() routine is designed to free the data area previously returned via a call to 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, the use of dce_db_free() is recommended.

Related Information
Routines
dce_db_fetch dce_db_fetch_by_name dce_db_fetch_by_uuid
**dce_db_header_fetch**

**Purpose**
Retrieves the standard header from a backing store object.

**Format**
```
#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. Possible error codes and their meanings are:
  - `db_s_key_not_found`
    The key was not found in the backing store.

**Usage**
The `dce_db_header_fetch()` routine returns a pointer to a copy of the standard 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.

**Related Information**

**Routines**
- `dce_db_std_header_init`
dce_db_inq_count

Purpose
Returns the number of items in a backing store.

Format

```c
#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.

Usage

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.

Related Information

Routines

dce_db_iter_next
Purpose
Frees the state associated with iteration.

Format
#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.

Usage
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().

Related Information

Routines
dce_db_iter_next_by_name dce_db_iter_next_by_uuid dce_db_iter_start
**dce_db_iter_next**

**Purpose**
During iteration, returns the next key from a backing store.

**Format**
```c
#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. Possible error codes and their meanings are:
- `db_s_no_more`: All the keys in the backing store have been accessed; there are no more iterations remaining to be done.

**Usage**

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.

**Related Information**

**Routines**

- `dce_db_fetch_by_name`
- `dce_db_fetch_by_uuid`
- `dce_db_iter_done`
- `dce_db_iter_next_by_name`
- `dce_db_iter_next_by_uuid`
- `dce_db_iter_start`
dce_db_iter_next_by_name

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

Format
```c
#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. Possible error codes and their meanings are:

- db_s_no_more
  All the keys in the backing store have been accessed; there are no more iterations remaining to be done.
- db_s_bad_index_type
  The backing store identified by the handle parameter is not indexed by name.

Usage
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.

Related Information

Routines
dce_db_iter_done  dce_db_iter_next_by_uuid  dce_db_iter_start
dce_db_iter_next
**dce_db_iter_next_by_uuid**

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

**Format**
```c
#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. Possible error codes and their meanings are:
  - `db_s_no_more` All the keys in the backing store have been accessed; there are no more iterations remaining to be done.
  - `db_s_bad_index_type` The backing store identified by the `handle` parameter is not indexed by name.

**Usage**
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.

**Related Information**

**Routines**
- `dce_db_iter_next`
- `dce_db_iter_next_by_name`
- `dce_db_iter_start`
dce_db_iter_start

Purpose
 Prepares a backing store for iteration.

Format
#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. Otherwise, it returns an error. Possible error codes and their meanings are:

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

Usage
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().

Related Information

Routines
dce_db_iter_next dce_db_iter_next_by_uuid dce_db_unlock
dce_db_iter_next_by_name dce_db_lock dce_db_open
dce_db_lock

Purpose
Applies an advisory lock on a backing store.

Format
#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. Possible error codes and their meanings are:

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

Usage
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.

Related Information

Routines
dce_db_delete_by_name  dce_db_store  dce_db_store_by_uuid
dce_db_delete_by_uuid  dce_db_store_by_name  dce_db_unlock
**dce_db_open**

**Purpose**
Opens an existing backing store or creates a new one.

**Format**
```c
#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 file name 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. Possible error codes and their meanings are:
  - **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.
dce_db_open

**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 file name is specified.

**db_s_open_failed_eaccess**
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.

**Usage**

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:

```idl
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:

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

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

```c
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);
```

**Related Information**

8-66 Application Development Reference Volumes 1 and 2
Routines:  dce_db_close
dce_db_std_header_init

Purpose
Initializes a standard backing store header.

Format
#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.

Related Information

Usage
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.

Related Information

Routines
dce_db_header_fetch
dce_db_store

Purpose
Stores data into a backing store.

Format
#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 data type 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.

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.

Usage
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 stand-alone 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.

Related Information
dce_db_store

Routines

dce_db_open  dce_db_store_by_name  dce_db_store_by_uuid
Purpose
Stores data into a string-indexed backing store.

Format
```c
#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.

  - `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.

Usage

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_smAllocate()`, 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 stand-alone 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.

Related Information
dce_db_store_by_name

Routines

dce_db_store  dce_db_store_by_uuid
dce_db_store_by_uuid

Purpose
Stores data into a UUID-indexed backing store.

Format
#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.

  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.

Usage
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.

Related Information
<table>
<thead>
<tr>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_db_store_by_uuid</td>
</tr>
</tbody>
</table>

Routines

<table>
<thead>
<tr>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_db_store</td>
</tr>
<tr>
<td>dce_db_store_by_name</td>
</tr>
</tbody>
</table>
dce_db_unlock

Purpose
Releases the backing store lock.

Format
#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.

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

Related Information
Routines: dce_db_lock
The DCE Messaging Interface

The only DCE Messaging interface routine supported in z/OS DCE is `dce_error_inq_text()`, which retrieves from the installed DCE component message catalogs the message text associated with an error status code returned by a DCE library routine.
dce_error_inq_text

Purpose
Returns the message text for a DCE component status code. Used by client, server, or management applications. Any DCE component can call this routine.

Format
```c
#include <dce/rpc.h>
#include <dce/dce_error.h>

void dce_error_inq_text(
    unsigned32 status_to_convert ,
    unsigned char *error_text ,
    int *status );
```

Parameters

Input

status_to_convert Specifies the DCE status code to convert to a text string.

Output

error_text Pointer to a buffer at least dce_c_error_string_len bytes long. Returns a NULL-terminated character string message corresponding to the status_to_convert parameter.

status Returns the status from this routine. This status indicates whether the routine completed successfully and, if not, the reason.

Possible status and their meanings are:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

Usage

The dce_error_inq_text API returns a NULL-terminated character-string message for the DCE status code specified.

The application must provide storage for the returned message. The buffer error_text must be at least dce_c_error_string_len bytes long. This constant is defined in <dce/dce_error.h>

If the call fails, a default message indicating only the status number is returned in the buffer error_text.

The current setting of LANG and NLSPATH influence where this routine searches for the message catalog that contains the input status code’s message text. See the Z/OS DCE Administration Guide for a complete list and description of these environment variables.

Return Values
None.

Related Information
None.
The DCE Server Routines

The DCE server routines 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 server routines are:

- **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* server'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* server'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* 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* 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 the introduction to the *dced_* routines for a complete description of **server_t**.

Files:  
* dce/dced.h  
* dce/dced_base.idl
dce_server_disable_service

Purpose
Disables an individual service of a server.

Format
#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.

Usage
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.

Related Information

Routines
dce_server_enable_service
dce_server_register
dce_server_enable_service

Purpose
Enables an individual service for a server.

Format
#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.

Usage
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 calling 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 requires 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.

Related Information

Routines
dce_server_disable_service dce_server_register
**dce_server_inq_attr**

**Purpose**
Oberains from **dced** the value of an attribute known to the server.

**Format**
```c
#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. On successful completion, the call returns `error_status_ok`. Otherwise, it returns an error. Possible status codes are:
  - `dced_s_server_attr_not_found` In the **dce_server_inq_attr** routine, the requested attribute was not found in the list of attributes in the `server_t` structure. This may be for information only, depending on whether or not a match was expected.
  - `dced_s_not_started_by_dced` The server was not started by the DCE Host daemon. The DCE Host daemon sets several environment variables that are available to servers started by it. One of those variables was not found.

**Usage**
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.

**Related Information**

**Routines**
- `dce_server_inq_uuids`
- `dce_server_inq_server`
**dce_server_inq_server**

**Purpose**
Obtains the server configuration data `dced` used to start the server.

**Format**
```c
#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. On successful completion, the call returns `error_status_ok`. Otherwise, it returns an error. Possible status codes are:
  - `dced_s_not_started_by_dced`: The server was not started by the DCE Host daemon. The DCE Host daemon sets several environment variables that are available to servers started by it. One of those variables was not found.
  - `dced_s_data_unavailable`: Cannot obtain dced server config data.

**Usage**
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()`.

**Related Information**
**Routines**
- `dce_server_register`
dce_server_inq_uuids

Purpose
Obtains the UUIDs that dced associates with the server's configuration and execution data.

Format
#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. On successful completion, the call returns error_status_ok. Otherwise, it returns an error. Possible status codes are:

dced_s_not_started_by_dced The server was not started by the DCE Host daemon.
The DCE Host daemon sets several environment variables that are available to servers started by it. One of those variables was not found.

Usage
The dce_server_inq_uuids() routine obtains the UUIDs that dced uses in its srvrconf and srvreexc 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.

Related Information

Routines
dce_server_inq_server
dce_server_register

Purpose
Registers a server with DCE.

Format
#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 the introduction to the sec_routines.

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:
    • 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. On successful completion, the call returns
error_status_ok. Otherwise, it returns an error. Possible status codes are:
    rpc_s_no_memory Not enough memory to perform the operation.

Usage
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 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.

**Related Information**

**Routines**

- `dce_server_inq_server`
- `rpc_server_listen`
- `dce_server_sec_begin`
- `dce_server_unregister`
dce_server_sec_begin

Purpose
Establishes a server to receive fully authenticated remote procedure calls (RPCs) and to also act as a client to do authenticated RPCs.

Format
#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. On successful completion, the call returns error_status_ok. Otherwise, it returns an error. Possible status codes are:
  dced_s_need_one_server_prin
    Exactly one server principal is required to be in the principal list in the server_t structure that is accessed by the dce_server_sec_begin routine. Either none or more than one was found.
  dced_s_not_started_by_dced
    The server was not started by the DCE Host daemon. The DCE Host daemon sets several environment variables that are available to servers started by it. One of those variables was not found.
  dced_s_no_server_keyfile
    Server keytab file name not found.
  dced_s_cannot_create_key_mgmt_thr
    Cannot create key management thread.
  dced_s_cannot_detach_key_mgmt_thr
    Cannot detach key management thread.

Usage
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.

Related Information
Routines
dce_server_register
rpc_server_listen
dce_server_sec_done
**dce_server_sec_done**

**Purpose**
Releases resources established for a server to receive (and when acting as a client, to send) fully authenticated remote procedure calls (RPCs).

**Format**
```
#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`.

**Usage**

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.

**Related Information**

**Routines**
```
dce_server_sec_begin rpc_server_listen
```
**dce_server_unregister**

### Purpose
Unregisters a DCE server.

### Format
```
#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`.

### Usage
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.

### Related Information
**Routines**
- `dce_server_register`
- `rpc_server_listen`
- `dce_server_disable_service`
**Purpose**
Tells DCE to use the specified protocol sequence for receiving remote procedure calls.
Used by server applications.

**Format**
```c
#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 introduction to this section.)

- **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`.

**Usage**
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.

**Related Information**

**Routines**
- `dce_server_register`  
- `rpc_server_use_protseq`
The DCE Host Daemon Routines

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.

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 above 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 dced.h 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 introductory information for the sec_APIs. 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 identifier 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 introductory information for the sec_APIs. sec_intro.

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 introductory information for the sec_ APIs.

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.
    
    Note: This function is not currently supported in z/OS DCE.
  - 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 unsuccessful exit status.

Several bits are also reserved for vendor-specific startup and include server_c_startup_vendor1, server_c_startup_vendor2, server_c_startup_vendor3, and 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.

In z/OS DCE, if the server is a shell script, the first two characters of the file must be #!. They should be entered using the same code page that DCEKERN will use at the time the server is executed.

The shell (/bin/sh) is invoked to execute the script; stdin, stdout, and stderr will not be open and the server program must open them as necessary (for example, use redirection within a shell script to route output to an HFS file.

- arguments: This is a list of arguments for the server and is of type dced_string_list_t.
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 relevant servers.

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

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

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

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`.

In z/OS UNIX, if no directory is specified, the default directory will be `/tmp`.

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:
    ```c
    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`.

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 introductory information for the `sec_` APIs.
- **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 and include service_c_vendor1, service_c_vendor2, service_c_vendor3, and 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.

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

Note: This function is not currently supported when targeted at DCE servers running on MVS hosts.
	srverexec_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.
	srverexec_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.
	srverexec_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

8-96 Application Development Reference Volumes 1 and 2
dced_binding_create

Purpose
Establishes a dced binding to one of the host services of a remote (or the local) dced.

Format
```
#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. Preexisting 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.
  - `/.*cell/host_name/config/service_name` - A complete specification for `service_name@host`, where the host is anywhere in the local namespace.

- **binding_flags**: Currently, the only valid flag value 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. Possible status codes are:
  - `error_status_ok` - Operation completed successfully.
  - `dce Cf_e_no_mem` - There is not enough storage to perform the operation.
  - `dced_s_invalid_arg` - One of the routine arguments is invalid or out of range.
  - `dced_s_no_memory` - There is not enough storage to perform the operation.
  - `dced_s_unknown_service` - The DCE Host Daemon service you requested is not supported by this release.
  - `rpc_s_entry_not_found` - Inquiry context is not valid.
  - `rpc_s_incomplete_name` - Incomplete object or directory name supplied to an NSI operation.
Usage

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 (srvreexec)
- 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

This 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);

Related Information

Routines:

dced_binding_free           dced_binding_from_rpc_binding
**dced_binding_free**

**Purpose**
Releases the resources associated with a dced binding handle.

**Format**
```
#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. Possible status codes are:
  - `error_status_ok`: Operation completed successfully.
  - `rpc_s_invalid_binding`: Binding handle is not correct.
  - `rpc_s_wrong_kind_of_binding`: Type of binding handle (client or server) is not correct for the operation.

**Usage**
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.

**Related Information**

**Routines**
- `dced_binding_create`
- `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.

Format
#include <dce/dced.h>

void dced_binding_from_rpc_binding(
    dced_service_type_t service,
    rpc_binding_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.

Possible status codes are:

- error_status_ok: Operation completed successfully.
- dced_s_no_memory: There is not enough storage to perform the operation.
- dced_s_unknown_service: The DCE Host Daemon service you requested is not supported by this release.
- ept_s_cant_perform_op: Endpoint mapper cannot perform operation.
- ept_s_database_invalid: Endpoint database format is not in the correct form for the endpoint mapper.
- ept_s_invalid_context: Endpoint map or search context is not valid.
- ept_s_invalid_entry: Endpoint entry contains invalid data.
- rpc_s_comm_failure: RPC is not communicating with the remote server.
- rpc_s_fault_context_mismatch: Client and server contexts for a remote call do not match.
- rpc_s_invalid_arg: One of the routine arguments is invalid or out of range.
- rpc_s_invalid_binding: Binding handle is not correct.
- rpc_s_no_more_elements: No more profile elements are found.
- rpc_s_wrong_kind_of_binding: Type of binding handle (client or server) is not correct for the operation.
Usage

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 rpc_binding_from_string_binding, rpc_ns_binding_import_next, or rpc_ns_binding_lookup_next.

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.

```c
rpc_binding_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);
```

Related Information

Routines:

- dced_binding_free
- rpc_ns_binding_import_next
- rpc_binding_from_string_binding
- dced_binding_create
- rpc_ns_binding_lookup_next
- rpc_binding_copy
dced_binding_set_auth_info

Purpose
Sets authentication and authorization information for a dced binding handle.

Format
```
#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. Possible status codes are:
  - **error_status_ok**: Operation completed successfully.
  - **dced_s_bad_binding**: The dced binding handle supplied did not match the service requested.
  - **dced_s_no_support**: The requested function is not supported by this release.
  - **ept_s_not_registered**: Entry is not registered in the endpoint map.
  - **rpc_s_authn_authz_mismatch**: Authentication does not support the call that the authorization service is making.
  - **rpc_s_binding_incomplete**: Binding does not contain endpoint and object UUID.
  - **rpc_s_comm_failure**: RPC is not communicating with the remote server.
  - **rpc_s_invalid_binding**: Binding handle is not correct.
  - **rpc_s_mgmt_op_disallowed**: Management operation is not allowed.
  - **rpc_s_rpcd_comm_failure**: RPC run time cannot communicate with RPC daemon.
  - **rpc_s_unknown_authn_service**: A client specified an authentication service that is not supported by the RPC runtime.
  - **rpc_s_unsupported_protect_level**: Requested protection level is not supported.
Usage

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.

```c
struct 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);

// Other routines including dced API routines. */
```

Related Information

Routines:

- `rpc_binding_set_auth_info`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
dced_entry_add

Purpose
Adds a keytab or hostdata entry to a host's dced for an existing file on that host.

Format
#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.
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. Possible status codes are:
    error_status_ok Operation completed successfully.
    db_s_readonly Database is opened read-only.
    db_s_store_failed Database store failure.
    dced_s_already_exists An object already exists with the same UUID or name as the object you are creating.
    dced_s_bad_binding The dced binding handle supplied did not match the service requested.
    dced_s_import_cant_access An existing file cannot be found to import.
    dced_s_no_support The requested function is not supported by this release.
    rpc_s_binding_has_no_auth Call is not authenticated.
    sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
    uuid_s_no_address RPC run time cannot obtain IEEE 802 hardware address.

Usage
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.

```c

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/NEWERprinter");

dced_entry_add(dced_bh, &entry, &status);
```

Related Information

Routines:

- `dced_entry_remove`
- `dced_keytab_create`
- `dced_hostdata_create`
- `dced_binding_from_rpc_binding`
- `dced_binding_create`
**dced_entry_get_next**

**Purpose**
Obtains one data entry from a list of entries of a dced service.

**Format**

```c
#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.

**Possible status codes are:**

- `error_status_ok`
  Operation completed successfully.

- `dced_s_no_more_entries`
  All available entries were exhausted.

**Usage**

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 `svrconf` and `svrexec` 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 exhausted.

```c
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);
```
Related Information

Routines

dced_initialize_cursor  dced_binding_create  dced_binding_from_rpc_binding

dced_release_cursor
Purpose
Removes a hostdata or keytab data entry from a dced service's list of entries.

Format
#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.

    error_status_ok Operation completed successfully.
    db_s_del_failed Database delete operation failure.
    db_s_key_not_found Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
    db_s_readonly Database is opened read-only.
    dced_s_bad_binding The dced binding handle supplied did not match the service requested.
    dced_s_no_support The requested function is not supported by this release.
    dced_s_not_found A matching object cannot be found. (May be informational only if a match was not expected.)
    sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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 srvrexe 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.

Related Information

Routines

- `dced_hostdata_delete`
- `dced_keytab_delete`
- `dced_server_delete`
- `dced_initialize_cursor`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
**dced_hostdata_create**

**Purpose**
Creates a hostdata item and the associated entry in dced on a specific host.

**Format**
```c
#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.

- **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.

- **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. Possible status codes are:
    - **error_status_ok**
      Operation completed successfully.
    - **db_s_key_not_found**
      Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
    - **db_s_readonly**
      Database is opened read-only.
    - **db_s_store_failed**
      Database store failure.
    - **dced_s_already_exists**
      An object already exists with the same UUID or name as the object you are creating.
    - **dced_s_bad_binding**
      The dced binding handle supplied did not match the service requested.
    - **dced_s_cant_open_storage_file**
      Failed to open a file associated with the dced object.
    - **dced_s_import_already_exists**
      An existing file was found when creating a dced object with no-import specified. A file cannot exist when no-import is specified.
    - **dced_s_unknown_attr_type**
      Specified dced attribute type is incorrect.
    - **sec_acl_invalid_permission**
      Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
Usage

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.

```c

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/NEWERprinter");

    /* 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);
```

Related Information

**Routines**

- **dced_entry_add**
- **dced_binding_create**
- **dced_binding_from_rpc_binding**
- **dced_hostdata_read**
dced_hostdata_delete

Purpose
Deletes a hostdata item from a specific host and removes the associated entry from dced.

Format
#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. Possible status codes are:
    error_status_ok
        Operation completed successfully.
    db_s_bad_index_type
        Index type is not known or incorrectly specified.
    db_s_del_failed
        Database delete operation failure.
    db_s_iter_not_allowed
        Operation is not allowed while iterating.
    db_s_key_not_found
        Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
    dced_s_bad_binding
        The dced binding handle supplied did not match the service requested.
    dced_s_cant_remove_storage_file
        Failed to remove a file referred to by the dced object.
    dced_s_not_found
        A matching object cannot be found. (May be informational only if a match was not expected.)
    sec_acl_invalid_permission
        Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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.
Caution

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.

Related Information

Routines

dced_entry_remove  dced_binding_create  dced_binding_from_rpc_binding
dced_hostdata_read
dced_hostdata_read

 Purpose
 Reads a hostdata item maintained by dced on a specific host.

 Format
 #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 Specifications 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 introductory information for the sec_ APIs 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.

 Possible status codes are:

 error_status_ok Operation completed successfully.
 db_s_bad_index_type Specified key was not found in the database. This
 means the search failed. It is not necessarily a problem.
 db_s_key_not_found Index type is not known or incorrectly specified.

dce_cf_e_file_open Unable to open DCE configuration file.

dce_cf_e_no_match No match for object object in DCE configuration file.

dce_cf_e_no_mem There is not enough storage to perform the operation.

dced_s_bad_binding The dced binding handle supplied did not match the service requested.

dced_s_cant_open_storage_file Failed to open a file associated with the dced object.

dced_s_invalid_attr_type Specified attribute type was incorrect.

dced_s_no_memory There is not enough storage to perform the operation.

sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

uuid_s_bad_version UUID version unknown to the RPC runtime.
**Usage**

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.

**Related Information**

**Routines**

- `dced_objects_release`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `dced_object_read`
**dced_hostdata_write**

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

**Format**
```c
#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. Possible status codes are:
  - **error_status_ok**
    Operation completed successfully.
  - **db_s_bad_index_type**
    Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
  - **dced_s_bad_binding**
    The dced binding handle supplied did not match the service requested.
  - **dced_s_cant_open_storage_file**
    Failed to open a file associated with the dced object.
  - **dced_s_no_postprocessors**
    The postprocessor item in the **hostdata** database was not defined. (Always occurs the first time the DCE Host Daemon is run on a host or if the **hostdata** database is explicitly deleted.)
  - **dced_s_postprocessor_file_fail**
    DCE Host Daemon cannot read the **hostdata** postprocessor list file specified, either because it does not exist or because DCE Host Daemon does not have permission to read it.
  - **dced_s_postprocessor_spawn_fail**
    Cannot spawn the program specified in the postprocessor file for the **hostdata** object. May be insufficient resources to create another process.
  - **dced_s_unknown_attr_type**
    Specified dced attribute type is incorrect.
  - **sec_acl_invalid_permission**
    Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
  - **uuid_s_bad_version**
    UUID version unknown to the RPC runtime.
Usage

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.

Related Information

Routines

- `dced_hostdata_read`
- `dced_hostdata_create`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
**dced_initialize_cursor**

**Purpose**
Sets a cursor to the start of a cached list of data entries for a dced service.

**Format**
```c
#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. Possible status codes are:
  - **error_status_ok**
    Operation completed successfully.
  - **db_s_bad_index_type**
    Index type is unknown or incorrectly specified.
  - **db_s_iter_not_allowed**
    Operation requested is not allowed while iterating.
  - **db_s_key_not_found**
    Specified key not found in database; search failed. This is not necessarily a problem.
  - **dced_s_bad_binding**
    The dced binding handle supplied did not match the service requested.
  - **dced_s_no_memory**
    There is not enough storage to perform the operation.
  - **dced_s_no_support**
    The requested function is not supported by this release.
  - **sec_acl_invalid_permission**
    Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

**Usage**

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 the list is obtained with one remote procedure call. However, your application is scalable if you use the 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 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.
Related Information

Routines

dced_entry_get_next          dced_list_get          dced_binding_from_rpc_binding

dced_release_cursor          dced_binding_create
dced_inq_id

Purpose
Obtains the UUID that a DCE Host daemon dced associates with a name.

Format
#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. Possible status codes are:
error_status_ok Operation completed successfully.
db_s_bad_index_type Index type is not known or incorrectly specified.
db_s_iter_not_allowed Operation is not allowed while iterating.
db_s_key_not_found Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
dced_s_not_found A matching object cannot be found. (May be informational only if a match was not expected.)
sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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_inq_id

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);

Related Information

Routines

dced_inq_name dced_binding_create dced_binding_from_rpc_binding
Purpose
Obtains the entry name that dced associates with a UUID.

Format
#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.

Possible status codes are:

- **error_status_ok**: Operation completed successfully.
- **db_s_bad_index_type**: Index type is not known or incorrectly specified.
- **db_s_iter_not_allowed**: Operation is not allowed while iterating.
- **db_s_key_not_found**: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- **dced_s_not_found**: A matching object cannot be found. (May be informational only if a match was not expected.)
- **sec_acl_invalid_permission**: Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
- **uuid_s_bad_version**: UUID version unknown to the RPC runtime.

Usage

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);
...

Related Information

Routines

dced_inq_id  dced_binding_create  dced_binding_from_rpc_binding
dced_keytab_add_key

Purpose
Adds a key (server password) to a specified key table on a specific host.

Format

```c
#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.

**Output**

- **status**
  Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. Possible status codes are:

  - **error_status_ok**
    Operation completed successfully.

  - **db_s_bad_index_type**
    Index type is not known or incorrectly specified.

  - **db_s_key_not_found**
    Specified key was not found in the database. This means the search failed. It is not necessarily a problem.

  - **dced_s_bad_binding**
    The dced binding handle supplied did not match the service requested.

  - **dced_s_key_v0_not_allowed**
    Adding a version zero key is not allowed.

  - **dced_s_key_version_mismatch**
    Specified version in the dced_key_t structure and its enclosed sec_passwd_rec_t do not match.

  - **dced_s_need_privacy**
    Permission denied without proper protection level. rpc_c_protect_level_pkt_privacy required for this operation when DCE Host Daemon was started with DCE privacy encryption.

  - **dced_s_random_key_not_allowed**
    Request for a random key in this operation not allowed.

  - **rpc_s_binding_has_no_auth**
    Call is not authenticated.

  - **rpc_s_invalid_binding**
    Binding handle is not correct.

  - **rpc_s_wrong_kind_of_binding**
    Type of binding handle (client or server) is not correct for the operation.

  - **sec_acl_invalid_permission**
    Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

  - **sec_key_mgmt_e_authn_invalid**
    Invalid authentication service passed to the Security client run time. The only permitted value is rpc_c_authn_dce_secret.
**Usage**

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` 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.

**Related Information**

**Routines**

- `dced_keytab_change_key`
- `sec_key_mgmt_set_key`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `dced_keytab_add_key`
dced_keytab_change_key

Purpose
Changes a key (server password) in both a key table and in the security registry.

Format
#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.

Possible status codes are:

error_status_ok
Operation completed successfully.

db_s_bad_index_type
Index type is not known or incorrectly specified.

db_s_key_not_found
Specified key was not found in the database. This means the search failed. It is not necessarily a problem.

dced_s_bad_binding
The dced binding handle supplied did not match the service requested.

dced_s_key_version_mismatch
Specified version in the dced_key_t structure and its enclosed sec_passwd_rec_t do not match.

dced_s_need_privacy
Permission denied without proper protection level.

rpc_s_binding_has_no_auth
Call is not authenticated.

rpc_s_incorrect_binding
Binding handle is not correct.

rpc_s_wrong_kind_of_binding
Type of binding handle (client or server) is not correct for the operation.

sec_acl_invalid_permission
Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

sec_key_mgmt_e_authn_invalid
Invalid authentication service passed to the Security client run time. The only permitted value is rpc_c_authn_dce_secret.

sec_key_mgmt_e_key_unavailable
Requested key not found.

sec_key_mgmt_e_key_unsupported
Key type requested is not supported.
**Usage**

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 if the server calls `sec_key_mgmt_garbage_collect()` to purge its old keys explicitly, or `sec_key_mgmt_manage_key()` to purge them implicitly.

When more than one server shares the same principal identity, they use the same key. If you need to change the same key in more than one key table, use `dced_keytab_change_key()` for one change and then use the `dced_keytab_add_key()` routine for all others.

**Related Information**

**Routines**

- `dced_keytab_add_key`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `sec_key_mgmt_change_key`
dced_keytab_create

Purpose
Creates a key table with a list of keys (server passwords) in a new file on a specific host.

Format
#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. Possible status codes are:

- **error_status_ok** Operation completed successfully.
- **db_s_bad_header_type** Header type is incorrectly specified.
- **db_s_bad_index_type** Index type is not known or incorrectly specified.
- **db_s_iter_not_allowed** Operation is not allowed while iterating.
- **db_s_key_not_found** Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- **db_s_readonly** Database is opened read-only.
- **db_s_store_failed** Database store failure.
- **dced_s_already_exists** An object already exists with the same UUID or name as the object you are creating.
- **dced_s_bad_binding** The *dced* binding handle supplied did not match the service requested.
- **dced_s_import_already_exists** An existing file was found when creating a *dced* object with no-import specified. A file cannot exist when no-import is specified.
- **dced_s_need_privacy** Permission denied without proper protection level. *rpc_c_protect_level_pkt_privacy* required for this operation when DCE Host Daemon was started with DCE privacy encryption.
- **rpc_s_binding_has_no_auth** Call is not authenticated.
- **rpc_s_invalid_binding** Binding handle is not correct.
- **rpc_s_wrong_kind_of_binding** Type of binding handle (client or server) is not correct for the operation.
Usage

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.

Related Information

Routines

- `sec_key_mgmt_set_key`
- `dced_binding_from_rpc_binding`
- `dced_binding_create`

Entries

- `sec_acl_invalid_permission`: Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
- `sec_key_mgmt_e_authn_invalid`: Invalid authentication service passed to the Security client run time. The only permitted value is `rpc_c_authn_dce_secret`.
- `sec_key_mgmt_e_key_unavailable`: Requested key not found.
- `sec_key_mgmt_e_key_unsupported`: Key type requested is not supported.
- `sec_key_mgmt_e_key_version_exists`: Key version already exists in local key store.
- `sec_key_mgmt_e_unauthorized`: User is not authorized to list, add, change or delete key.
- `uuid_s_bad_version`: UUID version unknown to the RPC runtime.
dced_keytab_delete

Purpose
Deletes a key table file from a specific host.

Format
#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.

Possible status codes are:

error_status_ok
Operation completed successfully.

db_s_bad_index_type
Index type is not known or incorrectly specified.

db_s_del_failed
Database delete operation failure.

db_s_iter_not_allowed
Operation is not allowed while iterating.

db_s_key_not_found
Specified key was not found in the database. This means the search failed. It is not necessarily a problem.

dced_s_bad_binding
The dced binding handle supplied did not match the service requested.

dced_s_cant_remove_storage_file
Failed to remove a file referred to by the dced object.

dced_s_need_privacy
Permission denied without proper protection level.

rpc_c_protect_level_pkt_privacy
required for this operation when DCE Host Daemon was started with DCE privacy encryption.

rpc_s_binding_has_no_auth
Call is not authenticated.

rpc_s_invalid_binding
Binding handle is not correct.

rpc_s_wrong_kind_of_binding
Type of binding handle (client or server) is not correct for the operation.

sec_acl_invalid_permission
Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
Usage

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 want to only make the key table inaccessible remotely (via dced) but not 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.

Related Information

Routines

- `dced_keytab_remove_key`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `dced_entry_remove`
**Purpose**
Returns a key from a cached list, and advances the cursor in the list.

**Format**

```c
#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. Possible status codes are:

  - `error_status_ok`: Operation completed successfully.
  - `dced_s_no_more_entries`: No more dced entries available.

**Usage**

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 established 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.

**Related Information**

**Routines**

- `dced_keytab_initialize_cursor`
- `dced_keytab_release_cursor`
- `sec_key_mgmt_get_next_key`
dced_keytab_initialize_cursor

Purpose
Obtains a list of keys from a key table and sets a cursor at the beginning of the list.

Format
#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.

Possible status codes are:

- error_status_ok: Operation completed successfully.
- dced_s_bad_binding: The dced binding handle supplied did not match the service requested.
- dced_s_need_privacy: Permission denied without proper protection level.
- rpc_c_protect_level_pkt_privacy: Required for this operation when DCE Host Daemon was started with DCE privacy encryption.
- dced_s_no_memory: There is not enough storage to perform the operation.
- dced_s_no_support: The requested function is not supported by this release.
- sec_acl_invalid_permission: Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
- sec_key_mgmt_e_authn_invalid: Invalid authentication service passed to the Security client run time. The only permitted value is rpc_c_authn_dce_secret.
- sec_key_mgmt_e_unauthorized: User is not authorized to list, add, change or delete key.

Usage

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.
**dced_keytab_initialize_cursor**

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.

**Related Information**

**Routines**

- `dced_keytab_get_next_key`
- `dced_keytab_release_cursor`
- `sec_key_mgmt_initialize_cursor`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
**dced_keytab_release_cursor**

**Purpose**
Releases the resources of a cursor that traverses a key table's list of keys (server passwords).

**Format**
```
#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. Possible status codes are:
    - `error_status_ok` Operation completed successfully.
    - `dced_s_bad_binding` The dced binding handle supplied did not match the service requested.
    - `dced_s_no_support` The requested function is not supported by this release.

**Usage**
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.

**Related Information**

- **Routines**
  - `dced_keytab_initialize_cursor`
  - `dced_keytab_get_next_key`
**Purpose**
Removes a key (server password) from a specified key table on a specific host.

**Format**
```c
#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.

**Possible status codes are:**
- `error_status_ok`: Operation completed successfully.
- `db_s_bad_index_type`: Index type is not known or incorrectly specified.
- `db_s_key_not_found`: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- `dced_s_bad_binding`: The dced binding handle supplied did not match the service requested.
- `dced_s_need_privacy`: Permission denied without proper protection level.
- `rpc_c_protect_level_pkt_privacy`: required for this operation when DCE Host Daemon was started with DCE privacy encryption.
- `rpc_s_binding_has_no_auth`: Call is not authenticated.
- `rpc_s_invalid_binding`: Binding handle is not correct.
- `rpc_s_wrong_kind_of_binding`: Type of binding handle (client or server) is not correct for the operation.
- `sec_acl_invalid_permission`: Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.
- `sec_key_mgmt_e_authn_invalid`: Invalid authentication service passed to the Security client run time. The only permitted value is `rpc_c_authn_dce_secret`.
- `sec_key_mgmt_e_key_unavailable`: Requested key not found.
- `sec_key_mgmt_e_unauthorized`: User is not authorized to list, add, change or delete key.
Usage

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.

Related Information

Routines

<table>
<thead>
<tr>
<th>sec_key_mgmt_delete_key</th>
<th>dced_binding_create</th>
<th>dced_binding_from_rpc_binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>dced_keytab_delete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
dced_list_get

Purpose
Returns the list of data entries maintained by a dced service on a specific host.

Format
#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. Possible status codes are:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_status_ok</td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td>dced_s_bad_binding</td>
<td>The dced binding handle supplied did not match the service requested.</td>
</tr>
<tr>
<td>dced_s_no_memory</td>
<td>There is not enough storage to perform the operation.</td>
</tr>
<tr>
<td>dced_s_no_support</td>
<td>The requested function is not supported by this release.</td>
</tr>
<tr>
<td>sec_acl_invalid_permission</td>
<td>Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.</td>
</tr>
</tbody>
</table>

Usage
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 file name 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 it guarantees 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() set of 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.

8-138 Application Development Reference Volumes 1 and 2
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);

Related Information

Routines

<table>
<thead>
<tr>
<th>dced_list_release</th>
<th>dced_binding_create</th>
<th>keytabdced_key_list_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>dced_initialize_cursor</td>
<td>dced_binding_from_rpc_binding</td>
<td></td>
</tr>
</tbody>
</table>
dced_list_release

dced_list_release

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

Format
#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.

Input/Output
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. The only possible status code is error_status_ok.

Usage
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.

Related Information

Routines
dced_list_get               dced_binding_create               dced_binding_from_rpc_binding

Books:  z/OS DCE Application Development Guide: Core Components
dced_object_read

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

Format

```c
#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:
  - `hostdata`: sec_attr_t
  - `srvrcconf`: 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. Possible status codes are:
  - `error_status_ok`: Operation completed successfully.
  - `db_s_bad_index_type`: Index type is not known or incorrectly specified.
  - `db_s_key_not_found`: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
  - `dce_cf_e_file_open`: Unable to open DCE configuration file.
  - `dce_cf_e_no_match`: No match for object object in DCE configuration file.
  - `dce_cf_e_no_mem`: There is not enough storage to perform the operation.
  - `dced_s_bad_binding`: The dced binding handle supplied did not match the service requested.
  - `dced_s_need_privacy`: Permission denied without proper protection level. rpc_c_protect_level_pkt_privacy required for this operation when DCE Host Daemon was started with DCE privacy encryption.
  - `dced_s_no_memory`: There is not enough storage to perform the operation.
  - `dced_s_no_support`: The requested function is not supported by this release.
  - `dced_s_not_found`: A matching object cannot be found. (May be informational only if a match was not expected.)
dced_object_read

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.

```c

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);
```

### Related Information

- [8-142 Application Development Reference Volumes 1 and 2](#)
**Routines**

<table>
<thead>
<tr>
<th>dced_objects_release</th>
<th>dced_keytab_initialize_cursor</th>
<th>dced_binding_create</th>
</tr>
</thead>
<tbody>
<tr>
<td>dced_objects_read_all</td>
<td>dced_initialize_cursor</td>
<td>dced_binding_from_rpc_binding</td>
</tr>
<tr>
<td>dced_hostdata_read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Purpose
Reads all the data for a service of the DCE Host daemon (dced) on specific host.

Format
#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:

<table>
<thead>
<tr>
<th>Service</th>
<th>Data Type of Array Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostdata</td>
<td>sec_attr_t</td>
</tr>
<tr>
<td>srvrconf</td>
<td>server_t</td>
</tr>
<tr>
<td>srvrexec</td>
<td>server_t</td>
</tr>
<tr>
<td>keytab</td>
<td>dced_key_list_t</td>
</tr>
</tbody>
</table>

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

- error_status_ok: Operation completed successfully.
- db_s_bad_index_type: Index type is not known or incorrectly specified.
- db_s_key_not_found: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- dce_cf_e_file_open: Unable to open DCE configuration file.
- dce_cf_e_no_match: No match for object object in DCE configuration file.
- dce_cf_e_no_mem: There is not enough storage to perform the operation.
- dced_s_bad_binding: The dced binding handle supplied did not match the service requested.
- dced_s_need_privacy: Permission denied without proper protection level.
  
  *rpc_c_protect_level_PKT_PRIVACY* required for this operation when DCE Host Daemon was started with DCE privacy encryption.
- dced_s_no_memory: There is not enough storage to perform the operation.
- dced_s_no_support: The requested function is not supported by this release.
- dced_s_not_found: A matching object cannot be found. (May be informational only if a match was not expected.)
### dced_object_read_all

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_s_binding_has_no_auth</td>
<td>Call is not authenticated.</td>
</tr>
<tr>
<td>rpc_s_invalid_binding</td>
<td>Binding handle is not correct.</td>
</tr>
<tr>
<td>rpc_s_wrong_kind_of_binding</td>
<td>Type of binding handle (client or server) is not correct for the operation.</td>
</tr>
<tr>
<td>sec_acl_invalid_permission</td>
<td>Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.</td>
</tr>
<tr>
<td>sec_key_mgmt_e_authn_invalid</td>
<td>Invalid authentication service passed to the Security client run time. The only permitted value is rpc_c_authn_dce_secret.</td>
</tr>
<tr>
<td>sec_key_mgmt_e_key_unavailable</td>
<td>Requested key not found.</td>
</tr>
<tr>
<td>sec_key_mgmt_e_unauthorized</td>
<td>User is not authorized to list, add, change or delete key.</td>
</tr>
<tr>
<td>sec_s_no_memory</td>
<td>Requested memory not available.</td>
</tr>
<tr>
<td>uuid_s_bad_version</td>
<td>UUID version unknown to the RPC runtime.</td>
</tr>
</tbody>
</table>

### Usage

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, srvrevec, 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.

```c
struct dced_binding_handle_t dced_bh;
struct 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);
```

### Related Information

**Routines**
- dced_objects_release
- dced_binding_create
- dced_binding_from_rpc_binding
- dced_object_read
dced_objects_release

Purpose
Releases the resources allocated for data read from a dced service.

Format
#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. Possible status codes are:
  error_status_ok Operation completed successfully.
  dced_s_bad_binding The dced binding handle supplied did not match the service requested.
  dced_s_no_support The requested function is not supported by this release.

Usage
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_bindings_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);
}

Related Information

Routines

<table>
<thead>
<tr>
<th>dced_object_read</th>
<th>dced_hostdata_read</th>
<th>dced_binding_from_rpc_binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>dced_object_read_all</td>
<td>dced_binding_create</td>
<td></td>
</tr>
</tbody>
</table>
**dced_release_cursor**

**Purpose**
Releases the resources of a cursor which traverses a dced service’s list of entries.

**Format**
```c
#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. The only possible status code is `error_status_ok`.
  - There is not enough storage to perform the operation.

**Usage**
The `dced_release_cursor()` routine releases the resources of a cursor initially set by the `dced_initialize_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.

**Related Information**

**Routines**
- `dced_initialize_cursor`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `dced_entry_get_next`
dced_secval_start

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

Format
#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.

Possible status codes are:

- error_status_ok
  Operation completed successfully.

- dced_s_bad_binding
  The dced binding handle supplied did not match the service requested.

- dced_s_sv_already_enabled
  Secval service already enabled.

- sec_acl_invalid_permission
  Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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:
1. Maintain a login context for the host's self identity.
2. Validate and certify to applications (usually login programs) that the DCE Security daemon (secd) is legitimate.

The secval 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.

Related Information

Routines
dced_secval_stop                     dced_binding_create                     dced_binding_from_rpc_binding
Commands: dced

The secval object of dcecp.
dced_secval_status

Purpose
Indicates whether or not a specific host's security validation service of dced is running.

Format
#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. Possible status codes are:
 error_status_ok There is not enough storage to perform the operation.
 dcenced_s_bad_binding The dced binding handle supplied did not match the service requested.

Usage
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.

Related Information
Routines
dced_secval_start
 dced_binding_create
 dced_binding_from_rpc_binding
 dced_secval_stop

Commands: dced

The secval object of dcecp
dced_secval_stop

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

Format
#include <dce/dced.h>

void dced_secval_stop(
    dced_binding_handle_t dced_bh,
    error_status_t *status);

Parameters

Input
dced_bh
 Specifications 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. Possible status codes are:

- error_status_ok
  Operation completed successfully.
- dced_s_bad_binding
  The dced binding handle supplied did not match the service requested.
- dced_s_sv_not_enabled
  Secval service is not enabled.
- sec_acl_invalid_permission
  Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage

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.

Related Information

Routines
dced_secval_start dced_binding_create dced_binding_from_rpc_binding
Commands: dced

The secval object of dcecp
dced_secval_validate

Purpose
Validates and certifies that the secd used by a specific host is legitimate.

Format
#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.

error_status_ok
Operation completed successfully.
dced_s_bad_binding
The dced binding handle supplied did not match the service requested.
ept_s_not_registered
Entry is not registered in the endpoint map.
rpc_s_comm_failure
RPC is not communicating with the remote server.
rpc_s_invalid_binding
Binding handle is not correct.
rpc_s_rpcd_comm_failure
The RPC runtime cannot communicate with the DCE daemon.
rpc_s_wrong_kind_of_binding
Type of binding handle (client or server) is not correct for the operation.
sec_login_s_no_current_context
Login context for currently established network identity does not exist.

Usage
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.

*/

Related Information

Routines
sec_login* API
dced_secval_start
dced_binding_create
dced_binding_from_rpc_binding
Commands: dced

The secval object of dcecp
dced_server_create

Purpose
Creates a DCE server's configuration data for the host's dced.

Format
#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 introductory information for the dced APIs 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. Possible status codes are:

- error_status_ok: Operation completed successfully.
- db_s_bad_header_type: Header type is incorrectly specified.
- db_s_bad_index_type: Index type is not known or incorrectly specified.
- db_s_iter_not_allowed: Operation is not allowed while iterating.
- db_s_key_not_found: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- db_s_readonly: Database is opened read-only.
- db_s_store_failed: Database store failure.
- dced_s_already_exists: An object already exists with the same UUID or name as the object you are creating.
- dced_s_bad_binding: The dced binding handle supplied did not match the service requested.
- dced_s_name_missing: Name field of object to be created is NULL or empty.
- sec_acl_invalid_permission: Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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, it 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`.

```c

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 */
uid_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);
.
.

**Related Information**

**Routines**

- `dced_object_read`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`

**dcecp Objects:** `server`
dced_server_delete

Purpose
Deletes a DCE server's configuration data from dced.

Format
#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. Possible status codes are:
  error_status_ok Operation completed successfully.
  db_s_bad_index_type Index type is not known or incorrectly specified.
  db_s_del_failed Database delete operation failure.
  db_s_iter_not_allowed Operation is not allowed while iterating.
  dced_s_bad_binding The dced binding handle supplied did not match the service requested.
  dced_s_not_found A matching object cannot be found. (May be informational only if a match was not expected.)
  sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage
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. It does not delete the program from disk nor does it affect the server if it 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 that 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);

Related Information

Routines

dced_server_create        dced_binding_create        dced_binding_from_rpc_binding
dced_server_modify_attributes

dcecp Objects: server
dced_server_disable_if

Purpose
Disables a service (RPC interface) provided by a specific server on a specific host.

Format
#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 srvrexe
         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.

         Status codes:
         error_status_ok
                      Operation completed successfully.
         db_s_bad_index_type
                      Index type is not known or incorrectly specified.
         db_s_iter_not_allowed
                      Operation is not allowed while iterating.
         db_s_readonly
                      Database is opened read-only.
         db_s_store_failed
                      Database store failure.
         dced_s_bad_binding
                      The dced binding handle supplied did not match the service requested.
         dced_s_no_support
                      Requested function not supported by this release.
         dced_s_not_found
                      A matching object cannot be found. (May be informational only if a match was not expected.)
         sec_acl_invalid_permission
                      Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage

Note:  This routine is not supported when targeted at DCE servers running on MVS hosts.

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 re-enable a service again by calling the `dced_server_enable_if()` routine. (Servers re-enable
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 `srvrexecc`
service by calling either `dced_binding_create()` or `dced_binding_from_rpc_binding()`.

Related Information

**Routines**

<table>
<thead>
<tr>
<th>dce_server_register</th>
<th>dce_server_enable_if</th>
<th>dced_binding_create</th>
</tr>
</thead>
<tbody>
<tr>
<td>dce_server_disable_if</td>
<td>dced_server_enable_if</td>
<td>dced_binding_from_rpc_binding</td>
</tr>
</tbody>
</table>

**dcecp Objects**: server
dced_server_enable_if

Purpose
Enables a service (RPC interface) of a specific server on a specific host.

Format
#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 srvreexec 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.

Parameters

error_status_ok
    Operation completed successfully.

db_s_bad_index_type
    Index type is not known or incorrectly specified.

db_s_iter_not_allowed
    Operation is not allowed while iterating.

db_s_readonly
    Database is opened read-only.

db_s_store_failed
    Database store failure.

dced_s_bad_binding
    The dced binding handle supplied did not match the service requested.

dced_s_no_support
    Requested function not supported by this release.

dced_s_not_found
    A matching object cannot be found. (May be informational only if a match was not expected.)

sec_acl_invalid_permission
    Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage

Note: This routine is not supported when targeted at DCE servers running on MVS hosts.

The dced_server_enable_if() routine enables a service (or re-enables 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 re-enabled, as needed. A server enables and disables its own services by using dce_server_enable_service() and dce_server_disable_service() and a management application enables and disables a remote server's service using 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 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 `srvrexecc` service by calling either `dced_binding_create()` or `dced_binding_from_rpc_binding()`.

**Related Information**

**Routines**

- `dce_server_register`
- `dce_server_enable_if`
- `dce_server_disable_if`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`

**dcecp Objects:** `server`
dced_server_modify_attributes

Purpose
Modifies attributes for a DCE server’s configuration data.

Format
#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. Possible status codes are:

error_status_ok
Operation completed successfully.

db_s_bad_index_type
Index type is not known or incorrectly specified.

db_s_iter_not_allowed
Operation is not allowed while iterating.

db_s_readonly
Database is opened read-only.

db_s_store_failed
Database store failure.

dced_s_bad_binding
The dced binding handle supplied did not match the service requested.

dced_s_not_found
A matching object cannot be found. (May be informational only if a match was not expected.)

sec_acl_invalid_permission
Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

Usage

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().

Related Information

8-164 Application Development Reference Volumes 1 and 2
Routines

dced_object_read  dced_binding_create  dced_binding_from_rpc_binding

dcep Objects:  server
dced_server_start

Purpose
Starts a DCE-configured server on a specified host.

Format
#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. Possible status codes are:

- error_status_ok: Operation completed successfully.
- db_s_bad_header_type: Header type is incorrectly specified.
- db_s_iter_not_allowed: Operation is not allowed while iterating.
- db_s_key_not_found: Specified key was not found in the database. This means the search failed. It is not necessarily a problem.
- db_s_readonly: Database is opened read-only.
- db_s_store_failed: Database store failure.
- dced_s_bad_binding: The dced binding handle supplied did not match the service requested.
- dced_s_no_support: The requested function is not supported by this release.
- dced_s_not_found: A matching object cannot be found. (May be informational only if a match was not expected.)
- dced_s_sc_cant_fork: Server cannot be started because spawn() returned a bad return code. May be insufficient resources to create another process.
- dced_s_sc_invalid_attr_type: Incorrect attribute or attribute type specified for srvrconf object specified not correct.
**Usage**

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()` use 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.

```c

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);
```

**Related Information**

**Routines**

- `dced_server_create`
- `dced_binding_create`
- `dced_binding_from_rpc_binding`
- `dced_server_stop`

**Commands:** server
dced_server_stop

Purpose
Stops a DCE-configured server running on a specific host.

Format
#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, then 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 (SIGTERM).

      srvrexec_stop_hard
          Uses a "hard" local host mechanism (SIGTERM).

      srvrexec_stop_error
          Uses a mechanism that saves the program state
          (SIGABRT).

Output

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

      error_status_ok
          Operation completed successfully.

      dced_s_bad_binding
          The dced binding handle supplied did not match the service requested.

      dced_s_no_support
          The requested function is not supported by this release.

      dced_s_not_found
          A matching object cannot be found. (May be informational only if a match was not expected.)

      rpc_s_binding_incomplete
          Binding does not contain endpoint and object UUID.

      rpc_s_comm_failure
          RPC is not communicating with the remote server.

      rpc_s_invalid_binding
          Binding handle is not correct.

      rpc_s_mgmt_op_disallowed
          Management operation is not allowed.

      rpc_s_unknown_if
          Interface being called is unknown to the server.

      rpc_s_wrong_kind_of_binding
          Type of binding handle (client or server) is not correct for the operation.
dced_server_stop

sec_acl_invalid_permission Security server rejected ACL permission. Request specifies permissions not valid for ACL object being edited.

uuid_s_bad_version UUID version unknown to the RPC runtime.

Usage

Note: This routine is not supported when targeted at DCE servers running on MVS hosts.

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 srvreexec service by calling either dced_binding_create() or dced_binding_from_rpc_binding() routines.

Cautions

Using the value dced_g_uuid_all_servers for the exec_uuid parameter causes dced to shutdown all servers (on non-MVS hosts, this includes 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 srvreexec_stop_soft parameter.

dced_binding_handle_t conf_bh, exec_bh;
dced_string_t server_name;
void *data;
server_t *exec_ptr;
uuid_t srvrconf_id, srvreexec_id;
error_status_t status;
.
.
dced_binding_create("srvrconf@hosts/patrick",
    dced_c_binding_syntax_default,
    &conf_bh,
    &status);
dced_binding_create("srvreexec@hosts/patrick",
    dced_c_binding_syntax_default,
    &exec_bh,
    &status);

/* is server running? */
dced_inq_id(exec_bh, server_name, &srvreexec_id, &status);
/* also check to be sure server is not coming up or going down */
dced_object_read(exec_bh, &srvreexec_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, &srvreexec_id, srvreexec_stop_soft, &status);
}
dced_objects_release(exec_bh, 1, data, &status);
dced_binding_free(conf_bh, &status);
dced_binding_free(exec_bh, &status);
dced_server_stop

Related Information

Routines

dced_server_created  dced_server_start  dced_binding_create  dced_binding_from_rpc_binding
rpc_mgmt_stop_server_listening

dcecp Objects: server
Appendix A. z/OS DCE Supplied Header, IDL and ACF Files

This section contains a description of the header, IDL and ACF files supplied with z/OS DCE.

The following z/OS DCE supplied header files are for general use in your DCE applications:

- acct.h
- binding.h
- codesets.h
- codesets_stub.h
- cs_mgmt.h
- daclif.h
- daclmgr.h
- dce_cf.h
- dce_error.h
- dce_signal.h
- dced.h
- dced_base.h
- exc_handling.h
- keymgmt.h
- lbase.h
- misc.h
- pgo.h
- policy.h
- priv_attr_trig.h
- rdaclif.h
- rdaclifv0.h
- rpc.h
- rsec_pwd_mgmt.h
- sec_cred.h
- sec_login.h
- sec_pwd_mgmt.h
- sec_rgy_attr.h
- sec_rgy_attr_sch.h
- secidmap.h
- unix.h
- utc.h
- uuid.h
- xds.h
- xdsbdcp.h
- xdschs.d
- xsdgsds.h
- xsmdup.h
- xmh.h
- xmsga.h
- xom.h

The following header files may be subject to change, so your applications should not explicitly include them or otherwise depend on them. The files that are in the following list are described for your reference only:

- aclbase.h
- binding.h
- cobolbas.h
- dce_cf_const.h
- dcecfgmsg.h
- dced_aclbits.h
- dced_data.h
- dcedhdmsg.h
- euvpdlws.h
- idlbases.h
- idlddefs.h
- idl_es.h
- id_base.h
- id_epac.h
- iovector.h
- keymgmt.h
- ldrlall.h
- ldrxa.h
- ldrxb.h
- marshall.h
- nbase.h
- ncastat.h
- ndr.types.h
- ndrld.h
- ndrdtypes.h
- ndrdold.h
- oride_base.h
- passwd.h
- rdaclbase.h
- rgybase.h
- rgybasebase.h
- rpcbase.h
- rpcexc.h
- rpcpvt.h
- rpcxdl.h
- rpcxstub.h
- sec_attr_base.h
- sec_attr_tools.h
- secbase.h
- sec_base.h
- sec_login.h
- stubbase.h
- twr.h
- utc.ttypes.h
- xomi.h

The following header files are located in the /usr/lpp/dce/share/include directory. To include these files in your DCE applications, enclose the header file name within angle brackets in your source code, and do not use the dce/ prefix. For example, to include the binding.h header file, use #include <binding.h>. The purpose and content of each file is described as follows:

**Header File Name** | **Description**
---|---
**xds.h** | Contains definitions for the XDS routines and Directory Service Package.

The xds.h header file is a mandatory include file for all applications using the XDS APIs. It declares the interface routines, the structures passed to and from those routines, and the defined constants used by the routines and structures.

All application programs that include this header must first include the xom.h Object Management header.

---

1 If you are using PDS, the header file is named codestub.h.
**xdsbdcp.h**
Contains definitions for Basic Directory Contents Package.

Include the **xdsbdcp.h** header file in your application if you want to use the Basic Directory Contents Package. This header defines the object identifiers of directory attribute types and object classes supported by the **DS_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 file must include the **xom.h** Object Management header and the **xds.h** header.

**xscds.h**
Contains definitions for Cell Directory Services.

Include the **xdsgds.h** header file in your applications if you want to use CDS. This header declares the object identifiers of directory attribute types supported by CDS.

All application programs that include this header file must first include the **xom.h** Object Management header and the **xds.h** header.

**xdsgds.h**
Contains definitions for the Global Directory Service Package.

Include the **xdsgds.h** header file in your applications if you want to use the GDS package. This header declares the object identifiers of directory attribute types and directory object classes supported by the GDS Extension 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.

**xdsmdup.h**

Include the **xdsmdup.h** header file in your applications if you want to use the MHS Directory User Package. This 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 **xom.h** Object Management header and the **xds.h** header.

**xmhp.h**
Contains definitions for the MHS Directory objects/attributes.

Include the **xmhp.h** header file in your applications if you want to use the MHS Directory User Package. This header defines the constants used by the Message Handling Packages. It also contains definitions for the X.400 Message Handling Package. Some of these definitions are needed for negotiating the use of the MDUP.

The **xdsmdup.h** header explicitly includes **xmhp.h**.

**xmsga.h**
Contains definitions for the Message Store General attributes.

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.

**xom.h**
Contains definitions for the XOM routines

The **xom.h** header file is a mandatory include file for all applications working with the XOM APIs. The **xom.h** header defines constants, types, and macros that are used with the interface.

**xomi.h**
Contains definitions of the XOM routines.

The **xomi.h** header contains the definitions for the XOM interface.

The **xom.h** header explicitly includes **xomi.h**. You should not explicitly include this header in your applications.
Header Files in /usr/lpp/dce/share/include/dce

The following header files are located in the /usr/lpp/dce/share/include/dce directory. To include these header files in your DCE applications, use the dce/ prefix before the header file name, all enclosed in angle brackets. For example, to include the rpcexc.h header file, use #include <dce/rpcexc.h>. The purpose and content of each file is described as follows:

<table>
<thead>
<tr>
<th>Header File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct.h</td>
<td>Contains function prototypes generated from acct.idl for the Security User Registry account management APIs. The acct.h header file is a mandatory include file for all applications using the sec_rgy_acct_* APIs. It defines the interface routines for the Security User registry account management APIs.</td>
</tr>
<tr>
<td>aclbase.h</td>
<td>Contains data definitions generated from aclbase.idl for the Security ACL management APIs. You do not have to explicitly include aclbase.h as it is included by daclif.h, daclmgr.h and rdaclif.h. This is the base data types header file for the Security ACL management APIs. Which declares the structures passed to and from the ACL routines and defines the constants used by those routines.</td>
</tr>
<tr>
<td>binding.h</td>
<td>Contains function prototypes and data definitions generated from binding.idl for the Security Registry binding management APIs. The binding.h header file is a mandatory include file for all applications using the sec_rgy_site_* APIs. It defines the interface routines for the Security Registry binding management APIs. Which declares the structures passed to and from the sec_rgy_site_* routines, and defines the constants used by those routines.</td>
</tr>
<tr>
<td>cobolbas.h</td>
<td>Contains function prototypes and data definitions for COBOL to C conversion routines. This header file is only included in the extended server stubs for the Application Support product.</td>
</tr>
<tr>
<td>codesets.h</td>
<td>Contains the function prototypes for the routines that encode/decode the code set management data structures. This file is generated from codesets.idl. The codesets.h header file is a mandatory include file for all servers using the DCE RPC Internationalization support.</td>
</tr>
<tr>
<td>codesets_stub.h</td>
<td>Contains the function prototypes for the stub support routines that process marshalling and unmarshalling of internationalization data. You must include this file in an application that uses the rpc_cs_get_tags, *_local_size, *_to_netcs, or *_from_netcs APIs. If you are using PDS, the codesets_stub.h and codesets.h header files will resolve to the same name (codesets.h) when the file names are reduced to eight characters. Therefore, a duplicate of the file codesets_stubs.h is created and named codestub.h.</td>
</tr>
<tr>
<td>cs_mgmt.h</td>
<td>Contains the code set management data types generated from cs_mgmt.idl.</td>
</tr>
<tr>
<td>daclif.h</td>
<td>Contains function prototypes and data definitions generated from daclif.idl for the client ACL APIs. The daclif.h header file is a mandatory include file for applications using any of the sec_acl_* APIs. It defines the interface routines for the client ACL APIs and defines some constants used by the sec_acl_* routines.</td>
</tr>
<tr>
<td>daclmgr.h</td>
<td>Contains generated function prototypes and data definitions (from daclmgr.idl) for the ACL operations manager APIs local to servers. The daclmgr.h header file is a mandatory include file for all applications using the sec_acl_mgr* functions. It defines the interface routines for the ACL operations manager functions and defines some constants used by the sec_acl_mgr* routines.</td>
</tr>
<tr>
<td>dce_cf.h</td>
<td>Contains the prototypes for the DCE configuration APIs (dce_cf_<em>). You should include this header file in your application if it uses the dce_cf_</em> APIs.</td>
</tr>
<tr>
<td>dce Cf_const.h</td>
<td>Contains cell related constants. As this file is included by dce_cf.h, you should not include it explicitly in your applications.</td>
</tr>
<tr>
<td>dcecfgmsg.h</td>
<td>Provides return values used by the DCE configuration APIs. As this file is included by dce_cf.h, you should not include it explicitly in your applications.</td>
</tr>
</tbody>
</table>
dce_error.h Contains the function prototype and data types for the dce_error_inq_test API.

You should include this header file in your application if it uses the dce_error_inq_test API.

dce_signal.h Redefines the C/C++ sigaction() call to cpa_sigaction(). This is required to change the behavior of signal handlers from process-level handling to thread-level handling. In C/C++, a signal is directed to a process, and a random thread can accept the signal. In DCE a signal is directed to a particular thread.

dce_threads.h Contains definitions for the DCE Threads package.

dced.h Contains data types, data definitions, and prototypes for the DCE Host Daemon (dced) APIs.

dced_aclbits.h Contains data definitions for the DCE Host Daemon (dced) ACL permission bits generated from dced_aclbits.idl.

dced_base.h Contains data types and definitions for the DCE Host Daemon (dced) services generated from dced_base.idl.

dced_data.h Contains data definitions for the DCE Host Daemon (dced).

dcedhdmsg.h Contains definitions for the DCE Host Daemon (dced) status codes.

dcerpcmsg.h Contains the definition of DCE RPC status codes.
You do not need to explicitly include this header file in your application as it is included by the rpc.h header file.

dcesecmsg.h Contains the definition of DCE Security status codes.
You do not need to explicitly include this header file in your application as it is included by other Security header files.

dtsprovider.h Contains the interface definitions for time provider programs.
The dtsprovider.h header file is used for building time provider programs that interact with the DTS daemon.

euvpdlws.h Contains the global variable access for user applications.
You do not explicitly include this header file in your applications. DCE header files exposing global variables will automatically include this header.

exc_handling.h Contains definitions for the DCE Exceptions package.
Include the exc_handling.h header file when using the exceptions constructs, that is, TRY/CATCH.
To use this header file, you need to define the constant _DCE_THREADS.

idlbase.h This file is included by all .h files generated by the IDL compiler. This file defines various primitives that are missing from C but present in IDL (for example, Boolean and handles).
You do not need to explicitly include this header file in your application as it is included by the rpc.h header file.

idlddefs.h This file contains the definitions and prototypes that the IDL-generated stubs need.

idl_es.h This file contains the prototypes for the public IDL encoding service routines.

id_base.h Contains generated data definitions (from id_base.idl) for the Security login and ACL APIs.
The id_base.h header file does not have to be explicitly included as it is included by aclbase.h and sec_login.h. This is the base data types header file defining the structures of Security identity and PAC used by the Security login and ACL routines.

id_epac.h Contains data definitions generated from id_epac.idl for the Security Privilege Attribute and Security
login APIs.
You do not have to explicitly include id_epac.h as it is included by sec_credh and sec_login.h. This is the base data type header file defining the structures of the security EPAC.

iovector.h Contains generated function prototypes and data definitions (from iovector.idl) for the iovector
handling routines.
You do not need to explicitly include this header file in your application as it is included by the rpc.h header file.
keymgmt.h  Contains generated function prototypes and data definitions (from keymgmt.idl) for the Security key management APIs.

The keymgmt.h header file is a mandatory include file for all applications using the sec_key_* APIs. It defines the interface routines for the Security key management APIs and and defines some constants used by sec_key_* routines.

libase.h  Contains generated common local RPC data types (from libase.idl).

You do not need to explicitly include this header in your application as it is included by rpc.h.

ldrall.h  Contains constants, prototypes and macros used by the Application Support product. z/OS DCE applications do not use this header file. For information on using the Application Support product, see z/OS DCE Application Support Programming Guide

ldrx.a.h  This header file is used by the Application Support product. z/OS DCE applications do not use this header file. For information on using the Application Support product, see z/OS DCE Application Support Programming Guide

ldrxb.h  This header file is used by the Application Support product. z/OS DCE applications do not use this header file. For information on using the Application Support product, see z/OS DCE Application Support Programming Guide

marshall.h  This file contains the architecture-specific definitions. This file is always included as part of stubbase.h.

misc.h  Contains generated function prototypes (from misc.idl) for certain Security User Registry miscellaneous APIs.

The misc.h header file is a mandatory include file for all applications using the sec_rgy_login_get_info, sec_rgy_login_get_effective, sec_rgy_wait_until_consistent and sec_rgy_cursor_reset_* APIs. It defines the interface routines for these Security Registry APIs.

nbase.h  Contains generated NCA architecturally defined types and constants (from nbase.idl).

You do not need to explicitly include this header file in your application as it is included by rpc.h.

ncastat.h  Contains generated NCA architecturally defined status codes (from ncastat.idl).

You do not need to explicitly include this header file in your application as it is included by rpc.h.

ndr_rep.h  This file contains the architecture-specific definitions of the local scalar data representation used. This file is always included as part of stubbase.h.

ndrold.h  Contains generated old NDR data types (from ndrold.idl) for compatibility with the old stubs.

You do not need to explicitly include this header file in your application as it is included by rpc.h.

ndrtypes.h  This is a platform-specific file that defines the base-level ndr types. This file is indirectly included in all files via the idlbase.h file.

oride_base.h  Contains generated data definitions (from oride_base.idl) for the Security User Registry login APIs.

The oride_base.h header file does not have to be explicitly included as it is included by misc.h. This is the base data types header file defining the path of the password override file and constants used by the Security password override code.

passwd.h  Contains generated definitions (from passwd.idl) of the base password data structure used by Security APIs.

The passwd.h header file does not have to be explicitly included as it is included by rgynbase.h and sec_login.h. This is the base data types header file defining the Security password structures.

pgo.h  Contains generated function prototypes (from pgo.idl) for the Security User Registry PGO management APIs.

The pgo.h header file is a mandatory include file for all applications using the sec_rgy_pgo_* APIs. It defines the interface routines for the Security user registry PGO management APIs.

policy.h  Contains generated function prototypes (from policy.idl) for the Security User Registry policy management APIs.

The policy.h header file is a mandatory include file for all applications using the sec_rgy_properties_* , sec_rgy_plcy_* and sec_rgy_auth_plcy_* APIs. It defines the interface routines for the Security user registry policy management APIs.
**priv_attr_trig.h**
Contains function prototypes and data definitions generated from **priv_attr_trig.idl** for the Security Attribute Interface API.

This file must be included if you are writing an attribute trigger server that will receive **priv_attr_trig_query** input and supply its output. It defines the interface routine for the Security Attribute interface API.

**rdaclbase.h**
Contains data definitions generated from **rdaclbase.idl** for the security remote ACL management APIs.

You do not have to explicitly include **rdaclbase.h** as it is included by **rdaclif.h** and **rdaclifv0.h**. This is the base data type header file defining the structures for the security remote ACL management APIs.

**rdaclif.h**
Contains generated function prototypes and data definitions (from **rdaclif.idl**) for the remote ACL manager APIs.

The **rdaclif.h** header file is a mandatory include file for all applications using the **rdacl_*** APIs. It defines the interface routines for the remote ACL manager APIs and defines some constants used by the **rdacl_*** routines.

**rdaclifv0.h**
Contains function prototypes generated from **rdaclifv0.idl** for the security remote ACL management APIs.

The file must be included for all applications wanting to use the version 1 interface of the **rdacl_*** APIs. It defines the interface routines for the remote ACL manager APIs.

**rgybase.h**
Contains generated data definitions (from **rgybase.idl**) for the Security client registry agent APIs.

The **rgybase.h** header file does not have to be explicitly included as it is included by **acct.h**, **binding.h**, **misc.h**, **pgo.h**, **policy.h** and **secidmap.h**. It is the base data type header file defining the structures used by the Security client Registry Agent to communicate with the Security daemon.

**rgynbase.h**
Contains generated definitions (from **rgynbase.idl**) of the base network data types used by APIs that communicate with the Security User Registry.

The **rgynbase.h** header file does not have to be explicitly included as it is included by **keymgmt.h** and **rgybase.h**. It is the base data type header file defining the structures used for manipulating the User Registry in the Security daemon.

**rpc.h**
Contains generated function prototypes (from **rpc.idl**) for the RPC runtime APIs.

You must include this file in an application that uses the **rpc_*** APIs.

**rpcbase.h**
Contains generated base RPC data types (from **rpcbase.idl**).

You do not need to explicitly include this header file in your application as it is included by **rpc.h**.

**rpcexc.h**
Contains the definition of RPC exception variables.

You do not need to explicitly include this header file in your application as it is included by **rpc.h**.

**rpcpvt.h**
Contains generated private RPC data types and prototypes (from **rpcbase.idl**). As these declarations are subject to change your RPC applications must not depend on them.

You do not need to explicitly include this header file in your application as it is included by **rpc.h**.

**rpctypes.h**
Contains generated non-local RPC data types (from **rpctypes.idl**).

You do not need to explicitly include this header file in your application as it is included by **rpc.h**.

**rpcxdl.h**
Contains generated data types and status codes (from **rpcxdl.idl**) required for DCE client applications accessing the Application Support product.

This header file must be included in all DCE client applications accessing the Application Support product. See [z/OS DCE Application Support Programming Guide](https://www.ibm.com/support/knowledgecenter/SSLTBW_AIX_7.1.0/com.ibm.aix.cac1.doc/01660200919645.html) for programming with the Application Support product.

**rpcxstub.h**
Contains data types used by the extended stubs for the Application Support product. DCE client applications do not need to include this header file.
rsec_pwd_mgmt.h Contains function prototypes generated from rsec_pwd_mgmt.idl for the security password management server APIs.

You must include this file if you are writing your own password management server. It defines the interface routines for the security password management server APIs.

secabase.h Has the same content as sec_attr_base.h, but is shipped in a PDS for compilation under TSO/E.

sec_attr_base.h Contains data definitions generated from sec_attr_base.idl for the security attribute trigger, security login, and security registry schema APIs.

This file does not have to be explicitly included as it is included by id_epac.idl, sec_attr_trig.idl, sec_login.h, and sec_attr_sch.idl. This is the base data type header file defining the structures of security attributes.

sec_attr_tools.h Contains macros useful for manipulating registry attributes.

secattrig.h Has the same content as sec_attr_trig.h, but is shipped in a PDS for compilation under TSO/E.

sec_attr_trig.h Contains function prototypes and data definitions generated from sec_attr_trig.idl for the security attribute trigger APIs.

This file must be included for all applications using the sec_attr_trig APIs. It defines the interface routines for the security attribute trigger APIs.

sec_base.h Contains generated definitions (from sec_base.idl) of constants used by the Security code.

The sec_base.h header file does not have to be explicitly included as it is included by rgynbase.h. It defines the Security Registry and privilege server names and some other constants used by the Security Registry code.

sec_cred.h Contains function prototypes and data definitions generated from sec_cred.idl for the security privilege attribute APIs.

This file must be included in all applications using the sec_cred_* APIs. It defines the interface routines for the security privilege attribute APIs.

sec_login.h Contains generated function prototypes and data definitions (from sec_login.idl) for the Security login APIs.

The sec_login.h header file is a mandatory include file for all applications using the sec_login_* APIs. It defines the interface routines for the Security login APIs and defines some constants and structures used by sec_login_* routines.

secidmap.h Contains generated function prototypes and data definitions (from secidmap.idl) for the Security identity mapping APIs.

The secidmap.h header file is a mandatory include file for all applications using the sec_id_* APIs. It defines the interface routines for the Security identity mapping APIs.

sec_pwd_mgmt.h Contains function prototypes and data definitions generated from sec_pwd_mgmt.idl for the security password management APIs.

This file must be included in all applications using the sec_pwd_mgmt_* APIs. It defines the interface routines for the security password management APIs.

secrattr.h Has the same content as sec_rgy_attr.h, but is shipped in a PDS for compilation under TSO/E.

secratts.h Has the same content as sec_rgy_attr_sch.h, but is shipped in a PDS for compilation under TSO/E.

sec_rgy_attr.h Contains function prototypes and data definitions generated from sec_rgy_attr.idl for the security registry attribute APIs.

This file must be included in all applications using the sec_rgy_attr_* APIs. It defines the interface routines for the security registry attribute APIs.

sec_rgy_attr_sch.h Contains function prototypes generated from sec_rgy_attr_sch.idl for the security registry attribute schema APIs.

This file must be included in all applications using the sec_rgy_attr_sch_* APIs. It defines the interface routines for the security registry attribute schema APIs.
stubbase.h
Contains prototypes and data types used by the client and server stubs.
You do not need to explicitly include this header file in your application as it is included by the rpc.h header file.

twr.h
Contains generated function prototypes and data definitions (from twr.idl) for internal RPC routines. As these declarations are subject to change and your RPC applications must not depend on them.
You do not need to explicitly include this header file in your application as it is included by the rpc.h header file.

unix.h
Contains function prototypes generated from unix.idl for the security registry UNIX APIs.
This file must be included in all applications using the sec_rgy_unix_* APIs. If defines the interface routines for the security registry UNIX APIs.

utc.h
Contains definitions for the Distributed Time Services programming routines.
The utc.h header file is a mandatory include file for all applications using the UTC APIs. It declares the interface routines, the structures passed to and from those routines, and the defined constants used by the routines.

utctypes.h
Contains the definition for the utc_t type used in the UTC routines.
You do not have to explicitly include this file in your application as it is included by utc.h. The utctypes.h is an IDL generated header file that defines the utc_t type that is passed between UTC routines. The utc_t type is used to represent absolute and relative timestamps.

uuid.h
Contains generated function prototypes and data definitions (from uuid.idl) for the uuid_* APIs.
This header file must included by applications that use that uuid_* APIs but does not include the rpc.h header.

z/OS DCE IDL and ACF Files

The following IDL and ACF files are also located in the /usr/lpp/dce/share/include/dce directory. Many of the header files listed above are generated from the following IDL files. Their content is provided for your reference only.

The purpose and content of each file is described as follows:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct.idl</td>
<td>Contains the interface definition language for generating the function prototypes for the Security User Registry account management APIs.</td>
</tr>
<tr>
<td>aclbase.idl</td>
<td>Contains the interface definition language for generating the data definitions for the Security ACL management APIs.</td>
</tr>
<tr>
<td>binding.idl</td>
<td>Contains the interface definition language for generating the function prototypes and data definitions for the Security Registry binding management APIs.</td>
</tr>
<tr>
<td>codesets.idl</td>
<td>Contains the interface definition language for generating the function prototypes for the routines that encode/decode the code set management data structures.</td>
</tr>
<tr>
<td>cs_mgmt.idl</td>
<td>Contains the interface definition language for generating the code set management data types. This file is shared between rpc.idl and codesets.idl.</td>
</tr>
<tr>
<td>daclif.idl</td>
<td>Contains the interface definition language for generating the function prototypes and data definitions for the client ACL APIs.</td>
</tr>
<tr>
<td>daclmgr.idl</td>
<td>Contains the interface definition language for generating the function prototypes and data definitions for the ACL operations manager APIs local to servers.</td>
</tr>
<tr>
<td>dce_cf_const.idl</td>
<td>Contains the interface definition language for generating the definitions contained in dce_cf_const.h.</td>
</tr>
<tr>
<td>dced_aclbits.idl</td>
<td>Contains the interface definition language for generating the ACL permission bits data definitions for the DCE Host Daemon (dced) APIs.</td>
</tr>
<tr>
<td>dced_base.idl</td>
<td>Contains the interface definition language for generating the data types and definitions for the DCE Host Daemon (dced) APIs.</td>
</tr>
</tbody>
</table>
dcesecmsg.idl Contains the interface definition language to be included by other idl files for inclusion of DCE Security status codes.
dtsprovider.idl Contains interface definitions for time provider servers.
Use the dtsprovider.idl file to build a time provider RPC server that operates with the DTS daemon. It must be used to define the RPC interface to the DTS daemon for a time provider program.
id_base.idl Contains the interface definition language for generating the data definitions for the Security login and ACL APIs.
iovector.idl Contains the interface definition language for generating the function prototypes and data definitions for the iovector handling routines.
keymgmt.idl Contains the interface definition language for generating the function prototypes and data definitions for the Security key management APIs.
lbase.idl Contains the interface definition language for generating the function prototypes and data definitions for the iovector handling routines.
misc.idl Contains the interface definition language for generating the function prototypes for some Security User Registry miscellaneous APIs.
nbase.idl Contains the interface definition language for generating the NCA architecturally defined types and constants.
ncastat.idl Contains the interface definition language for generating the NCA architecturally defined status codes.
ndrold.idl Contains the interface definition language for generating the old NDR data types for compatibility with the old stubs.
oride_base.idl Contains the interface definition language for generating the data definitions for the Security User Registry login APIs.
passwd.idl Contains the interface definition language for generating the data definitions of the base password structure used by Security APIs.
pgo.idl Contains the interface definition language for generating the function prototypes for the Security User Registry PGO management APIs.
policy.idl Contains the interface definition language for generating the function prototypes for the Security User Registry policy management APIs.
rdaclif.idl Contains the interface definition language for generating the function prototypes and data definitions for the remote ACL manager APIs.
rgybase.idl Contains the interface definition language for generating the data definitions for the Security client registry agent APIs.
rgynbase.idl Contains the interface definition language for generating the definitions of the base network data types used by APIs that communicate with the Security User Registry.
rpc.idl Contains the interface definition language for generating the RPC runtime APIs function prototypes.
rpcbase.idl Contains the interface definition language for generating the base RPC data types.
rpcpvt.idl Contains the interface definition language for generating the private RPC data types and prototypes.
rptypes.idl Contains the interface definition language for generating the non-local RPC data types.
rpxdl.idl Contains the interface definition language for generating the data types and status codes required for DCE client applications accessing the Application Server.
sec_base.idl Contains the interface definition language for generating the definitions of constants used by the Security code.
sec_login.idl Contains the interface definition language for generating the function prototypes and data definitions for the Security login APIs.
secidmap.idl Contains the interface definition language for generating the function prototypes and data definitions for the Security identity mapping APIs.
sec_rgy_attr.idl Contains the interface definition language for generating the function prototypes and data definitions for the registry attribute APIs.
**sec_rgy_attr_sch.idl**  Contains the interface definition language for generating the function prototypes and data definitions for the registry attribute schema APIs.

**twr.idl**  Contains the interface definition language for generating the function prototypes and data definitions for the internal RPC routines. As these declarations are subject to change and your RPC applications must not depend on them.

**utctypes.idl**  Contains the interface definition language for generating the `utc_t` type used in the UTC routines.

**uuid.idl**  Contains the interface definition language for generating the function prototypes and data definitions for the `uuid` APIs.

**dtsprovider.acf**  Contains time provider interface attribute information.

The `dtsprovider.acf` file contains information about the status communications failure that occur in RPC calls to time provider programs. Use it when you build a time provider program.
Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.
Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

This product contains code licensed from RSA Data Security Incorporated.

**Trademarks**

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

<table>
<thead>
<tr>
<th>AIX</th>
<th>C/400</th>
</tr>
</thead>
<tbody>
<tr>
<td>CICS</td>
<td>IBM</td>
</tr>
<tr>
<td>IBMLink</td>
<td>IMS/ESA</td>
</tr>
<tr>
<td>Library Reader</td>
<td>RACF</td>
</tr>
<tr>
<td>SecureWay</td>
<td>z/OS</td>
</tr>
</tbody>
</table>

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

**Programming Interface Information**

This [z/OS DCE Application Development Reference](#) documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/OS DCE.
Glossary

This glossary defines technical terms and abbreviations used in z/OS DCE documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS DCE manual or view the IBM Glossary of Computing Terms, located at:

http://www.ibm.com/ibm/terminology

This glossary includes terms and definitions from:

- Information Technology—Portable Operating System Interface (POSIX), from the POSIX series of standards for applications and user interfaces to open systems, copyrighted by the Institute of Electrical and Electronics Engineers (IEEE).
- American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1.SC1).
- Open Software Foundation (OSF).

The following abbreviations indicate terms that are related to a particular DCE service:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDS</td>
<td>Cell Directory Service</td>
</tr>
<tr>
<td>CICS/ESA®</td>
<td>Customer Information Control System/ESA</td>
</tr>
<tr>
<td>DTS</td>
<td>Distributed Time Service</td>
</tr>
<tr>
<td>GDS</td>
<td>Global Directory Service</td>
</tr>
<tr>
<td>IMS/ESA®</td>
<td>Information Management System/ESA</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>Security</td>
<td>Security Service</td>
</tr>
<tr>
<td>Threads</td>
<td>Threads Service</td>
</tr>
<tr>
<td>XDS</td>
<td>X/Open Directory Services</td>
</tr>
<tr>
<td>XOM</td>
<td>X/Open OSI-Abstract-Data Manipulation</td>
</tr>
</tbody>
</table>

A

absolute time. A point on a time scale.

abstract syntax notation one (ASN.1). A data representation scheme that enables complicated types to be defined and enables values of these types to be specified.

access control list (ACL). (1) GDS: Specifies the users with their access rights to an object. (2) Security: Data that controls access to a protected object. An ACL specifies the privilege attributes needed to access the object and the permissions that may be granted, to the protected object, to principals that possess such privilege attributes.

access point. GDS, XDS: The point at which an abstract service is obtained. A connection between a directory user agent (DUA) and a directory system agent (DSA).

access right. Synonym for permission.

accessible. Pertaining to an object whose client possesses a valid designator or handle.

account. Data in the Registry database that allows a principal to log in. An account is a registry object that relates to a principal.

ACF. Attribute configuration file.

ACL. Access control list.

active context handle. RPC: A context handle in RPC applications that the RPC has set to a nonnull value and passed back to the calling program. The calling program supplies the active context handle in any future calls to procedures that share the same client context. See client context and context handle.

directory. An unambiguous name, label, or number that identifies the location of a particular entity or service. See presentation address.

alias. Synonym for alias name.

alias entry. GDS: A directory entry, of object class alias, containing information used to provide an alternative name for an object.

alias name. (1) GDS: A name for a directory object that consists of one or more alias entries in the directory information tree (DIT). (2) Security: An optional alternate for a principal’s primary name. Synonymous with alias. The alias shares the same UUID with the primary name.

aliased object. GDS: An object to which an alias entry refers.
anonymous user. A user who is not entered in the directory as an object and who logs in to the Global Directory Service without giving a name and password.

API. Application program interface.

application program interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application thread. A thread of execution created and managed by application code. See client application thread, local application thread, RPC thread, and server application thread.

architecture. (1) The organizational structure of a computer system, including the interrelationships among its hardware and software. (2) The logical structure and operating principles of a computer network. The operating principles of a network include those of services, functions, and protocols.

ASN.1. Abstract syntax notation one.

association (connection-oriented). A connection between a client and a server.

asynchronous. Without a regular time relationship; unexpected or unpredictable with respect to the running of program instructions.

attribute. (1) RPC: An Interface Definition Language (IDL) or attribute configuration file (ACF) that conveys information about an interface, type, field, parameter, or operation. (2) DTS: A qualifier used with DTS commands. DTS has four attribute categories: characteristics, counters, identifiers, and status. (3) XDS: Information of a particular type concerning an object and appearing in an entry that describes the object in the directory information base (DIB). It denotes the attribute’s type and a sequence of one or more attribute values, each accompanied by an integer denoting the value’s syntax.

attribute configuration file (ACF). RPC: An optional companion to an interface definition file that changes how the Interface Definition Language (IDL) compiler locally interprets the interface definition. See also interface definition and Interface Definition Language.

attribute syntax. GDS: A definition of the set of values that an attribute may assume. Attribute syntax includes the data type, in ASN.1, and usually one or more matching rules by which values may be compared.

attribute type. (1) XDS: The component of an attribute that indicates the type of information given by that attribute. Because it is an object identifier, it is unique among other attribute types. (2) XOM: Any of various categories into which the client dynamically groups values on the basis of their semantics. It is an integer unique only within the package.

attribute value. XDS, XOM: A particular instance of the type of information indicated by an attribute type.

attribute value assertion (AVA). GDS: An attribute type and attribute value pair. A relative distinguished name is comprised of one or more AVAs.

authentication. In computer security, a method used to verify the identity of a principal.

authentication level. Synonym for protection level.


Authentication Service. One of three services provided by the Security Service: it verifies principals according to a specified authentication protocol. The other Security services are the Privilege Service and the Registry Service.

authorization. (1) The determination of a principal’s permissions with respect to a protected object. (2) The approval of a permission sought by a principal with respect to a protected object.

authorization protocol. A formal procedure for establishing the authorization of principals with respect to protected objects. Authorization protocols supported by the Security Service include DCE authorization and name-based authorization.

authorization service. RPC: An implementation of an authorization protocol.

automatic binding method. RPC: A method of managing the binding for a remote procedure call. It completely hides binding management from client application code. If the client makes a series of remote procedure calls, the stub passes the same binding handle with each call. See binding handle, explicit binding method, and implicit binding method.

AVA. Attribute value assertion.

B

Basic Encoding Rules (BER). A set of rules used to encode ASN.1 values as strings of octets.

BER. Basic Encoding Rules.

big endian. An attribute of data representation that reflects how multi-octet data is stored. In big endian representation, the lowest addressed octet of a multi-octet data item is the most significant. See little endian.

binary timestamp. An opaque 128-bit (16-octet) structure that represents a DTS time value.

binding. RPC: A relationship between a client and a server involved in a remote procedure call.
binding handle.  RPC: A reference to a binding. See binding information.

binding information.  RPC: Information about one or more potential bindings, including an RPC protocol sequence, a network address, an endpoint, at least one transfer syntax, and an RPC protocol version number.  See binding. See also endpoint, network address, RPC protocol, RPC protocol sequence, and transfer syntax.

broadcast.  A notification sent to all members within an arbitrary grouping such as nodes in a network or threads in a process. See also signal.

Browser.  CDS: A Motif-based program that lets users view the contents and structure of a cell name space.

Cell Directory Service (CDS).  A DCE component.  A distributed replicated database service that stores names and attributes of resources located in a cell.  CDS manages a database of information about the resources in a group of machines called a DCE cell.

cell-relative name.  Synonym for local name.

central processing unit (CPU).  The part of a computer that includes the circuits that control the interpretation and processing of instructions.

chaining.  GDS, XDS: A mode of interaction optionally used by a directory system agent (DSA) that cannot perform an operation itself.  The DSA chains by calling the operation in another DSA and then relaying the outcome to the original requester.

child process.  A process, created by a parent process, that shares the resources of the parent process to carry out a request.  Contrast with parent process. See also fork.

CICS®.  Customer Information Control System.

class.  A category into which objects are placed on the basis of their purpose and internal structure.

clerk.  (1) DTS: A software component that synchronizes the clock for its client system by requesting time values from servers, calculating a new time from the values, and supplying the computed time to client applications.  (2) CDS: A software component that receives CDS requests from a client application, ascertains an appropriate CDS server to process the requests, and returns the results of the requests to the client application.

client.  A computer or process that accesses the data, services, or resources of another computer or process on the network.  Contrast with server.

client application thread.  RPC: A thread executing client application code that makes one or more remote procedure calls.  See application thread, local application thread, RPC thread, and server application thread.

client binding information.  Information about a calling client provided by the client runtime to the server runtime, including the address where the call originated, the RPC protocol used for the call, the requested object UUID, and client authentication information.  See binding information and server binding information.

client context.  RPC: The state within an RPC server generated by a set of remote procedures and maintained across a series of calls for a particular client.  See context handle.  See also manager.

client stub.  RPC: The surrogate code for an RPC interface that is linked with and called by the client application code. In addition to general operations such as marshalling data, a client stub calls the RPC runtime to perform remote
procedure calls and, optionally, to manage bindings. See server stub.

clock. The combined hardware interrupt timer and software register that maintains the system time.

code page. (1) A table showing codes assigned to character sets. (2) An assignment of graphic characters and control function meanings to all code points. (3) Arrays of code points representing characters that establish numeric order of characters. [OSF] (4) A particular assignment of hexadecimal identifiers to graphic elements. (5) Synonymous with code set. (6) See also code point, extended character.

code set. Synonym for code page.

collapse. CDS: To remove the contents of a directory from the display (close it) using the CDS Browser. To collapse an open directory, double-click on its icon. Double-clicking on a closed directory expands it. Contrast with expand.

compatible server. RPC: A server that offers the requested RPC interface and RPC object and that is accessible over a valid combination of network and transport protocols. It is supported by both the client and server RPC run times.

computed time. DTS: The resulting time after a DTS clock synchronization. The time value that the clerk or server process computes according to the values it receives from several servers.

condition variable. Threads: A synchronization object used in conjunction with a mutex. It allows a thread to suspend running until some condition is true.

conformant array. RPC: An array whose size is determined at runtime. A structure containing a conformant array as a field is a conformant structure.

connectionless protocol. RPC: A transport protocol such as UDP that does not require a connection to be established prior to data transfer. Contrast with connection-oriented protocol.

connection-oriented protocol. RPC: An RPC protocol that runs over a connection-based transport protocol. It is a connection-based, reliable, virtual-circuit transport protocol, such as TCP. Contrast with connectionless protocol.

Consultative Committee on International Telegraphy and Telephone (CCITT). A United Nations Specialized Standards group whose membership includes common carriers concerned with devising and proposing recommendations for international telecommunications representing alphabets, graphics, control information, and other fundamental information interchange issues.

context handle. RPC: A reference to state (client context) maintained across remote procedure calls by a server on behalf of a client. See client context.

control access. CDS: An access right that grants users the ability to change the access control on a name and to perform other powerful management tasks, such as replicate a directory or move a clearinghouse.

convergence. CDS: The degree to which CDS attempts to keep all replicas of a directory consistent. Two factors control the persistence and speed at which CDS keeps directory replicas up to date: the setting of a directory’s CDS_Convergence attribute (high, medium, or low) and the background skulk time. By default, every directory inherits the convergence setting of its parent.

conversation key. Synonym for session key.

copy. GDS, XDS: Either a copy of an entry stored in other DSAs through bilateral agreement or a locally and dynamically stored copy of an entry resulting from a request (a cache copy).

CPU. central processing unit

creation timestamp (CTS). An attribute of all CDS clearinghouses, directories, soft links, child pointers, and object entries that contains a unique value reflecting the date and time the name was created. The timestamp consists of two parts; a time portion and a portion containing the system identifier of the node on which the name was created. These two parts guarantee uniqueness among timestamps generated on different nodes.

credentials. Security: A general term for privilege attribute data that has been certified by a trusted privilege certification authority.

CTS. Creation timestamp.

Customer Information Control System (CICS). An IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases.

D

daemon. (1) A long-lived process that runs unattended to perform continuous or periodic system-wide functions such as network control Some daemons are triggered automatically to perform their task; others operate periodically. An example is the cron daemon, which periodically performs the tasks listed in the crontab file. Many standard dictionaries accept the spelling demon. (2) A DCE server process.

daemon configuration file. A file containing information on which daemons are configured on the host, which environment variables to set, the parameters to pass to the process, minimum restart interval, and the time-out period.

Data Encryption Standard (DES). The National Institute of Standards and Technology (NIST) Data Encryption Standard,
adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

datagram. RPC: A network data packet that is independent of all other packets and does not guarantee delivery or sequentiality.

datagram protocol. RPC: A datagram-based transport protocol, such as User Datagram Protocol (UDP), that runs over a connectionless transport protocol.

DCE. Distributed Computing Environment.

DCECONF. program used to configure and start the DCE daemons.

DCEKERN. The address space that contains the DCE daemons.

decrypt. Security: To decipher data.

default DSA. GDS: The directory system agent (DSA) generally used when the user does not specify any particular DSA when connecting to the directory system.

default element. RPC: An optional profile element that contains a nil interface identifier and object UUID and that specifies a default profile. Each profile can contain only one default element. See default profile, profile, and profile element.

default profile. RPC: A backup profile referred to by the default element in another profile. The NSI import and lookup operations use the default profile, if present, whenever a search based on the current profile fails to find any useful binding information. See default element and profile.

default profile. RPC: A backup profile referred to by the default element in another profile. The NSI import and lookup operations use the default profile, if present, whenever a search based on the current profile fails to find any useful binding information. See default element and profile.

DES. Data Encryption Standard.

descriptor. (1) XOM: The means by which the client and service exchange an attribute value and the integers that denote its representation, type, and syntax. (2) XDS: A defined data structure that is used to represent an OM attribute type and a single value.

descriptor list. GDS: An ordered sequence of descriptors that is used to represent several OM attribute types and values.

destructor. A user-supplied routine that is expected to finalize and then deallocate a per-thread context value.

DFS. Distributed File Service.

DIB. Directory information base.

directory. (1) A logical unit for storing entries under one name (the directory name) in a CDS namespace. Each physical instance of a directory is called a replica. (2) A collection of open systems that cooperates to hold a logical database of information about a set of objects in the real world.

directory ID. Directory identifier.

directory information base (DIB). GDS: The complete set of information to which the directory provides access, which includes all of the pieces of information that can be read or manipulated using the operations of the directory.

directory information tree (DIT). GDS: The directory information base (DIB) considered as a tree, whose vertices (other than the root) are the directory entries.

directory schema. GDS: The set of rules and constraints concerning directory information tree (DIT) structure, object class definitions, attribute types, and syntaxes that characterize the directory information base (DIB).


directory system. GDS: A system for managing a directory, consisting of one or more DSA. Each DSA manages part of the DIB.

directory system agent (DSA). GDS: An open systems interconnection (OSI) application process that is part of the directory.

directory system protocol (DSP). GDS: The protocol used by a directory system agent (DSA) to access another DSA. The DSA runs in the GDS server machine and manages the GDS data base.

directory user agent (DUA). GDS: An open systems interconnection (OSI) application process that represents a user accessing the directory.

discriminator. RPC: The data item that determines which union case is currently used.

distinguished name (DN). GDS: One of the names of an object, formed from the sequence of RDNs of its object entry and each of its superior entries.

distinguished value. GDS: An entry’s attribute value that has been designated to appear in the RDN of the entry.

distributed computing. A type of computing that allows computers with different hardware and software to be combined on a network, to function as a single computer, and to share the task of processing application programs.

Distributed Computing Environment (DCE). A comprehensive, integrated set of services that supports the development, use, and maintenance of distributed applications. DCE is independent of the operating system and network; it provides interoperability and portability across heterogeneous platforms.
Distributed File Service (DFS). A DCE component. DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file. Files are part of a single, global name space, so that a user can be found anywhere in the network by means of the same name.

Distributed Time Service (DTS). A DCE component. It provides a way to synchronize the times on different hosts in a distributed system.

DIT. Directory information tree.

DN. Distinguished name.

DNS. Domain Name System.

Domain Name System (DNS). A hierarchical scheme for giving meaningful names to hosts in a TCP/IP network.

domain name. A unique network name that is associated with a network’s unique address.

DSA. Directory system agent.

DSP. Directory system protocol.

DTS. Distributed Time Service.

DTS entity. DTS: The server or clerk software on a system.

DUA. Directory user agent.

DUA cache. GDS: The part of the DUA that stores information to optimize name lookups. Each cache contains copies of recently accessed object entries as well as information about DSAs in the directory.

dynamic endpoint. RPC: An endpoint that is generated by the RPC runtime for an RPC server when the server registers its protocol sequences. It expires when the server stops running. See endpoint and well-known endpoint.

E

effective permissions. Security: The permissions granted to a principal as a result of a masking operation.

element. RPC: Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.

encrypt. To systematically encode data so that it cannot be read without knowing the coding key.

encryption key. A value used to encrypt data so that only possessors of the encryption key can decipher it.

endian. An attribute of data representation that reflects how certain multi-octet data is stored in memory. See big endian and little endian.

effective permissions. Security: The permissions granted to a principal as a result of a masking operation.

element. RPC: Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.
explicit binding method. RPC: The explicit method of managing the binding for a remote procedure call in which a remote procedure call passes a binding handle as its first parameter. The binding handle is initialized in the application code. See automatic binding method, binding handle, and implicit binding method.

export. (1) RPC: To place the server binding information associated with an RPC interface or a list of object UUIDs or both into an entry in a name service database. (2) To provide access information for an RPC interface. Contrast with unexport.

F

fault. RPC: An exception condition, occurring on a server, that is transmitted to a client.

filter. An assertion about the presence or value of certain attributes of an entry to limit the scope of a search.

FIFO. first-in-first-out

first-in-first-out (FIFO). A queueing technique in which the next item to be retrieved is the item that has been in the queue the longest time.

fixed array. RPC: The size of the array is defined in the IDL. All of the data in the array is transmitted during a remote procedure call.

foreign cell. A cell other than the one to which the local machine belongs. A foreign cell and its binding information are stored in either GDS or the Domain Name System (DNS). The act of contacting a foreign cell is called intercell. Contrast with local cell.

fork. To create and start a child process. Forking is similar to creating an address space and attaching. It creates a copy of the parent process, including open file descriptors.

full name. CDS: The complete specification of a CDS name, including all parent directories in the path from the cell root to the entry being named.

full pointer. RPC: A pointer without the restrictions of a reference pointer.

fully bound binding handle. RPC: A server binding handle that contains a complete server address including an endpoint. Contrast with partially bound binding handle.

G

General-Use Programming Interface (GUPI). An interface, with few restrictions, for use in customer-written programs. The majority of programming interfaces are general-use programming interfaces, and are appropriate in a wide variety of application programs. A general-use programming interface requires the knowledge of the externals of the interface and perhaps the externals of related programming interfaces. Knowledge of the detailed design or implementation of the software product is not required.

GDS. Global Directory Service.

Global Directory Agent (GDA). A DCE component that makes it possible for the local CDS to access names in foreign cells. The GDA provides a connection to foreign cells through either the GDS or the Domain Name System (DNS).

Global Directory Service (GDS). A DCE component. A distributed replicated directory service that provides a global namespace that connects the local DCE cells into one worldwide hierarchy. DCE users can look up a name outside a local cell with GDS.

global name. A name that is universally meaningful and usable from anywhere in the DCE naming environment. The prefix /... indicates that a name is global.

group. (1) RPC: A name service entry that corresponds to one or more RPC servers that offer common RPC interfaces, RPC objects, or both. A group contains the names of the server entries, other groups, or both that are members of the group. See NSI group attribute. (2) Security: Data that associates a named set of principals that can be granted common access rights. See subject identifier.

group member. (1) RPC: A name service entry whose name occurs in the group. (2) Security: A principal whose name appears in a security group. See group.

H

handle. RPC: An opaque reference to information. See binding handle, context handle, interface handle, name service handle, and thread handle.

heterogeneous. Pertaining to a collection of dissimilar host computers such as those from different manufacturers. Contrast with homogeneous.

home cell. Synonym for local cell.

homogeneous. Pertaining to a collection of similar host computers such as those of one model or one manufacturer. Contrast with heterogeneous.

host ID. Synonym for network address.

I

identity mapping. Application Support Server: A record in the Security Registry that contains the mapping between a client’s DCE identity and an MVS user ID.

IDL. Interface Definition Language.

IDL compiler. RPC: A compiler that processes an RPC interface definition and an optional attribute configuration file
(ACF) to generate client and server stubs, and header files. See Interface Definition Language.

**immediate superior.** GDS: In the directory information tree (DIT), an entry is the immediate superior of another if its distinguished name, followed by the relative distinguished name (RDN) of the other, forms the distinguished name of the other entry.

**implicit binding method.** RPC: The implicit method of managing the binding for a remote procedure call in which a global variable in the client application holds a binding handle that the client stub passes to the RPC runtime. See automatic binding method, binding handle, and explicit binding method.

**import.** (1) RPC: To obtain binding information from a name service database about a server that offers a given RPC interface by calling the RPC NSI import operation. (2) RPC: To incorporate constant, type, and import declarations from one RPC interface definition into another RPC interface definition by means of the IDL import statement.

**import context.** The context set up by the client to import compatible binding handles from the name space. Name service interfaces (NSI) are used to set up and free the import context.

**IMS™.** Information Management System.

**inaccessible.** Pertaining to an object for which the client does not possess a valid designator or handle.

**inaccuracy.** DTS: The bounded uncertainty of a clock value as compared to a standard reference.

**Information Management System (IMS).** A database and data communication system capable of managing complex databases and networks in virtual storage.

**inquiry context.** The context set up by the client, server, or management applications to view the elements in a name space profile. Name service interfaces (NSI) are used to set up and free the inquiry context.

**instance.** XOM: An object in the category represented by a class.

**instance UUID.** RPC: An object Universal Unique Identifier (UUID) that is associated with a single server instance and is provided to clients to identify that instance unambiguously. See object UUID and server instance.

**integrity.** RPC: A protection level that may be specified in secure RPC communications to ensure that data transferred between two principals has not been changed in transit.

**interface.** RPC: A shared boundary between two or more functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. See RPC interface.

**interface definition.** RPC: A description of an RPC interface written in the DCE Interface Definition Language (IDL). See RPC interface.

**Interface Definition Language (IDL).** A high-level declarative language that provides syntax for interface definitions.

**interface handle.** RPC: A reference in code to an interface specification. See binding handle and interface specification.

**interface identifier.** RPC: A string containing the interface Universal Unique Identifier (UUID) and major and minor version numbers of a given RPC interface. See RPC interface.

**interface UUID.** RPC: The Universal Unique Identifier (UUID) generated for an RPC interface definition using the UUID generator. See interface definition and RPC interface.

**International Organization for Standardization (ISO).** An international body composed of the national standards organizations of 89 countries. ISO issues standards on a vast number of goods and services including networking software.

**Internet address.** The 32-bit address assigned to hosts in a TCP/IP network.

**Internet Protocol (IP).** In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment. IP provides the interface from the higher level host-to-host protocols to the local network protocols. Addressing at this level is usually from host to host.

**interval.** DTS: The combination of a time value and the inaccuracy associated with it; the range of values represented by a combined time and inaccuracy notation. As an example, the interval 08:00.00100:05:00 (eight o’clock, plus or minus five minutes) contains the time 07:57.00.

**invoke ID.** XDS: An integer used to distinguish one asynchronous (directory) operation from all other outstanding ones.

**IP.** Internet Protocol

**ISO.** International Organization for Standardization
junction. A specialized entry in the DCE namespace that contains binding information to enable communications between different DCE services.

Kerberos. The authentication protocol used to carry out DCE private key authentication. Kerberos was developed at the Massachusetts Institute of Technology.

Kerberos. The authentication protocol used to carry out DCE private key authentication. Kerberos was developed at the Massachusetts Institute of Technology.

key. A value used to encrypt and decrypt data.

key file. A file that contains encryption keys for noninteractive principals.

knowledge reference. GDS: Knowledge that associates, either directly or indirectly, a DIT entry with the DSA in which it is located.

layer. In network architecture, a group of services, functions, and protocols that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture.

LDAP. The Lightweight Directory Access Protocol API.

leaf entry. A directory entry that has no subordinates. It can be an alias entry or an object entry.

LFS. local file system

Lightweight Directory Access Protocol (LDAP). An interface that provides access through TCP/IP to directory services which accept the LDAP protocol.

little endian. An attribute of data representation that reflects how multi-octet data is stored. In little endian representation, the lowest addressed octet of a multi-octet data item is the least significant. See big endian.

local. (1) Pertaining to a device directly connected to a system without the use of a communication line. (2) Pertaining to devices that have a direct, physical connection. Contrast with remote.

local application thread. RPC: An application thread that runs within the confines of one address space on a local system and passes control exclusively among local code segments. See application thread, client application thread, RPC thread and server application thread.

local cell. The cell to which the local machine belongs. Synonymous with home cell. Contrast with foreign cell.

local file system (LFS). An organized collection of data in the form of a root directory and its subdirectories and files.

An LFS supports special features useful in a distributed environment: the ability to replicate data; to log file system data, enabling quick recovery after a crash; to simplify administration by dividing the file system into easily managed units called filesets; and to associate access control lists (ACLs) with files and directories. An LFS is located on a disk that is physically attached to a machine. In other file systems, a single disk partition contains only one file system. In DCE LFS an aggregate can contain multiple file systems (filesets). See also access control list (ACL).

local name. A name that is meaningful and usable only within the cell where an entry exists. The local name is a shortened form of a global name. Local names begin with the prefix /.; and do not contain a cell name. Synonymous with cell-relative name.

local server. DTS: A server that synchronizes with its peers and provides its clock value to other servers and clerks in the same network.

local type. RPC: A type named in a represent_as clause and used by application code to manipulate data that is passed in a remote procedure call as a network type. See network type.

logical unit (LU). A host port through which a user gains access to the services of a network.

lookup context. The context setup by the client to locate compatible binding handles from the name space. Name service interfaces (NSI) are used to setup and free the lookup context.

M

manager. RPC: A set of remote procedures that implement the operations of an RPC interface and that can be dedicated to a given type of object. See also object and RPC interface.

manager entry point vector. RPC: The runtime code on the server side uses this entry point vector to dispatch incoming remote procedure calls. See entry point vector and manager.

manager thread. See call thread.

marshalling. RPC: The process by which a stub converts local arguments into network data and packages the network data for transmission. Contrast with unmarshalling.

mask. (1) A pattern of characters used to control the retention or deletion of portions of another pattern of characters (2) Security: Used to establish maximum permissions that can then be applied to individual ACL entries. (3) GDS: The administration screen interface menus.

master replica. CDS: The first instance of a specific directory in the namespace. After copies of the directory
have been made, a different replica can be designated as the master, but only one master replica of a directory can exist at a time. CDS can create, update, and delete object entries and soft links in a master replica.

**mutex.** Mutual exclusion. A read/write lock that grants access to only a single thread at any one time. A mutex is often used to ensure that shared variables are always seen by other threads in a consistent way.

**N**

**name.** GDS, CDS: A construct that singles out a particular (directory) object from all other objects. A name must be unambiguous (denote only one object); however, it need not be unique (be the only name that unambiguously denotes the object).

**name service.** A central repository of named resources in a distributed system. In DCE, this is the same as Directory Service.

**name service handle.** RPC: An opaque reference to the context used by the series of next operations called during a specific name service interface (NSI) search or inquiry.

**name service interface (NSI).** RPC: A part of the application program interface (API) of the RPC run time. NSI routines access a name service, such as CDS, for RPC applications.

**namespace.** CDS: A complete set of CDS names that one or more CDS servers look up, manage, and share. These names can include directories, object entries, and soft links.

**naming attribute.** GDS: An attribute used to form the relative distinguished name (RDN) of an entry.

**NCA.** Network Computing Architecture.

**NDR.** Network Data Representation.

**network.** A collection of data processing products connected by communications lines for exchanging information between stations.

**network address.** An address that identifies a specific host on a network. Synonymous with host ID.

**Network Computing Architecture (NCA).** RPC: An architecture for distributing software applications across heterogeneous collections of networks, computers, and programming environments using UDP. NCA specifies part of the DCE Remote Procedure Call architecture.

**network data.** RPC: Data represented in a format defined by a transfer syntax. See also transfer syntax.

**Network Data Representation (NDR).** RPC: The transfer syntax defined by the Network Computing Architecture. See transfer syntax.

**network protocol.** A communications protocol from the Network Layer of the Open Systems Interconnection (OSI) network architecture, such as the Internet Protocol (IP).

**network type.** RPC: A type defined in an interface definition and referenced in a represent_as clause that is converted into a local type for manipulation by application code. See local type.

**node.** (1) An endpoint of a link, or a junction common to two or more links in a network. Nodes can be preprocessors, controllers, or workstations, and they can vary in routing and other functional capabilities. (2) In network topology, the point at an end of a branch. It is usually a physical machine.

**nonspecific subordinate reference.** GDS: A knowledge reference that holds information about the directory system agent (DSA) that holds one or more unspecified subordinate entries.

**NSI.** Name service interface.

**NSI attribute.** RPC: An RPC-defined attribute of a name service entry used by the RPC name service interface. A name service interface (NSI) attribute stores one of the following: binding information, object Universal Unique Identifiers (UUIDs), a group, or a profile. See NSI binding attribute, NSI group attribute, NSI object attribute, and NSI profile attribute.

**NSI binding attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry; the binding attribute stores binding information for one or more interface identifiers offered by an RPC server and identifies the entry as an RPC server entry. See binding information and NSI object attribute. See also server entry.

**NSI group attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the entry names of the members of an RPC group and identifies the entry as an RPC group. See group.

**NSI object attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the object UUIDs of a set of RPC objects. See object.

**NSI profile attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores a collection of RPC profile elements and identifies the entry as an RPC profile. See profile.

**NULL.** In the C language, a pointer that does not point to a data object.
**Object**. (1) A data structure that implements some feature and has an associated set of operations. (2) RPC: For RPC applications, anything that an RPC server defines and identifies to its clients using an object Universal Unique Identifier (UUID). An RPC object is often a physical computing resource such as a database, directory, device, or processor. Alternatively, an RPC object can be an abstraction that is meaningful to an application, such as a service or the location of a server. See **object UUID**. (3) XDS: Anything in the world of telecommunications and information processing that can be named and for which the directory information base (DIB) contains information. (4) XOM: Any of the complex information objects created, examined, changed, or destroyed by means of the interface.

**Object class**. GDS, CDS: An identified family of objects that share certain characteristics. An object class can be specific to one application or shared among a group of applications. An application interprets and uses an entry’s class-specific attributes based on the class of the object that the entry describes.

**Object class table (OCT)**. A recurring attribute of the directory schema with the description of the object classes permitted.

**Object entry**. CDS: The name of a resource (such as a node, disk, or application) and its associated attributes, as stored by CDS. CDS administrators, client application users, or the client applications themselves can give a resource an object name. CDS supplies some attribute information (such as a creation timestamp) to become part of the object, and the client application may supply more information for CDS to store as other attributes. See **entry**.

**Object identifier (OID)**. A value (distinguishable from all other such values) that is associated with an information object. It is formally defined in the CCITT X.208 standard.

**Object management (OM)**. The creation, examination, change, and deletion of potentially complex information objects.

**Object name**. CDS: A name for a network resource.

**Object UUID**. RPC: The Universal Unique Identifier (UUID) that identifies a particular RPC object. A server specifies a distinct object UUID for each of its RPC objects. To access a particular RPC object, a client uses the object UUID to find the server that offers the object. See **object**.

**OCT**. Object class table.

**Octet**. A byte that consists of eight bits.

**OID**. Object identifier.

**OM**. Object management.

**OM attribute**. XOM: An object management (OM) attribute consists of one or more values of a particular type (and therefore syntax).

**OM class**. XOM: A static grouping of object management (OM) objects, within a specification, based on both their semantics and their form.

**Opaque**. A datum or data type whose contents are not visible to the application routines that use it.

**Open Software Foundation (OSF)**. A nonprofit research and development organization set up to encourage the development of solutions that allow computers from different vendors to work together in a true open-system computing environment.

**Open system**. A system whose characteristics comply with standards made available throughout the industry and that can be connected to other systems complying with the same standards.

**Open systems interconnection (OSI)**. The interconnection of open systems in accordance with standards of the International Organization for Standardization (ISO) for the exchange of information.

**Operation**. (1) GDS: Processing performed within the directory to provide a service, such as a read operation. (2) RPC: The task performed by a routine or procedure that is requested by a remote procedure call.

**Organization**. (1) The third field of a subject identifier. (2) Security: Data that associates a named set of users who can be granted common access rights that are usually associated with administrative policy.

**OSF**. Open Software Foundation.

**OSI**. Open systems interconnection

**P**

**PAC**. Privilege attribute certificate.

**Package**. XOM: A specified group of related object management (OM) classes, denoted by an object identifier.

**Packet**. (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. [1] The data, call control signals, and error control information are arranged in a specific format. (2) See **call-accepted packet**, **call-connected packet**, **call-request packet**. See **clear-confirmation packet**, **clear-indication packet**, **clear-request packet**. See **data packet**, **incoming-call packet**.

**Parent directory**. CDS: Any directory that has one or more levels of directories beneath it in a cell name space. A directory is the parent of any directory immediately beneath it in the hierarchy.
**parent process.** A process created to carry out a program. The parent process in turn creates child processes to process requests. Contrast with *child process.*

**partially bound binding handle.** RPC: A server binding handle that contains an incomplete server address lacking an endpoint. Contrast with *fully bound binding handle.*

**Partitioned data set (PDS).** A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

**password.** A secret string of characters shared between a computer system and a user. The user must specify the character string to gain access to the system.

**PCS.** Portable Character Set.

**PDS.** Partitioned data set

**permission.** (1) The modes of access to a protected object. The number and meaning of permissions with respect to an object are defined by the access control list (ACL) Manager of the object. (2) GDS: One of five groups that assigns modes of access to users: MODIFY PUBLIC, READ STANDARD, MODIFY STANDARD, READ SENSITIVE, or MODIFY SENSITIVE. Synonymous with access right. See also access control list.

**person.** See *principal.*

**pickle.** A type of data encoding. When a Remote Procedure Call (RPC) sends data between a client and a server, it serializes the user's data structures by using the IDL Encoding Services (ES). This serialization scheme for encoding and decoding data is informally called pickling.

**pipe.** (1) RPC: A mechanism for passing large amounts of data in a remote procedure call. (2) The data structure that represents this mechanism.

**plaintext.** The input to an encryption function or the output of a decryption function. Encryption transforms plaintext to ciphertext and decryption transforms ciphertext into plaintext.

**platform.** The operating system environment in which a program runs.

**port.** (1) Part of an Internet Protocol (IP) address specifying an endpoint. (2) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer.

**Portable Character Set.** A set of characters to enable internationalization. A character set used by DCE to enable word wide connectivity by ensuring that a minimum group of characters is supported in DCE. All DCE RPC clients and servers are required to support the DCE PCS.

**position (within an attribute).** XOM: The ordinal position of one element of a string relative to another.

**predicate.** A Boolean logic term denoting a logical expression that determines the state of some variables. For example, a predicate can be an expression stating that variable A must have the value 3. The control expression used in conjunction with condition variables is based on a predicate. A condition variable can be used to wait for some predicate to become true, for example, to wait for something to be in a queue.

**presentation address.** An unambiguous name that is used to identify a set of presentation service access points. Loosely, it is the network address of an open systems interconnection (OSI) service.

**primary name.** The string name of an object to which any aliases for that object refer. The DCE refers to objects by their primary names, although DCE users may refer to them by their aliases.

**principal.** Security: An entity that can communicate securely with another entity. In the DCE, principals are represented as entries in the Registry database and include users, servers, computers, and authentication surrogates.

**privacy.** RPC: A protection level that encrypts RPC argument values. in secure RPC communications.

**private key.** See *secret key.*

**private object.** (1) XDS: An OM object created in a work space using the object management functions. Contrast with public object. (2) XOM: An object that is represented in an unspecified fashion.

**privilege attribute.** Security: An attribute of a principal that may be associated with a set of permissions. DCE privilege attributes are identity-based and include the principal's name, group memberships, and local cell.

**privilege attribute certificate (PAC).** Security: Data describing a principal's privilege attributes that has been certified by an authority. In the DCE, the Privilege Service is the certifying authority; it seals the privilege attribute data in a ticket. The authorization protocol, DCE Authorization, determines the permissions granted to principals by comparing the privilege attributes in PACs with entries in an access control list.

**privilege service.** Security: One of three services provided by the Security Service; the Privilege Service certifies a principal's privileges. The other services are the Registry Service and the Authentication Service.

**privilege ticket.** Security: A ticket that contains the same information as a simple ticket, and also includes a privilege attribute certificate. See *service ticket, simple ticket,* and *ticket-granting ticket.*
procedure declaration. RPC: The syntax for an operation, including its name, the data type of the value it returns (if any), and the number, order, and data types of its parameters (if any).

product-sensitive programming interface (PSPI). (1) A special interface that is intended only to be used for a specialized task such as diagnosis, modification, monitoring, repairing, tailoring, or tuning. (2) A special interface that is dependent on or requires the customer to understand significant aspects of the detailed design and implementation of the IBM software product.

profile. RPC: An entry in a name service database that contains a collection of elements from which name service interface (NSI) search operations construct search paths for the database. Each search path is composed of one or more elements that refer to name service entries corresponding to a given RPC interface and, optionally, to an object. See NSI profile attribute and profile element.

profile element. RPC: A record in an RPC profile that maps an RPC interface identifier to a profile member (a server entry, group, or profile in a name service database). See profile. See also group, interface identifier and server entry.

profile member. RPC: A name service entry whose name occupies the member field of an element of the profile. See profile.

programming interface. The supported method through which customer programs request software services. The programming interface consists of a set of callable services provided with the product.

protection level. The degree to which secure network communications are protected. Synonymous with authentication level.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication.

protocol sequence. Synonym for RPC protocol sequence.

protocol sequence vector. RPC: A data structure that contains an array-size count and an array of pointers to RPC protocol-sequence strings. See RPC protocol sequence.

public object. (1) XOM: An object that is represented by a data structure whose format is part of the service’s specification. (2) XDS: A descriptor list that contains all the OM attributes of an OM object.

R

RACF. Resource Access Control Facility.

RDN. Relative distinguished name.

read access. CDS: An access right that grants the ability to view data.

read-only replica. (1) CDS: A copy of a CDS directory in which applications cannot make changes. Although applications can look up information (read) from it, they cannot create, change, or delete entries in a read-only replica. Read-only replicas become consistent with other, changeable replicas of the same directory during skulks and routine propagation of updates. (2) Security: A replicated Registry server.

realm. Security: A cell, considered exclusively from the point of view of Security; this term is used in Kerberos specifications. The term cell designates the basic unit of DCE configuration and administration and incorporates the notion of a realm.

reference monitor. Code that controls access to an object. In the DCE, servers control access to the objects they maintain; for a given object, the ACL manager associated with that object makes authorization decisions concerning the object.

reference pointer. RPC: A nonnull pointer whose value is invariant during a remote procedure call and cannot point at aliased storage.

referral. GDS: An outcome that can be returned by a DSA that cannot perform an operation itself. The referral identifies one or more other DSAs more able to perform the operation.

register. (1) RPC: To list an RPC interface with the RPC runtime. (2) To place server-addressing information into the local endpoint map. (3) To insert authorization and authentication information into binding information. See endpoint map and RPC interface.

Registry database. Security: A database of security information about principals, groups, organizations, accounts, and security policies.

Registry Service. Security: One of three services provided by the Security Service; the Registry Service manages information about principals, accounts, and security policies. The other services are the Privilege Service and the Authentication Service.

relative distinguished name (RDN). GDS, XDS: A set of Attribute Value Assertions (AVAs).

relative time. A discrete time interval that is usually added to or subtracted from an absolute time. See absolute time.
remote. Pertaining to a device, file or system that is accessed by your system through a communications line. Contrast with local.

remote procedure. RPC: An application procedure located in a separate address space from calling code. See remote procedure call.

remote procedure call. RPC: A client request to a service provider located anywhere in the network.

Remote Procedure Call (RPC). A DCE component. It allows requests from a client program to access a procedure located anywhere in the network.

replica. CDS: A directory in the CDS namespace. The first instance of a directory in the name space is the master replica. See master replica and read-only replica.

replication. The making of a shadow of a database to be used by another node. Replication can improve availability and load-sharing.

request. A command sent to a server over a connection.

resource. Items such as printers, plotters, data storage, or computer services. Each has a unique identifier associated with it for naming purposes.

Resource Access Control Facility (RACF). An IBM licensed program, that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, and logging the detected unauthorized access to protected resources.

return value. A function result that is returned in addition to the values of any output or input/output arguments.

ROM. Read-only memory.

RPC. Remote Procedure Call.

RPC control program (RPCCP). An interactive administrative facility for managing name service entries and endpoint maps for RPC applications.

RPCCP. RPC control program

RPC interface. A logical group of operations, data types, and constant declarations that serves as a network contract for a client to request a procedure in a server. See also interface definition and operation.

RPC protocol. An RPC-specific communications protocol that supports the semantics of the DCE RPC API and runs over either connectionless or connection-oriented communications protocols.

RPC protocol sequence. A valid combination of communications protocols represented by a character string. Each RPC protocol sequence typically includes three protocols: a network protocol, a transport protocol, and an RPC protocol that works with the network and transport protocols. See network protocol, RPC protocol, and transfer protocol. Synonymous with protocol sequence.

RPC runtime. A set of operations that manages communications, provides access to the name service database, and performs other tasks, such as managing servers and accessing security information, for RPC applications. See RPC runtime library.

RPC runtime library. A group of routines of the RPC runtime that support the RPC applications on a system. The runtime library provides a public interface to application programmers, the application programming interface (API), and a private interface to stubs, the stub programming interface (SPI). See RPC runtime.

RPC thread. A logical thread within which a remote procedure call is executed. See thread.

scalability. The ability of a distributed system to expand in size without changes to the system structure, applications, or the way users deal with the system.

schema. See directory schema.


Security Service. A DCE component that provides trustworthy identification of users, secure communications, and controlled access to resources in a distributed system.

segment. One or more contiguous elements of a string.

server. (1) On a network, the computer that contains programs, data, or provides the facilities that other computers on the network can access. (2) The party that receives remote procedure calls. Contrast with client.

server addressing information. RPC: An RPC protocol sequence, network address, and endpoint that represent one way to access an RPC server over a network; a part of server binding information. See network address. See also binding information, endpoint, and RPC protocol sequence.

server application thread. RPC: A thread running the server application code that initializes the server and listens for incoming calls. See application thread, client application thread, local application thread, and RPC thread.

server binding information. RPC: Binding information for a particular RPC server. See binding information and client binding information.

server entry. RPC: A name service entry that stores the binding information associated with the RPC interfaces of a particular RPC server and object Universal Unique Identifiers (UUIDs) for any objects offered by the server. See also binding information, NSI binding attribute, NSI object attribute, object and RPC interface.
server instance. RPC: A server running in a specific address space. See server.

server stub. RPC: The surrogate calling code for an RPC interface that is linked with server application code containing one or more sets of remote procedures (managers) that implement the interface. See client stub. See also manager.

service. In network architecture, the capabilities that the layers closer to the physical media provide to the layers closer to the end user.

service ticket. Security: A ticket for a specified service other than the ticket-granting service. See privilege ticket, simple ticket, and ticket-granting ticket.

session. GDS: A sequence of directory operations requested by a particular user of a particular directory user agent (DUA) using the same session object management (OM) object.

session key. Security: A short-lived encryption key provided by the Authentication Service to two principals for the purpose of ensuring secure communications between them. Synonymous with conversation key.

SID. Subject identifier.

signal. Threads: To wake only one thread waiting on a condition variable. See broadcast.

signed. Security: Pertaining to information that is appended to an enciphered summary of the information. This information is used to ensure the integrity of the data, the authenticity of the originator, and the unambiguous relationship between the originator and the data.

simple name. CDS: One element in a CDS full name. Simple names are separated by slashes in the full name.

simple ticket. Security: A ticket that contains the principal’s identity, a session key, a timestamp and other information, sealed using the target’s secret key. See privilege ticket, service ticket, and ticket-granting ticket.

socket. A unique host identifier created by the concatenation of a port identifier with a TCP/IP address.

specific. XOM: The attribute types that can appear in an instance of a given class, but not in an instance of its superclasses.

SPI. (1) System programming interface. (2) Stub programming interface.

SRT. Structure rule table.

standard. A model that is established and widely used.

string. An ordered sequence of bits, octets, or characters, accompanied by the string’s length.

structure rule table (SRT). GDS: A recurring attribute of the directory schema with the description of the permitted structures of distinguished names.

stub. RPC: A code module specific to an RPC interface that is generated by the Interface Definition Language (IDL) compiler to support remote procedure calls for the interface. RPC stubs are linked with client and server applications and hide the intricacies of remote procedure calls from the application code. See client stub and server stub.

Stub programming interface (SPI). RPC: A private runtime interface whose routines are unavailable to application code.

subclass. GDS, XOM: One of the classes whose attribute types are a superset of those of another class.

subject identifier (SID). A string that identifies a user or set of users. Each SID consists of three fields in the form person.group.organization. In an account, each field must have a specific value; in an access control list (ACL) entry, one or more fields may use a wildcard.

subobject. XOM: An object that is in a subordinate relationship to a given object.

subordinate. GDS, XDS: In the directory information tree (DIT), an entry whose distinguished name includes that of the other as a prefix.

superior. XDS: In the directory information tree (DIT), an entry whose distinguished name is included as a prefix of the distinguished name of the other. Each entry has exactly one immediate superior.

synchronization. DTS: The process by which a Distributed Time Service entity requests clock values from other systems, computes a new time from the values, and adjusts its system clock to the new time.

syntax. (1) XOM: An object management (OM) syntax is any of the various categories into which the OM specification statically groups values on the basis of their form. These categories are additional to the OM type of the value. (2) A category into which an attribute value is placed on the basis of its form. See attribute syntax.

System programming interface (SPI). A private interface reserved for use by other services within a system and not available to application code. Contrast with API.

system time. The time value maintained and used by the operating system.
TCP. Transmission Control Protocol

TCP/IP. Transmission Control Protocol/Internet Protocol

TDF. Time differential factor.

thread. A single sequential flow of control within a process.

thread handle. RPC: A data item that enables threads to share a storage management environment.

Threads Service. A DCE component that provides portable facilities that support concurrent programming. The threads service includes operations to create and control multiple threads of execution in a single process and to synchronize access to global data within an application.

ticket. Security: An application-transparent mechanism that transmits the identity of an initiating principal to its target. See privilege ticket, service ticket, simple ticket and ticket-granting ticket.

ticket-granting ticket. Security: A ticket to the ticket-granting service. See privilege ticket, service ticket, and simple ticket.

time differential factor (TDF). DTS: The difference between universal time coordinated (UTC) and the time in a particular time zone.

time provider (TP). DTS: A process that queries universal time coordinated (UTC) from a hardware device and provides it to the server.

time provider interface (TPI). An interface between the DTS server and external time provider process. The DTS server uses the interface to communicate with the time provider and to obtain timestamps from an external time source.

time provider program. DTS: An application that functions as a time provider.

tower. CDS: A set of physical address and protocol information for a particular server. CDS uses this information to locate the system on which a server resides and to determine which protocols are available at the server. Tower values are contained in the CDS_Towers attribute associated with the object entry that represents the server in the cell namespace.

transfer syntax. RPC: A set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations. See also Network Data Representation.

Transmission Control Protocol (TCP). A communications protocol used in Internet and any other network following the U.S. Department of Defense standards for inter-network protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in an interconnected system of such networks. It assumes that the Internet Protocol is the underlying protocol. The protocol that provides a reliable, full-duplex, connection-oriented service for applications.


transport layer. A network service that provides end-to-end communications between two parties, while hiding the details of the communications network. The Transmission Control Protocol (TCP) and International Organization for Standardization (ISO) TP4 transport protocols provide full-duplex virtual circuits on which delivery is reliable, error free, sequenced, and duplicate free. User Datagram Protocol (UDP) provides no guarantees. The connectionless RPC protocol provides some guarantees on top of UDP.

transport protocol. A communications protocol, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

type. XOM: A category into which attribute values are placed on the basis of their purpose. See attribute type.

type UUID. RPC: The Universal Unique Identifier (UUID) that identifies a particular type of object and an associated manager. See also manager and object.

UDP. User Datagram Protocol.

unexport. RPC: To remove binding information from a server entry in a name service database. Contrast with export.

Universal Time Coordinated (UTC). The basis of standard time throughout the world. Synonymous with Greenwich mean time (GMT).

Universal Unique Identifier (UUID). RPC: An identifier that is immutable and unique across time and space. A UUID can uniquely identify an entity such as an object or an RPC interface. See interface UUID, object UUID, and type UUID.

unmarshalling. RPC: The process by which a stub disassembles incoming network data and converts it into local data in the appropriate local data representation. Contrast with marshalling.

update timestamp (UTS). CDS: An attribute that identifies the time at which the most recent change was made to any attribute of a particular CDS name. For directories, the UTS reflects changes made only to attributes that apply to the actual directory (not one of its replicas).
user. A person who requires the services of a computing system.

User Datagram Protocol (UDP). In TCP/IP, a packet-level protocol built directly on the Internet protocol layer. UDP is used for application-to-application programs between TCP/IP host systems.

UTC. Universal Time Coordinated

UTS. Update timestamp.

UUID. Universal unique identifier

V

value. XOM: An arbitrary and complex information item that can be viewed as a characteristic or property of an object. See attribute value.

varying array. RPC: An array in which part of its contents is transmitted during a remote procedure call.

vector. RPC: An array of references to other structures.

vendor. Supplier of software products.

W

well-known endpoint. RPC: A preassigned, stable endpoint that a server can use every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol. An application declares a well-known endpoint either as an attribute in an RPC interface header or as a variable in the server application code. See dynamic endpoint and endpoint.

work space. XDS, XOM: A space in which OM objects of certain OM classes can be created, together with an implementation of the object management functions that supports those OM classes.

X

X.500. The CCITT/ISO standard for the open systems interconnection (OSI) application-layer directory. It allows users to register, store, search, and retrieve information about any objects or resources in a network or distributed system.

XDS. The X/Open Directory Services API.

X/Open Directory Services (XDS). An application program interface that DCE uses to access its directory service components. XDS provides facilities for adding, deleting, and looking up names and their attributes. The XDS library detects the format of the name to be looked up and directs the calls it receives to either GDS or CDS. XDS uses the XOM API to define and manage its information.

XOM. The X/Open OSI-Abstract-Data Manipulation API.
Bibliography

This bibliography is a list of publications for z/OS DCE and other products. The complete title, order number, and a brief description is given for each publication.

z/OS DCE Publications

This section lists and provides a brief description of each publication in the z/OS DCE library.

Overview

- **z/OS DCE Introduction** GC24-5911
  This book introduces z/OS DCE. Whether you are a system manager, technical planner, z/OS system programmer, or application programmer, it will help you understand DCE and evaluate the uses and benefits of including z/OS DCE as part of your information processing environment.

Planning

- **z/OS DCE Planning** GC24-5913
  This book helps you plan for the organization and installation of z/OS DCE. It discusses the benefits of distributed computing in general and describes how to develop plans for a distributed system in a z/OS environment.

Administration

- **z/OS DCE Configuring and Getting Started** SC24-5910
  This book helps system and network administrators configure z/OS DCE.
- **z/OS DCE Administration Guide** SC24-5904
  This book helps system and network administrators understand z/OS DCE and tells how to administer it from the batch, TSO, and shell environments.
- **z/OS DCE Command Reference** SC24-5909
  This book provides reference information for the commands that system and network administrators use to work with z/OS DCE.
- **z/OS DCE User’s Guide** SC24-5914
  This book describes how to use z/OS DCE to work with your user account, use the directory service, work with namespaces, and change access to objects that you own.

Application Development

- **z/OS DCE Application Development Guide: Introduction and Style** SC24-5907
  This book assists you in designing, writing, compiling, linking, and running distributed applications in z/OS DCE.
- **z/OS DCE Application Development Guide: Core Components** SC24-5905
  This book assists programmers in developing applications using application facilities, threads, remote procedure calls, distributed time service, and security service.
- **z/OS DCE Application Development Guide: Directory Services** SC24-5906
  This book describes the z/OS DCE directory service and assists programmers in developing applications for the cell directory service and the global directory service.
- **z/OS DCE Application Development Reference** SC24-5908
  This book explains the DCE Application Program Interfaces (APIs) that you can use to write distributed applications on z/OS DCE.

Reference

- **z/OS DCE Messages and Codes** SC24-5912
  This book provides detailed explanations and recovery actions for the messages, status codes, and exception codes issued by z/OS DCE.

z/OS SecureWay® Security Server Publications

This section lists and provides a brief description of books in the z/OS SecureWay Security Server library that may be needed for z/OS SecureWay Security Server DCE and for RACF® interoperability.

- **z/OS SecureWay Security Server DCE Overview** GC24-5921
  This book describes the z/OS SecureWay Security Server DCE and provides z/OS SecureWay Security Server DCE information about the z/OS DCE library.
• **z/OS SecureWay Security Server LDAP Client Programming**, SC24-5924
  This book describes the Lightweight Directory Access Protocol (LDAP) client APIs that you can use to write distributed applications on z/OS DCE and gives you information on how to develop LDAP applications.

• **z/OS SecureWay Security Server LDAP Server Administration and Use**, SC24-5923
  This book describes how to install, configure, and run the LDAP server. It is intended for administrators who will maintain the server and database.

  This book explains RACF concepts and describes how to plan for and implement RACF.

• **z/OS SecureWay Security Server LDAP Server Administration and Use**, SC24-5923
  This book describes how to install, configure, and run the LDAP server. It is intended for administrators who will maintain the server and database.

• **z/OS SecureWay Security Server Firewall Technologies**, SC24-5922
  This book provides the configuration, commands, messages, examples and problem determination for the z/OS Firewall Technologies. It is intended for network or system security administrators who install, administer and use the z/OS Firewall Technologies.

---

**Tool Control Language Publication**

• *Tcl and the Tk Toolkit*, John K. Osterhout, (c)1994, Addison—Wesley Publishing Company.
  This non-IBM book on the Tool Control Language is useful for application developers, DCECP script writers, and end users.

**IBM C/C++ Language Publication**

• **z/OS C/C++ Programming Guide**, SC09-4765
  This book describes how to develop applications in the C/C++ language in z/OS.

**z/OS DCE Application Support Publications**

This section lists and provides a brief description of each publication in the z/OS DCE Application Support library.

• **z/OS DCE Application Support Configuration and Administration Guide**, SC24-5903
  This book helps system and network administrators understand and administer Application Support.

• **z/OS DCE Application Support Programming Guide**, SC24-5902
  This book provides information on using Application Support to develop applications that can access CICS® and IMS™ transactions.
Encina Publications

- **z/OS Encina Toolkit Executive Guide and Reference**  
  SC24-5919  
  This book discusses writing Encina applications for z/OS.

- **z/OS Encina Transactional RPC Support for IMS**  
  SC24-5920  
  This book is to help software designers and programmers extend their IMS transaction applications to participate in a distributed, transactional client/server application.
Index

A
absolute time, definition 5-1
absolute value 5-4
Access Control List (ACL)
  access
testing for object 6-159
testing on behalf of process 6-161
accessing 6-145
adding permissions
for ACL entry 6-16
for any_other entry 6-13
for foreign user 6-14
for group 6-15
for object 6-17
for unauthorized entry 6-18
for user 6-19
adding permissions for entry 6-16
binding handle
  getting by network address 6-190
  getting for access list 6-189
  releasing 6-204
constants 6-166, 6-187
copying 6-3
data types 6-164
File Group Class, setting permissions mask 6-191
freeing space 6-20
getting error information 6-194
initializing 6-21
manager
  listing 6-195, 6-197
  registering object type of 6-22
manager, writing 6-145
object
  getting 6-155
  testing access to 6-159
  testing on behalf of process 6-161
object permissions
  listing 6-201
object protection, listing types 6-147, 6-149
of file system 6-189
of root directory 6-189
print strings
  getting, internal 6-151
printable print strings
  listing 6-199
replacing
  with information in sec_acl_list structure 6-157
retrieving types of schema objects 6-400
retrieving UUID from item in backing store 6-4
storage for access list
  releasing 6-203
strings for an object
  obtaining 6-389

Access Control List (ACL) (continued)
testing access on behalf of another process 6-209
testing access to directory object 6-207
update site, referral to 6-153
UUID
  finding by name 6-25
  finding by object UUID 6-27
access to directory object, ACL testing 6-207
access to object
testing 6-159
testing on behalf of process 6-161
access, determining principal 6-8
account data, registry database, listing 6-362
account information, registry API, getting 6-426
account, login
  changing name details 6-366
  defining name 6-354
  deleting from registry 6-358
  replacing all data 6-368
acct.h  A-3
acct.idl  A-8
ACL
  See Access Control List (ACL)
ACL permissions 2-19
aclbase.h  A-3
aclbase.idl  A-8
administrative registry account, updating 6-356
alias, cell
  freeing 8-29
  listing 8-30
APIs
  supported
cs_byte_from_netcs 2-23
cs_byte_local_size 2-26
cs_byte_net_size 2-29
cs_byte_to_netcs 2-32
dce_acl_copy_acl 6-3
dce_acl_inq_acl_from_header 6-4
dce_acl_inq_client_creds 6-5
dce_acl_inq_client_permset 6-6
dce_acl_inq_permset_for_creds 6-8
dce_acl_inq_prin_and_group. 6-10
dce_acl_is_client_authorized 6-12
dce_acl_obj_add_any_other_entry 6-13
dce_acl_obj_add_foreign_entry 6-14
dce_acl_obj_add_group_entry 6-15
dce_acl_obj_add_id_entry 6-16
dce_acl_obj_add_obj_entry 6-17
dce_acl_obj_add_unauth_entry 6-18
dce_acl_obj_add_user_entry 6-19
dce_acl_obj_free_entries 6-20
dce_acl_obj_init 6-21
dce_acl_register_object_type 6-22
dce_acl_resolve_by_name 6-25
APIs (continued)
supported (continued)

dce_acl_resolve_by_uuid 6-27

dce_attr_sch_bind 8-3

dce_attr_sch_bind_free 8-5

dce_attr_sch_create_entry 8-6

dce_attr_sch_cursor_alloc 8-7

dce_attr_sch_cursor_init 8-8

dce_attr_sch_cursor_release 8-10

dce_attr_sch_cursor_reset 8-11

dce_attr_sch_delete_entry 8-12

dce_attr_sch_get_acl_mgrs 8-13

dce_attr_sch_lookup_by_id 8-15

dce_attr_sch_lookup_by_name 8-16

dce_attr_sch_scan 8-17

dce_attr_sch_update_entry 8-19

dce_aud_close 6-34

dce_aud_commit 6-35

dce_aud_discard 6-38

dce_aud_free_ev_info 6-39

dce_aud_free_header 6-40

dce_aud_get_ev_info 6-41

dce_aud_get_header 6-42

dce_aud_length 6-43, 6-45

dce_aud_next 6-45

dce_aud_open 6-48

dce_aud_print 6-51

dce_aud_set_trail_size_limit 6-54

dce_aud_start 6-56

dce_aud_start_with_name 6-59

dce_aud_start_with_pac 6-62

dce_aud_start_with_server_binding 6-65

dce_aud_start_with_uuid 6-68

dce_cf_binding_entry_from_host 8-23

dce_cf_binding_from_rpc_binding 8-25

dce_cf_find_name_by_key 8-27

dce_cf_free_cell_aliases 8-29

dce_cf_get_cell_aliases 8-30

dce_cf_get_cell_name 8-31

dce_cf_get_csrpy_filename 8-33

dce_cf_get_host_name 8-35

dce_cf_prin_name_from_host 8-37

dce_cf_profile_entry_from_host 8-39

dce_cf_same_cell_name 8-41

dce_cs_loc_to_rgy 2-35

dce_cs_rgy_to_loc 2-37

dce_db_close 8-46

dce_db_delete 8-47

dce_db_delete_by_name 8-48

dce_db_delete_by_uuid 8-49

dce_db_fetch 8-50

dce_db_fetch_by_name 8-52

dce_db_fetch_by_uuid 8-54

dce_db_free 8-56

dce_db_header_fetch 8-57

dce_db_inq_count 8-58

dce_db_iter_done 8-59

dce_db_iter_next 8-60

dce_db_iter_next_by_name 8-61

dce_db_iter_next_by_uuid 8-62

dce_db_iter_start 8-63

dce_db_lock 8-64

dce_db_open 8-65

dce_db_std_header_init 8-68

dce_db_store 8-70

dce_db_store_by_name 8-71

dce_db_store_by_uuid 8-73

dce_db_unlock 8-75

dce_error_inq_text 8-77

dce_server_disable_service 8-80

dce_server_enable_service 8-81

dce_server_inq_attr 8-82

dce_server_inq_server 8-83

dce_server_inq_uuids 8-84

dce_server_register 8-85

dce_server_sec_begin 8-87

dce_server_sec_done 8-88

dce_server_unregister 8-89

dce_server_use_protseq 8-90

dced_binding_create 8-97

dced_binding_free 8-99

dced_binding_from_rpc_binding 8-100

dced_binding_set_auth_info 8-102

dced_entry_add 8-104

dced_entry_get_next 8-106

dced_entry_remove 8-108

dced_hostdata_create 8-110

dced_hostdata_delete 8-112

dced_hostdata_read 8-114

dced_hostdata_write 8-116

dced_initialize_cursor 8-118

dced_inq_id 8-120

dced_inq_name 8-122

dced_keytab_add_key 8-124

dced_keytab_change_key 8-126

dced_keytab_create 8-128

dced_keytab_delete 8-130

dced_keytab_get_next_key 8-132

dced_keytab_initialize_cursor 8-133

dced_keytab_release_cursor 8-135

dced_keytab_remove_key 8-136

dced_list_get 8-138

dced_list_release 8-140

dced_object_read 8-141

dced_object_read_all 8-144

dced_objects_release 8-146

dced_release_cursor 8-148

dced_secval_start 8-149

dced_secval_status 8-151

dced_secval_stop 8-152

dced_secval_validate 8-154
**APIs (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dced_server_create</td>
<td>8-156</td>
</tr>
<tr>
<td>dced_server_delete</td>
<td>8-158</td>
</tr>
<tr>
<td>dced_server_disable_if</td>
<td>8-160</td>
</tr>
<tr>
<td>dced_server_enable_if</td>
<td>8-162</td>
</tr>
<tr>
<td>dced_server_modify_attributes</td>
<td>8-164</td>
</tr>
<tr>
<td>dced_server_start</td>
<td>8-166</td>
</tr>
<tr>
<td>dced_server_stop</td>
<td>8-168</td>
</tr>
<tr>
<td>ds_add_entry</td>
<td>4-4</td>
</tr>
<tr>
<td>ds_bind</td>
<td>4-6</td>
</tr>
<tr>
<td>ds_compare</td>
<td>4-8</td>
</tr>
<tr>
<td>ds_initialize</td>
<td>4-10</td>
</tr>
<tr>
<td>ds_list</td>
<td>4-11</td>
</tr>
<tr>
<td>ds_modify_entry</td>
<td>4-13</td>
</tr>
<tr>
<td>ds_read</td>
<td>4-15</td>
</tr>
<tr>
<td>ds_remove_entry</td>
<td>4-17</td>
</tr>
<tr>
<td>ds_search</td>
<td>4-19</td>
</tr>
<tr>
<td>ds_shutdown</td>
<td>4-21</td>
</tr>
<tr>
<td>ds_unbind</td>
<td>4-22</td>
</tr>
<tr>
<td>ds_version</td>
<td>4-24</td>
</tr>
<tr>
<td>dsX_extract_attr_values</td>
<td>4-26</td>
</tr>
<tr>
<td>exc_first_catch</td>
<td>3-13</td>
</tr>
<tr>
<td>exc_get_reason_code</td>
<td>3-14</td>
</tr>
<tr>
<td>exc_get_status</td>
<td>3-15</td>
</tr>
<tr>
<td>exc_matches</td>
<td>3-16</td>
</tr>
<tr>
<td>exc_raise_status</td>
<td>3-17</td>
</tr>
<tr>
<td>exc_set_status</td>
<td>3-18</td>
</tr>
<tr>
<td>gss_accept_sec_context</td>
<td>6-80</td>
</tr>
<tr>
<td>gss_acquire_credential</td>
<td>6-84</td>
</tr>
<tr>
<td>gss_add_credential</td>
<td>6-86</td>
</tr>
<tr>
<td>gss_add_oid_set_member</td>
<td>6-88</td>
</tr>
<tr>
<td>gss_canonicalize_name</td>
<td>6-89</td>
</tr>
<tr>
<td>gss_compare_name</td>
<td>6-90</td>
</tr>
<tr>
<td>gss_context_time</td>
<td>6-91</td>
</tr>
<tr>
<td>gss_create_empty_oid_set</td>
<td>6-92</td>
</tr>
<tr>
<td>gss_delete_sec_context</td>
<td>6-93</td>
</tr>
<tr>
<td>gss_display_name</td>
<td>6-94</td>
</tr>
<tr>
<td>gss_display_status</td>
<td>6-95</td>
</tr>
<tr>
<td>gss_duplicate_name</td>
<td>6-97</td>
</tr>
<tr>
<td>gss_get_mic</td>
<td>6-98</td>
</tr>
<tr>
<td>gss_import_name</td>
<td>6-100</td>
</tr>
<tr>
<td>gss_indicate_mechs</td>
<td>6-102</td>
</tr>
<tr>
<td>gss_init_sec_context</td>
<td>6-103</td>
</tr>
<tr>
<td>gss_inquire_context</td>
<td>6-107</td>
</tr>
<tr>
<td>gss_inquire_credential</td>
<td>6-109</td>
</tr>
<tr>
<td>gss_inquire_credential_by_mech</td>
<td>6-111</td>
</tr>
<tr>
<td>gss_inquire_mechs_for_name</td>
<td>6-113</td>
</tr>
<tr>
<td>gss_inquire_names_for_mech</td>
<td>6-114</td>
</tr>
<tr>
<td>gss_krb5_get_ccache</td>
<td>6-142</td>
</tr>
<tr>
<td>gss_krb5_get_tkt_flags</td>
<td>6-143</td>
</tr>
<tr>
<td>gss_oid_to_str</td>
<td>6-115</td>
</tr>
<tr>
<td>gss_process_context_token</td>
<td>6-116</td>
</tr>
<tr>
<td>gss_release_buffer</td>
<td>6-117</td>
</tr>
<tr>
<td>gss_release_credential</td>
<td>6-118</td>
</tr>
<tr>
<td>gss_release_name</td>
<td>6-119</td>
</tr>
<tr>
<td>gss_release_oid</td>
<td>6-120</td>
</tr>
<tr>
<td>gss_release_oid_set_member</td>
<td>6-121</td>
</tr>
<tr>
<td>gss_release_oid_set</td>
<td>6-121</td>
</tr>
<tr>
<td>gss_seal</td>
<td>6-128</td>
</tr>
<tr>
<td>gss_sign</td>
<td>6-98</td>
</tr>
<tr>
<td>gss_str_to_oid</td>
<td>6-122</td>
</tr>
<tr>
<td>gss_test_oid_set_member</td>
<td>6-123</td>
</tr>
<tr>
<td>gss_unseal</td>
<td>6-124</td>
</tr>
<tr>
<td>gss_unwrap</td>
<td>6-124</td>
</tr>
<tr>
<td>gss_verify</td>
<td>6-126</td>
</tr>
<tr>
<td>gss_verify_mic</td>
<td>6-126</td>
</tr>
<tr>
<td>gss_wrap</td>
<td>6-128</td>
</tr>
<tr>
<td>gss_wrap_size_limit</td>
<td>6-130</td>
</tr>
<tr>
<td>gssdce_add_oid_set_member</td>
<td>6-88</td>
</tr>
<tr>
<td>gssdce_create_empty_oid_set</td>
<td>6-92</td>
</tr>
<tr>
<td>gssdce_cred_to_login_context</td>
<td>6-132</td>
</tr>
<tr>
<td>gssdce_export_credential</td>
<td>6-133</td>
</tr>
<tr>
<td>gssdce_extract_creds_from_sec_context</td>
<td>6-134</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_credential</td>
<td>6-135</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_sec_context</td>
<td>6-136</td>
</tr>
<tr>
<td>gssdce_import_credential</td>
<td>6-137</td>
</tr>
<tr>
<td>gssdce_login_context_to_credential</td>
<td>6-138</td>
</tr>
<tr>
<td>gssdce_register_acceptor_identity</td>
<td>6-139</td>
</tr>
<tr>
<td>gssdce_set_cred_context_to_oid</td>
<td>6-140</td>
</tr>
<tr>
<td>gssdce_test_oid_set_member</td>
<td>6-123</td>
</tr>
<tr>
<td>idl_es_decode_buffer</td>
<td>2-39</td>
</tr>
<tr>
<td>idl_es_decode_incremental</td>
<td>2-41</td>
</tr>
<tr>
<td>idl_es_encode_dyn_buffer</td>
<td>2-43</td>
</tr>
<tr>
<td>idl_es_encode_fixed_buffer</td>
<td>2-45</td>
</tr>
<tr>
<td>idl_es_encode_incremental</td>
<td>2-47</td>
</tr>
<tr>
<td>idl_es_handle_free</td>
<td>2-49</td>
</tr>
<tr>
<td>idl_es_inq_atts</td>
<td>2-50</td>
</tr>
<tr>
<td>idl_es_inq_encoding_id</td>
<td>2-52</td>
</tr>
<tr>
<td>idl_es_set_atts</td>
<td>2-54</td>
</tr>
<tr>
<td>idl_es_set_transfer_syntax</td>
<td>2-56</td>
</tr>
<tr>
<td>krb5_address_compare</td>
<td>7-10</td>
</tr>
<tr>
<td>krb5_address_search</td>
<td>7-11</td>
</tr>
<tr>
<td>krb5_auth_con_free</td>
<td>7-12</td>
</tr>
<tr>
<td>krb5_auth_con_genaddr</td>
<td>7-13</td>
</tr>
<tr>
<td>krb5_auth_con_getaddr</td>
<td>7-14</td>
</tr>
<tr>
<td>krb5_auth_con_getauthenticator</td>
<td>7-15</td>
</tr>
<tr>
<td>krb5_auth_con_getflags</td>
<td>7-16</td>
</tr>
<tr>
<td>krb5_auth_con_getivector</td>
<td>7-17</td>
</tr>
<tr>
<td>krb5_auth_con_getkey</td>
<td>7-18</td>
</tr>
<tr>
<td>krb5_auth_con_getlocalsequence_number</td>
<td>7-19</td>
</tr>
<tr>
<td>krb5_auth_con_getlocalsubkey</td>
<td>7-20</td>
</tr>
<tr>
<td>krb5_auth_con_getports</td>
<td>7-24</td>
</tr>
<tr>
<td>krb5_auth_con_getcache</td>
<td>7-21</td>
</tr>
<tr>
<td>krb5_auth_con_getremotesequencenumber</td>
<td>7-22</td>
</tr>
<tr>
<td>krb5_auth_con_getremotesubkey</td>
<td>7-23</td>
</tr>
<tr>
<td>krb5_auth_con_init</td>
<td>7-25</td>
</tr>
<tr>
<td>krb5_auth_con_initvector</td>
<td>7-26</td>
</tr>
<tr>
<td>krb5_auth_con_set_key</td>
<td>7-27</td>
</tr>
<tr>
<td>krb5_auth_con_set_safe_cryptotype</td>
<td>7-28</td>
</tr>
<tr>
<td>krb5_auth_con_set_subkey</td>
<td>7-29</td>
</tr>
<tr>
<td>krb5_auth_con_setflags</td>
<td>7-30</td>
</tr>
<tr>
<td>krb5_auth_con_setivector</td>
<td>7-31</td>
</tr>
</tbody>
</table>
APIs (continued)
supported (continued)
krb5_auth_con_setports 7-32
krb5_auth_con_setrcache 7-33
krb5_auth_con_setuserkey 7-34
krb5_auth_to_rep 7-35
krb5_build_principal 7-36
krb5_build_principal_ext 7-37
krb5_build_principal_ext_va 7-38
krb5_build_principal_va 7-40
krb5_cc_close 7-42
krb5_cc_default 7-43
krb5_cc_default_name 7-44
krb5_cc_destroy 7-45
krb5_cc_end_seq_get 7-46
krb5_cc_generate_new 7-47
krb5_cc_get_name 7-48
krb5_cc_get_principal 7-49
krb5_cc_get_type 7-50
krb5_cc_initialize 7-51
krb5_cc_next_cred 7-52
krb5_cc_register 7-53
krb5_cc_remove_cred 7-54
krb5_cc_resolve 7-56
krb5_cc_retrieve_creds 7-57
krb5_cc_set_flags 7-59
krb5_cc_start_seq_get 7-60
krb5_cc_store_cred 7-61
krb5_copy_address 7-62
krb5_copy_addresses 7-63
krb5_copy_authdata 7-64
krb5_copy_authenticator 7-65
krb5_copy_checksum 7-66
krb5_copy_creds 7-67
krb5_copy_data 7-68
krb5_copy_keyblock 7-69
krb5_copy_keyblock_contents 7-70
krb5_copy_principal 7-71
krb5_copy_ticket 7-72
krb5_free_address 7-73
krb5_free_addresses 7-74
krb5_free_ap_rep_enc_part 7-75
krb5_free_authdata 7-76
krb5_free_authenticator 7-77
krb5_free_authenticator_contents 7-78
krb5_free_checksum 7-79
krb5_free_context 7-80
krb5_free_creds_contents 7-81
krb5_free_creds 7-82
krb5_free_data 7-83
krb5_free_enc_tkt_part 7-84
krb5_free_error 7-85
krb5_free_host Realm 7-86
krb5_free_kdc_rep 7-87
krb5_free_keyblock 7-88
krb5_free_keyblock_contents 7-89
krb5_free_krbhst 7-90
krb5_free_principal 7-91
krb5_free_tgt_creds 7-92
krb5_free_ticket 7-93
krb5_free_tickets 7-94
krb5_gen_replay_name 7-95
krb5_generate_seq_number 7-96
krb5_generate_subkey 7-97
krb5_get_cred_from_kdc 7-98
krb5_get_cred_from_kdc_renew 7-99
krb5_get_cred_from_kdc_validate 7-100
krb5_get_cred_via_tkt 7-101
krb5_get_credentials 7-102
krb5_get_credentials_renew 7-103
krb5_get_credentials_validate 7-104
krb5_get_default_in_tkt_ktypes 7-105
krb5_get_default_realm 7-106
krb5_get_default_tgs_ktypes 7-107
krb5_get_host_realm 7-108
krb5_get_in_tkt_with_keytab 7-109
krb5_get_in_tkt_with_password 7-111
krb5_get_in_tkt_with_skey 7-113
krb5_get_krbhst 7-115
krb5_get_server_ccache 7-116
krb5_init_context 7-117
krb5_kt_add_entry 7-118
krb5_kt_close 7-119
krb5_kt_default 7-120
krb5_kt_default_name 7-121
krb5_kt_end_seq_get 7-122
krb5_kt_free_entry 7-123
krb5_kt_get_entry 7-124
krb5_kt_get_name 7-125
krb5_kt_get_type 7-126
krb5_kt_next_entry 7-127
krb5_kt_read_service_key 7-128
krb5_kt_register 7-129
krb5_kt_remove_entry 7-130
krb5_kt_resolve 7-131
krb5_kt_start_seq_get 7-132
krb5_md4_crypto_compat_ctl 7-133
krb5_md5_crypto_compat_ctl 7-134
krb5_mk_error 7-135
krb5_mk_priv 7-136
krb5_mk_rep 7-137
krb5_mk_req 7-138
krb5_mk_req_extended 7-140
krb5_mkr_safe 7-141
krb5_os_hostaddr 7-142
krb5_os_localaddr 7-143
krb5_parse_name 7-144
krb5_principal_compare 7-145
krb5_random_confounder 7-146
krb5_rc_close 7-147
krb5_rc_default 7-148
krb5_rc_default_name 7-149
### APIs (continued)

**supported (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>krb5_rc_destroy</td>
<td>7-150</td>
</tr>
<tr>
<td>krb5_rc_expgne</td>
<td>7-151</td>
</tr>
<tr>
<td>krb5_rc_free_entry_contents</td>
<td>7-152</td>
</tr>
<tr>
<td>krb5_rc_get_lifespan</td>
<td>7-153</td>
</tr>
<tr>
<td>krb5_rc_get_name</td>
<td>7-154</td>
</tr>
<tr>
<td>krb5_rc_get_type</td>
<td>7-155</td>
</tr>
<tr>
<td>krb5_rc_initialize</td>
<td>7-156</td>
</tr>
<tr>
<td>krb5_rc_recover</td>
<td>7-157</td>
</tr>
<tr>
<td>krb5_rc_register_type</td>
<td>7-158</td>
</tr>
<tr>
<td>krb5_rc_resolve</td>
<td>7-159</td>
</tr>
<tr>
<td>krb5_rc_store</td>
<td>7-160</td>
</tr>
<tr>
<td>krb5_rd_error</td>
<td>7-161</td>
</tr>
<tr>
<td>krb5_rd_priv</td>
<td>7-162</td>
</tr>
<tr>
<td>krb5_rd_rep</td>
<td>7-164</td>
</tr>
<tr>
<td>krb5_rd_req</td>
<td>7-165</td>
</tr>
<tr>
<td>krb5_rd_safe</td>
<td>7-167</td>
</tr>
<tr>
<td>krb5_realm_compare</td>
<td>7-168</td>
</tr>
<tr>
<td>krb5_set_config_files</td>
<td>7-169</td>
</tr>
<tr>
<td>krb5_set_default_in_tkt_ktypes</td>
<td>7-170</td>
</tr>
<tr>
<td>krb5_set_default_realm</td>
<td>7-171</td>
</tr>
<tr>
<td>krb5_set_default_tgs_ktypes</td>
<td>7-172</td>
</tr>
<tr>
<td>krb5_sname_to_principal</td>
<td>7-173</td>
</tr>
<tr>
<td>krb5_svc_get_level</td>
<td>7-174</td>
</tr>
<tr>
<td>krb5_svc_get_msg</td>
<td>7-175</td>
</tr>
<tr>
<td>krb5_timeofday</td>
<td>7-176</td>
</tr>
<tr>
<td>krb5_unparse_name</td>
<td>7-177</td>
</tr>
<tr>
<td>krb5_unparse_name_ext</td>
<td>7-178</td>
</tr>
<tr>
<td>krb5_us_timeofday</td>
<td>7-179</td>
</tr>
<tr>
<td>om_copy</td>
<td>4-64</td>
</tr>
<tr>
<td>om_copy_value</td>
<td>4-66</td>
</tr>
<tr>
<td>om_create</td>
<td>4-68</td>
</tr>
<tr>
<td>om_delete</td>
<td>4-70</td>
</tr>
<tr>
<td>om_get</td>
<td>4-72</td>
</tr>
<tr>
<td>om_instance</td>
<td>4-75</td>
</tr>
<tr>
<td>om_put</td>
<td>4-77</td>
</tr>
<tr>
<td>om_read</td>
<td>4-80</td>
</tr>
<tr>
<td>om_remove</td>
<td>4-82</td>
</tr>
<tr>
<td>om_write</td>
<td>4-84</td>
</tr>
<tr>
<td>omX_extract</td>
<td>4-86 , 4-89 , 4-90 , 4-91 , 4-93</td>
</tr>
<tr>
<td>pthread_attr_create</td>
<td>3-20</td>
</tr>
<tr>
<td>pthread_attr_delete</td>
<td>3-21</td>
</tr>
<tr>
<td>pthread_attr_getstacksize</td>
<td>3-22</td>
</tr>
<tr>
<td>pthread_attr_setstacksize</td>
<td>3-23</td>
</tr>
<tr>
<td>pthread_cancel</td>
<td>3-24</td>
</tr>
<tr>
<td>pthread_cleanup_pop</td>
<td>3-25</td>
</tr>
<tr>
<td>pthread_cleanup_push</td>
<td>3-26</td>
</tr>
<tr>
<td>pthread_cond_broadcast</td>
<td>3-27</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>3-28</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>3-29</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>3-31</td>
</tr>
<tr>
<td>pthread_cond_timedwait</td>
<td>3-32</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>3-34</td>
</tr>
<tr>
<td>pthread_condattr_create</td>
<td>3-36</td>
</tr>
<tr>
<td>pthread_condattr_delete</td>
<td>3-37</td>
</tr>
<tr>
<td>pthread_create</td>
<td>3-38</td>
</tr>
</tbody>
</table>

### Index

- pthread_delay_np 3-40
- pthread_detach 3-41
- pthread_equal 3-42
- pthread_exit 3-43
- pthread_getExpiration_np 3-44
- pthread_getspecific 3-45
- pthread_join 3-46
- pthread_keycreate 3-47
- pthread_lock_global_np 3-49
- pthread_mutex_destroy 3-50
- pthread_mutex_init 3-51
- pthread_mutex_lock 3-52
- pthread_mutex_trylock 3-54
- pthread_mutex_unlock 3-55
- pthread_mutexattr_create 3-56
- pthread_mutexattr_delete 3-57
- pthread_mutexattr_getkind_np 3-58
- pthread_mutexattr_setkind_np 3-59
- pthread_once 3-61
- pthread_self 3-62
- pthread_setsynccancel 3-63
- pthread_setscancel 3-65
- pthread_setspecific 3-67
- pthread_signal_to_cancel_np 3-68
- pthread_testcancel 3-69
- pthread_unlock_global 3-70
- pthread_yield 3-71
- rdac1_get_access 6-145
- rdac1_get_manager_types 6-147
- rdac1_get_manager_types_semantics 6-149
- rdac1_get_printstring 6-151
- rdac1_get_referral 6-153
- rdac1_lookup 6-155
- rdac1_replace 6-157
- rdac1_test_access 6-159
- rdac1_test_access_on_behalf 6-161
- rpc_binding_copy 2-58
- rpc_binding_free 2-60
- rpc_binding_from_string_binding 2-62
- rpc_binding_inq_auth_caller 2-64
- rpc_binding_inq_auth_client 2-67
- rpc_binding_inq_auth_info 2-70
- rpc_binding_inq_object 2-73
- rpc_binding_reset 2-74
- rpc_binding_server_from_client 2-76
- rpc_binding_set_auth_info 2-79
- rpc_binding_set_object 2-83
- rpc_binding_to_string_binding 2-84
- rpc_binding_vector_free 2-86
- rpc_cs_binding_set_tags 2-86, 2-87, 2-147
- rpc_cs_char_set_compact_check 2-89, 2-147
- rpc_cs_eval_with_universal 2-91, 2-147
- rpc_cs_eval_without_universal 2-93, 2-147
- rpc_cs_get_tags 2-95, 2-147
- rpc_ep_register 2-98
APIs (continued)
supported (continued)
rpc_ep_register_no_replace 2-105
rpc_ep_register_no_replace_wlb 2-108
rpc_ep_register_wlb 2-101
rpc_ep_resolve_binding 2-111
rpc_ep_unregister 2-114
rpc_get_ae_name 2-151
rpc_if_id_vector_free 2-116
rpc_if_inq_id 2-117
rpc_mgmt_ep_elt_inq_begin 2-118
rpc_mgmt_ep_elt_inq_done 2-121
rpc_mgmt_ep_elt_inq_next 2-122
rpc_mgmt_ep_elt_inq_next_wlb 2-124
rpc_mgmt_ep_set_activated_wlb 2-126
rpc_mgmt_ep_unregister 2-128
rpc_mgmt_inq_com_timeout 2-130
rpc_mgmt_inq_dflt_protect_level 2-132
rpc_mgmt_inq_if_ids 2-134
rpc_mgmt_inq_server_princ_name 2-136
rpc_mgmt_inq_stats 2-138
rpc_mgmt_is_server_listening 2-140
rpc_mgmt_set_authorization_fn 2-142
rpc_mgmt_set_cancel_timeout 2-144
rpc_mgmt_set_com_timeout 2-146
rpc_mgmt_stats_vector_free 2-148
rpc_mgmt_stop_server_listening 2-149
rpc_network_inq_protseqs 2-155
rpc_network_is_protseq_valid 2-156
rpc_ns_binding_export 2-158
rpc_ns_binding_import_begin 2-161
rpc_ns_binding_import_done 2-163
rpc_ns_binding_import_next 2-165
rpc_ns_binding_inq_entry_name 2-167
rpc_ns_binding_lookup_begin 2-169
rpc_ns_binding_lookup_done 2-171
rpc_ns_binding_lookup_next 2-173
rpc_ns_binding_select 2-176
rpc_ns_binding_unexport 2-178
rpc_ns_entry_expand_name 2-180
rpc_ns_entry_inq_resolution 2-182
rpc_ns_entry_object_inq_begin 2-184
rpc_ns_entry_object_inq_done 2-186
rpc_ns_entry_object_inq_next 2-188
rpc_ns_group_delete 2-190
rpc_ns_group_mbr_add 2-192
rpc_ns_group_mbr_inq_begin 2-194
rpc_ns_group_mbr_inq_done 2-196
rpc_ns_group_mbr_inq_next 2-198
rpc_ns_group_mbr_remove 2-200
rpc_ns_import_cbx_add_eval 2-202
rpc_ns_mgmt_binding_unexport 2-204
rpc_ns_mgmt_entry_create 2-207
rpc_ns_mgmt_entry_delete 2-209
rpc_ns_mgmt_entry_inq_if_ids 2-211
rpc_ns_mgmt_free_codesets 2-213
rpc_ns_mgmt_handle_set_exp_age 2-215
rpc_ns_mgmt_inq_com_timeout 2-217
rpc_ns_mgmt_inq_exp_age 2-217
rpc_ns_mgmt_read_codesets 2-219
rpc_ns_mgmt_remove_attribute 2-221
rpc_ns_mgmt_set_attribute 2-223
rpc_ns_mgmt_set_exp_age 2-225
rpc_ns_profile_delete 2-227
rpc_ns_profile_elt_add 2-229
rpc_ns_profile_elt_inq_begin 2-231
rpc_ns_profile_elt_inq_done 2-234
rpc_ns_profile_elt_next 2-236
rpc_ns_profile_elt_remove 2-238
rpc_object_inq_type 2-240
rpc_object_inq_fn 2-242
rpc_object_set_type 2-244
rpc_protseq_vector_free 2-246
rpc_rgy_get_codesets 2-247
rpc_rgy_get_max_bytes 2-249
rpc_server_inq_bindings 2-251
rpc_server_inq_if 2-253
rpc_server_unregister_if 2-255
rpc_server_register_auth_info 2-258
rpc_server_register_if 2-260
rpc_server_unregister_if 2-264
rpc_server_use_all_protseqs 2-266
rpc_server_use_all_protseqs_if 2-268
rpc_server_use_protseq 2-270
rpc_server_use_protseq_ep 2-272
rpc_server_use_protseq_if 2-274
rpc_set_ae_name 2-153
rpc_sm_allocate 2-276
rpc_sm_client_free 2-278
rpc_sm_destroy_client_context 2-279
rpc_sm_disable_allocate 2-280
rpc_sm_enable_allocate 2-281
rpc_sm_free 2-282
rpc_sm_get_thread_handle 2-283
rpc_sm_set_client_alloc_free 2-284
rpc_sm_set_thread_handle 2-286
rpc_sm_swap_client_alloc_free 2-288
rpc_ss_allocate 2-290
rpc_ss_bind_authn_client 2-292
rpc_ss_client_free 2-294
rpc_ss_destroy_client_context 2-295
rpc_ss_disable_allocate 2-296
rpc_ss_enable_allocate 2-297
rpc_ss_free 2-298
rpc_ss_get_thread_handle 2-299
rpc_ss_get_client_alloc_free 2-301
rpc_ss_set_thread_handle 2-303
rpc_string_binding_compose 2-307
rpc_string_binding_parse 2-309
rpc_tower_to_binding 2-311
rpc_tower_vector_free 2-313
APIs (continued)
supported (continued)

tactics
rpc_tower_vector_from_binding 2-314
sec_acl_bind 6-189
sec_acl_bind_to_addr 6-190
sec_acl_calc_mask 6-191
sec_acl_get_access 6-192
sec_acl_get_error_info 6-194
sec_acl_get_manager_types 6-195
sec_acl_get_mgr_types_semantics 6-197
sec_acl_get_printstring 6-199
sec_acl_lookup 6-201
sec_acl_release 6-203
sec_acl_release_handle 6-204
sec_acl_replace 6-205
sec_acl_test_access 6-207
sec_acl_test_access_on_behalf 6-209
sec_attr_trig_query 6-211
sec_attr_trig_update 6-214
sec_cred_free_attr_cursor 6-216
sec_cred_free_cursor 6-217
sec_cred_free_pa_handle 6-218
sec_cred_get_authz_session_info 6-219
sec_cred_get_client_princ_name 6-220
sec_cred_get_deleg_restrictions 6-221
sec_cred_get_delegation_type 6-222
sec_cred_get_delegation_type 6-224
sec_cred_get_extended_attrs 6-225
sec_cred_get_initiator 6-227
sec_cred_get_opt_restrictions 6-228
sec_cred_get_pa_data 6-229
sec_cred_get_req_restrictions 6-230
sec_cred_get_tgt_restrictions 6-231
sec_cred_get_v1_pac 6-232
sec_cred_initialize_attr_cursor 6-233
sec_cred_initialize_cursor 6-234
sec_cred_is_authenticated 6-235
sec_id_gen_group 6-237
sec_id_gen_name 6-239
sec_id_global_gen_group 6-241
sec_id_global_gen_name 6-243
sec_id_global_is_group_member 6-245
sec_id_global_parse_group 6-247
sec_id_global_parse_name 6-249
sec_id_is_group_member 6-251
sec_id_parse_group 6-253
sec_id_parse_name 6-255
sec_key_mgmt_change_key 6-258
sec_key_mgmt_delete_key 6-261
sec_key_mgmt_delete_key_type 6-263
sec_key_mgmt_free_key 6-265
sec_key_mgmt_get_next_key 6-270
sec_key_mgmt_get_next_kvno 6-274
sec_key_mgmt_initialize_cursor 6-276
	nex
sec_key_mgmt_manage_key 6-278
sec_key_mgmt_release_cursor 6-280
sec_key_mgmt_set_key 6-281
sec_login_become_delegate 6-284
sec_login_become_impersonator 6-287
sec_login_become_initiator 6-289
sec_login_certify_identity 6-291
sec_login_context_token_owner 6-293
sec_login_create_context_token 6-294
sec_login_cred_get_delegate 6-296
sec_login_cred_get_initiator 6-298
sec_login_cred_init_cursor 6-299
sec_login_delete_context_token 6-300
sec_login_disable_delegation 6-301
sec_login_export_context 6-304
sec_login_export_context_data 6-306
sec_login_free_net_info 6-308
sec_login_get_current_context 6-309
sec_login_get_expiration 6-311
sec_login_get_groups 6-313
sec_login_get_pwent 6-314
sec_login_import_context 6-315
sec_login_import_context_data 6-316
sec_login_init_first 6-314
sec_login_inquire_net_info 6-318
sec_login_newgroups 6-320
sec_login_purge_context 6-322
sec_login_refresh_identity 6-324
sec_login_release_context 6-326
sec_login_set_context 6-327
sec_login_set_extended_attrs 6-329
sec_login_setup_first 6-331
sec_login_setup_identity 6-333
sec_login_valid_and_cert_ident 6-335
sec_login_valid_from_keytable 6-338
sec_login_valid_from_system 6-341
sec_login_validate_first 6-343
sec_login_validate_identity 6-345
sec_pwd_mgmt_free_handle 6-348
sec_pwd_mgmt_gen_pwd 6-349
sec_pwd_mgmt_get_val_type 6-350
sec_pwd_mgmt_setup 6-351
sec_pwd_mgmt_setup 6-351
sec_rgy_acct_add 6-354
sec_rgy_acct_admin_replace 6-356
sec_rgy_acct_delete 6-358
sec_rgy_acct_get_project 6-360
sec_rgy_acct_get_project 6-360
sec_rgy_acct_lookup 6-362
sec_rgy_acct_passwd 6-364
sec_rgy_acct_rename 6-366
sec_rgy_acct_replace_all 6-368
sec_rgy_acct_user_replace 6-371
sec_rgy_attr_cursor_alloc 6-373
sec_rgy_attr_cursor_init 6-374
sec_rgy_attr_cursor_release 6-376
APIs (continued)
supported (continued)

- `sec_rgy_attr_cursor_reset` 6-377
- `sec_rgy_attr_delete` 6-378
- `sec_rgy_attr_get_effective` 6-380
- `sec_rgy_attr_lookup_by_id` 6-382
- `sec_rgy_attr_lookup_by_name` 6-385
- `sec_rgy_attr_lookup_no_expand` 6-387
- `sec_rgy_attr_sch_aclmgr_strings` 6-389
- `sec_rgy_attr_sch_create_entry` 6-391
- `sec_rgy_attr_sch_cursor_alloc` 6-393
- `sec_rgy_attr_sch_cursor_init` 6-394
- `sec_rgy_attr_sch_cursor_release` 6-396
- `sec_rgy_attr_sch_cursor_reset` 6-397
- `sec_rgy_attr_sch_delete_entry` 6-398
- `sec_rgy_attr_sch_get_acl_mgrs` 6-399
- `sec_rgy_attr_sch_lookup_by_id` 6-400
- `sec_rgy_attr_sch_lookup_by_name` 6-404
- `sec_rgy_attr_sch_scan` 6-405
- `sec_rgy_attr_sch_update_entry` 6-407
- `sec_rgy_attr_update` 6-409
- `sec_rgy_auth_plcy_get_effective` 6-410
- `sec_rgy_auth_plcy_get_info` 6-412
- `sec_rgy_auth_plcy_set_info` 6-414
- `sec_rgy_cell_bind` 6-415
- `sec_rgy_cursor_reset` 6-416
- `sec_rgy_login_get_effective` 6-417
- `sec_rgy_login_get_info` 6-419
- `sec_rgy_pgo_add` 6-420
- `sec_rgy_pgo_add_member` 6-422
- `sec_rgy_pgo_delete` 6-424
- `sec_rgy_pgo_delete_member` 6-426
- `sec_rgy_pgo_get_by_eff_unix_num` 6-428
- `sec_rgy_pgo_get_by_id` 6-429
- `sec_rgy_pgo_get_by_name` 6-431
- `sec_rgy_pgo_get_by_unix_num` 6-433
- `sec_rgy_pgo_get_effective` 6-434
- `sec_rgy_pgo_get_effective_unix_num` 6-436
- `sec_rgy_pgo_get_effective_unix_num_to_id` 6-438
- `sec_rgy_pgo_get_effective_unix_num_to_name` 6-440
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-442
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-444
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-446
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-448
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-450
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-452
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-454
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-456
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-458
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-460
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-462
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-464
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-466
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-468
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-470
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-472
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-474
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-476
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-478
- `sec_rgy_pgo_get_effective_unix_num_to_unix_num` 6-480
- `sec_rgy_site_close` 6-482
- `sec_rgy_site_open` 6-483
- `sec_rgy_site_is_readonly` 6-485
- `sec_rgy_site_open` 6-486
- `sec_rgy_site_open_query` 6-487
- `sec_rgy_site_open_update` 6-489
- `sec_rgy_unix_getgrgid` 6-491
- `sec_rgy_unix_getgrnam` 6-493
- `sec_rgy_unix_getpwnam` 6-495
- `sec_rgy_unix_getpwuid` 6-497
- `sec_rgy_wait_until_consistent` 6-499
- `sigwait` 3-72
- `utc_abstime` 5-4
- `utc_addtime` 5-6
- `utc_anytime` 5-8
- `utc_anyzone` 5-10
- `utc_ascanaytime` 5-12
- `utc_ascgmttime` 5-14
- `utc_asclocaltime` 5-16
- `utc_ascrreltime` 5-18
- `utc_binreltime` 5-19
- `utc_bintime` 5-21
- `utc_boundtime` 5-22
- `utc_cmpintervaltime` 5-24
- `utc_cmptime` 5-27
- `utc_gettime` 5-30
- `utc_getusertime` 5-31
- `utc_mtime` 5-32
- `utc_gmtime` 5-34
- `utc_localtime` 5-37
- `utc_localzone` 5-39
- `utc_mkanytime` 5-41
- `utc_mkascrreltime` 5-44
- `utc_mkascrreltime` 5-46
- `utc_mksdifftime` 5-48
- `utc_mksdifftime` 5-49
- `utc_mksdifftime` 5-51
- `utc_mksdifftime` 5-53
- `utc_mksdifftime` 5-55
- `utc_mksdifftime` 5-57
- `utc_mksdifftime` 5-59
- `utc_mksdifftime` 5-61
- `utc_mksdifftime` 5-62
- `utc_mksdifftime` 5-64
- `utc_mksdifftime` 5-66
- `uuid_compare` 2-315
- `uuid_create` 2-317
- `uuid_create` 2-318
- `uuid_equal` 2-319
- `uuid_from_string` 2-320
- `uuid_hash` 2-321
- `uuid_is_nil` 2-322
- `uuid_to_string` 2-323

Unsupported
list of 1-1

---

X-30 Application Development Reference Volumes 1 and 2
arbitrary time zone 5-12
asynchronous thread cancelation 3-63
Attribute Interface, DCE
data types 8-2
overview 8-2
routines 8-3
attribute object, thread
changing stacksize 3-23
creating 3-20
creating on condition variable 3-36
deleting 3-21
deleting on condition variable 3-37
getting stacksize 3-22
mutex, creating 3-56
mutex, deleting 3-57
attribute of group, deleting 2-190
attribute values, comparing (directory) 4-8
attribute values, extracting from OM object 4-26
Attribute, Privilege, Certificate
See Extended Privilege Attribute Certificate (EPAC)
attributes, security
obtaining for a delegation intermediary 6-222
passing to trigger server 6-214
reading 6-211
effective 6-380
expanded sets 6-382
object 6-385
object (without expansion) 6-387
attributes, security, routine parameter
allocating resources
cursor 6-373
deleting
cursor 6-378
initializing
cursor 6-374
re-initializing
cursor 6-377
releasing
cursor 6-376
audit
APIs 6-29
event information types 6-30
interface, DCE, overview 6-29
Audit APIs 6-34—6-68
audit event
starting 6-56
starting with name 6-59
starting with PAC 6-62
starting with server binding 6-65
starting with UUID 6-68
audit interface, DCE 6-29
authenticated identity lifetime, getting 6-313
authentication
accessing services 6-163
credential, security 6-235
identity, refreshing 6-324
authentication (continued)
obtaining for registry account 6-417
package, strong 4-47
policy
obtaining for registry account 6-417
registry account 6-419
registry organization, current 6-464
registry organization, getting 6-466
registry organization, setting 6-468
authentication for registry account, obtaining 6-417
authentication information, registering 2-258
authorization
accessing services 6-163
authentication information, registering 2-258
authentication protection level, setting 2-132
checking
client 6-12
controlling remote access 2-142
obtaining information from client 2-67
obtaining information from handle for client 2-64
obtaining information from server 2-70
server principal name, getting 2-136
setting for server binding handle 2-79

B
backing store
closing 8-46
deleting an item 8-47
header
retrieving 8-57
header, initializing 8-68
items
obtaining number 8-58
opening or creating 8-65
releasing data 8-56
retrieving an item 8-50
storing data in 8-69, 8-71, 8-73
string-indexed
deleting an item 8-48
retrieving data 8-52
unlocking 8-75
UUID-indexed
deleting an item 8-49
retrieving data 8-54

Basic Directory Contents Package 4-35
bibliography X-19
binary time
comparing
two 5-24, 5-27
converting
from GMT or UTC 5-51
from local 5-53
from null-terminated absolute time text 5-46
from relative 5-44
from timespec 5-48, 5-49
to GMT 5-14
binding handle (continued)
- partially bound
  - nil object UUID 2-112
  - non-nil object UUID 2-112
- registering with name service database 2-158
- releasing ACL 6-204
- removing
  - from endpoint map 2-128
  - from name service database entry 2-178
  - multiple 2-204
- replacing object UUID in server handle 2-83
- resetting for server 2-74
- returning compatible server 2-176
- returning object UUID 2-73
- routines for obtaining 2-20
- RPC runtime routine commands, listing 2-9
- server authorization, obtaining 2-70
- server handles, list of 2-251
- server, copying 2-58
- setting authorization for server 2-79
- setting timeout value 2-146
- storage, freeing 2-60, 2-86

binding to network identity, login context 6-311
binding vector, RPC 2-10
binding.h A-3
binding.idl A-8
books, list of DCE and related X-19
bounds, event time 5-22
buffer size, calculating for code set conversion 2-26

C
- C program listing of ACL manager types 6-195
- caller ACL permissions for object, listing 6-192
- calling process, testing ACL access 6-207
- calling thread, terminating 3-43
- cancelation, thread
  - actions taken upon 3-38
  - asynchronous 3-63
  - delivering request 3-69
  - destructor 3-39
  - general 3-65
  - signal mask method 3-68
  - terminating execution 3-24
- CATCH 3-9
- CATCH macro 3-9
- CATCH_ALL 3-10
- CATCH_ALL macro 3-10
- CDS
  - See Cell Directory Service (CDS)
- Cell Directory Service (CDS)
  - comparison to Global Directory Services (GDS) 4-2
  - namespace 4-2
  - xds definitions file 4-39
- cell registry, server handle, binding to 6-421
cell UUID, generating global name from 6-237, 6-239, 6-241, 6-243
cell UUID, translating global name into 6-247, 6-249, 6-253, 6-255
Certificate, Extended Privilege Attribute (EPAC)
   contents 6-161
   obtaining from privilege attribute handle 6-232
   reading 6-145
Certificate, Privilege Attribute (PAC)
   See Extended Privilege Attribute Certificate (EPAC)
character data, converting code sets 2-23
character string, freeing allocated storage 2-311
cleanup, thread
   handler, establishing 3-26
   handler, running 3-25
   stack 3-26
client
   authorization
      checking 6-12
client access, permitting 2-142
client account name (PGO), adding 6-428
client authorization, checking 6-12
client communication attempt, time limits 2-130
client credentials, obtaining 6-5
client permissions, obtaining 6-6
client resources, reclaiming 2-279
client stub
   exchange storage allocation with client caller 2-305
   freeing storage 2-278
   memory management, overriding defaults 2-301
   storage used by, releasing 2-294
cobolbas.h A-3
code set
   conversion 2-26
code set registry
   obtaining pathname 8-33
code sets
   converting character data 2-23
codesets_stub.h A-3
codesets.h A-3
codesets.idl A-8
codestub.h A-3
communication with server, getting handle 2-251
component strings, creating from binding handle 2-309
condition variable, thread
   creating 3-29
   creating attributes object on 3-36
   deleting 3-28
   deleting attributes object on 3-37
   waiting 3-34
   waiting for signal 3-32
   waiting threads, waking all 3-27
   waking all waiting threads 3-27
   waking one thread 3-31
condition variable, threads, definition 3-29
configuration APIs 8-46
configuration file 8-21
configuration routines
   cell alias
      freeing 8-29
      listing 8-30
      closing a backing store 8-46
      code set registry
         obtaining pathname 8-33
         deleting an item from backing store 8-47
         deleting an item from UUID-indexed backing store 8-49
         deleting item from string-indexed backing store 8-48
         freeing iteration state 8-59
         host name
            obtaining 8-35
            host principal name
               obtaining 8-37
               host profile name
                  obtaining 8-39
                  initializing backing store header 8-68
                  locking backing store 8-64
                  number of items in a backing store 8-58
                  opening or creating backing store 8-65
                  output 8-22
                  overview 8-21
                  preparing backing store for iteration 8-63
                  releasing data from a string-indexed backing store 8-56
                  retrieving an item from a backing store 8-50
                  retrieving data from a UUID-indexed backing store 8-54
                  retrieving data from string-indexed backing store 8-52
                  retrieving header from a backing store 8-57
                  returning dced entry name on a host 8-25
                  returning host binding entry name 8-23
                  returning next key during 8-60, 8-61, 8-62
                  server
                     disabling authenticated RPC 8-88
                     enabling authenticated RPC 8-87
                     obtaining attributes 8-82
                     obtaining configuration and execution UUIDs 8-84
                     obtaining server configuration data 8-83
                     registering with DCE 8-85
                     specifying RPC protocol sequences 8-90
                     unregistering with DCE 8-89
service
   disabling for server 8-80
   enabling for server 8-81
   storing data in backing store 8-69, 8-71, 8-73
   string key match 8-27
   unlocking backing store 8-75
constants
   acl 6-166
   registry 6-187
context handle, resetting to null 2-295
context, import
   removing name service 2-163
context, security (GSSAPI)  
See Generic Security Service Application Programming Interface (GSSAPI), context

coordinated universal time (UTC)  
definition 5-1
interval  

deriving from two binary timestamps 5-64

credentials, security  
authentication of, checking 6-12
cursor, initializing 6-234
extracting EPAC from 6-135, 6-136
freeing attribute cursor 6-216
freeing cursor 6-217
freeing privilege attribute handle 6-218
GSSAPI, acquiring 6-84
obtaining 6-5
obtaining client 6-219
obtaining principal name from 6-220

cs_byte_from_netcs API 2-23
converting  
data character code sets 2-23

cs_byte_local_size API 2-26
calculating buffer size  

code set conversion 2-26

cs_byte_net_size API 2-29
calculating buffer size  

code set conversion 2-29

cs_byte_to_netcs API 2-32
converting  
data character code sets 2-32

cs_mgmt.h A-3

cs_mgmt.idl A-8

current exception, catching 3-13

current registry account login data, getting 6-424
cursor  
locating in local key store 6-276
registry database, resetting 6-422
releasing initialized storage 6-280
security, initializing 6-233

d
daclif.h A-3
daclif.idl A-8
daclmgr.h A-3
daclmgr.idl A-8
data types  
acl 6-164
extended privilege attribute 6-167
key management 6-170
login 6-171
registry 6-172
data types and structures, RPC 2-9
database properties, registry  
freeing registry server binding handle 6-482
getting 6-470

database properties, registry (continued)  
setting 6-472
DCE Attribute Interface  
data types 8-2
overview 8-2
routines 8-3
DCE Attribute Interface Routines 8-3—8-21
DCE audit API 6-29
DCE Host Daemon APIs 8-97—A-1
DCE Name Service  
code sets array, freeing 2-213
code sets, getting supported 2-247
code sets, maximum bytes used for encoding 2-249
code sets, reading for server 2-219
database, removing import context 2-163
database, retrieving compatible binding handle 2-165
environment variables 2-8

group attribute, removing member 2-200
import context, adding evaluation routine 2-202
permissions 2-19
rpc server entry, adding attribute 2-223
rpc server entry, removing attribute 2-221
DCE Security Service  
See Security Service, DCE
DCE system time, retrieving 5-30, 5-31
dce_acl_copy_acl API 6-3
dce_acl_inq_acl_from_header API 6-4
dce_acl_inq_client_creds API 6-5
dce_acl_inq_client_permset API 6-6
dce_acl_inq_permset_for_creds API 6-8
dce_acl_inq_prin_and_group API 6-10
dce_is_client_authorized API 6-12
dce_obj_add_any_other_group API 6-13
dce_obj_add_foreign_entry API 6-14
dce_obj_add_group_entry API 6-15
dce_obj_add_id_entry API 6-16
dce_obj_add_object_entry API 6-17
dce_obj_add_unauth_entry API 6-18
dce_obj_add_user_entry API 6-19
dce_obj_free_entries API 6-20
dce_obj_init API 6-21
dce_register_object_type API 6-22
dce_attr_sch_resolve_by_name API 6-25
dce_attr_sch_resolve_by_uuid API 6-27
dce_attr_sch_bind API 8-3
dce_attr_sch_free API 8-5
dce_attr_sch_create_entry API 8-6
dce_attr_sch_create_entry API 8-7
dce_attr_sch_create_entry API 8-8
dce_attr_sch_release API 8-10
dce_attr_sch_release API 8-11
dce_attr_sch_delete_entry API 8-12
dce_attr_get_acls API 8-13
dce_attr_get_acls API 8-15
dce_attr_sch_lookup_by_id API 8-16

X-34 Application Development Reference Volumes 1 and 2
dce_attr_sch_scan API 8-17
  dce_attr_sch_update_entry API 8-19
  dce_aud_close API 6-34
  dce_aud_commit API 6-35
  dce_aud_discard API 6-38
  dce_aud_free_ev_info API 6-39
  dce_aud_free_header API 6-40
  dce_aud_get_ev_info API 6-41
  dce_aud_get_header API 6-42
  dce_aud_length API 6-43
  dce_aud_next API 6-45
  dce_aud_open API 6-48
  dce_aud_print API 6-51
  dce_aud_put_ev_info API 6-53
  dce_aud_set_trail_size_limit API 6-54
  dce_aud_start API 6-56
  dce_aud_start_with_name API 6-59
  dce_aud_start_with_pac API 6-62
  dce_aud_start_with_server_binding API 6-65
  dce_aud_start_with_uuid API 6-68
  dce_cf_binding_entry_from_host API 8-23
  dce_cf_const.h A-3
  dce_cf_const.idl A-8
  dce_cf_dced_entry_from_host API 8-25
  dce_cf_find_name_by_key API 8-27
  dce_cf_free_cell_aliases API 8-29
  dce_cf_get_cell_aliases API 8-30
  dce_cf_get_cell_name API 8-31
    cell name, local
    obtaining 8-31
  dce_cf_get_csrty_filename API 8-33
  dce_cf_get_host_name API 8-35
  dce_cf_prin_name_from_key API 8-37
  dce_cf_profile_entry_from_host API 8-39
  dce_cf_same_cell_name API 8-41
  dce_cf.h A-3
  dce_cs_loc_to_rgy API 2-35
    mapping
    code set names to values 2-35
  dce_cs_rgy_to_loc API 2-37
    mapping
    code set values to names 2-37
  dce_db_close API 8-46
  dce_db_delete API 8-47
  dce_db_delete_by_name API 8-48
  dce_db_delete_by_uuid API 8-49
  dce_db_fetch API 8-50
  dce_db_fetch_by_name API 8-52
  dce_db_fetch_by_uuid API 8-54
  dce_db_free API 8-56
  dce_db_header_fetch API 8-57
  dce_db_inq_count API 8-58
  dce_db_iter_done API 8-59
  dce_db_iter_next API 8-60
  dce_db_iter_next_by_name API 8-61
  dce_db_iter_next_by_uuid API 8-62
  dce_db_iter_start API 8-63
  dce_db_lock API 8-64
  dce_db_open API 8-65
  dce_db_std_header_init API 8-68
  dce_db_store API 8-69
  dce_db_store_by_name API 8-71
  dce_db_store_by_uuid API 8-73
  dce_db_unlock API 8-75
  dce_error_inq_text API 8-77
  dce_error.h A-4
  dce_server_disable_service API 8-80
  dce_server_enable_service API 8-81
  dce_server_inq_attr API 8-82
  dce_server_inq_server API 8-83
  dce_server_inq_uuids API 8-84
  dce_server_register API 8-85
  dce_server_sec_begin API 8-87
  dce_server_sec_done API 8-88
  dce_server_unregister API 8-89
  dce_server_use_protsq API 8-90
  dce_signal.h A-4
  DCE_THREADS constant 3-5
  dce_threads.h A-4
dced
  binding
    creating 8-97
    establishing 8-97
    from RPC binding 8-100
    releasing 8-99
    setting authorization and authentication information 8-102
    binding from RPC binding 8-100
    configuring 8-156
    deleting configuration 8-158
    disabling specific service 8-160
    enabling specific service 8-162
  entries, list
    initializing cursor to 8-118
  entry
    adding 8-104
    cursor to start of list 8-118
    listing 8-106
    name to UUID 8-120
    removing 8-108
    UUID to name 8-122
  hostdata item
    creating 8-110
    deleting 8-112
    reading 8-114
    replacing 8-116
  keytable
    adding keys to 8-124
    changing keys 8-126
    creating 8-128
    deleting 8-130
directory

definition of 4-2

tree

leaf entry, adding 4-4

object, private

copying 4-64

copying as public object 4-72

copying attribute value strings 4-66

copying attribute values 4-77

creating 4-68

deleting 4-70

deleting attribute values 4-82

determining class 4-75

extracting OM type 4-86

reading a string 4-80

writing a string 4-84

object, public

copying attribute values 4-77

creating new 4-72

deleting service-generated 4-70

determining class 4-75

session, opening 4-6

workspace

deleting 4-21

directory entry

searching 4-19

directory information tree (DIT), in CDS 4-2

directory object

ACL access for noncalling process 6-209

directory services, DCE

overview of interfaces to 4-1

packages

Basic Directory Contents Package 4-35

Cell Directory Services (CDS) 4-39

Directory Services Package 4-27

Global Directory Services (GDS) 4-41

Message Handling System (MHS) 4-9

Message Handling System (MHS) Directory User 4-44

MHS Message Store General Attributes 4-59
directory services, DCE (continued) packages (continued)  
  Strong Authentication 4-47  
  session, ending 4-22  
  terminology 4-2  
Directory System Agent (DSA) communication link 4-6  
Distributed Time Services (DTS)  
  See Time Services, Distributed (DTS)  
ds_add_entry API 4-4  
ds_bind API 4-6  
ds_compare API 4-8  
ds_initialize API 4-10  
ds_list API 4-11  
ds_modify_entry API 4-13  
ds_remove_entry API 4-17  
ds_read API 4-15  
ds_search API 4-19  
ds_shutdown API 4-21  
ds_unbind API 4-22  
ds_version API 4-24  
dsX_extract_attr_values API 4-26  
dtsprovider.acf A-10  
dtsprovider.h A-4  
dtsprovider.idl A-9  
entry (continued)  
  leaf  
    adding to directory 4-4  
    removing 4-17  
entry in name service database, deleting 2-209  
entry name, obtaining expanded version 2-180  
entry name, resolving 2-182  
entry point vector, manager  
    determining 2-253  
entry, group (NSI)  
    inquiry context  
      creating for viewing 2-194  
      removing 2-196  
    member name  
      adding 2-192  
      removing 2-200  
    viewing individually 2-198  
environment variables  
  Kerberos 7-9  
  RPC_DEFAULT_ENTRY 2-8  
  RPC_DEFAULT_ENTRY_SYNTAX 2-8  
EPA 6-167  
equal UUIDs, determining 2-319  
erro  
  variable, threads 3-5  
erro status for routine, determining 8-77  
errors  
  binding  
    listing, ACL handle, last one 6-194  
euvpdlws.h A-4  
event information  
  adding to audit record 6-53  
  freeing 6-39  
  obtaining 6-41  
event information types, audit 6-30  
event information, audit  
  adding to audit record 6-53  
  freeing 6-39  
  obtaining 6-41  
event time bounds 5-22  
event, audit  
  starting 6-56  
  starting with name 6-59  
  starting with PAC 6-62  
  starting with server binding 6-65  
  starting with UUID 6-68  
exc_first_catch 3-13  
exc_first_catch API 3-13  
exc_get_reason_code API 3-14  
exc_get_status 3-15  
exc_get_status API 3-15  
exc_handling.h A-4  
exc_matches 3-16  
exc_matches API 3-16  
exc_raise_status 3-17  
exc_raise_status API 3-17  
E  
effective login data, current account 6-424  
encoding  
  operation, IDL, getting flags from 2-50  
  operation, IDL, identifying within an interface 2-52  
  operation, IDL, setting flags 2-54  
  operation, IDL, specifying handle 2-56  
encrypting messages (GSSAPI) 6-128  
encryption signature (GSSAPI) 6-98  
endpoint address information registering 2-268  
endpoint map  
  element search parameters  
    removing 2-121  
    setting 2-118  
  server address information  
    activating 2-126  
    adding 2-101, 2-105, 2-108  
    changing 2-98  
    removing 2-114, 2-128  
    replacing 2-101  
  server binding handle endpoint, removing 2-74  
  viewing server endpoint elements 2-122, 2-124  
endpoint, specifying with protocol sequence 2-272  
entry  
dced  
  adding 8-104  
  cursor to start of list 8-118  
  listing 8-106  
  name to UUID 8-120  
  removing 8-108  
  UUID to name 8-122  

Index X-37
exc_set_status 3-18
exc_set_status API 3-18
EXCEPTION_INIT 3-6
EXCEPTION_INIT macro 3-6
exceptions, threads
  catching a particular 3-9
  catching all 3-10
  catching current 3-13
  clauses 3-12
  comparing two objects 3-16
  defining actions 3-12
  defining scope for 3-8
  enabling exception-returning interface 3-5
GOTO, nonlocal, use of 3-10
object, initializing 3-6
overview 3-5
raising current 3-11
raising status 3-17
reason code, retrieving 3-14
RETURN, use of 3-10
searching for 3-7
setting status value 3-18
status value, retrieving 3-15
status, creating 3-15
exchanging storage allocation with client caller 2-305
expiration age
  changing 2-225
  getting 2-217
  setting 2-215
expiration time, thread 3-44
expiration, key 6-278
Extended Privilege Attribute
data types 6-167
Extended Privilege Attribute Certificate (EPAC)
  contents 6-161
  obtaining from privilege attribute handle 6-232
  reading 6-145
Extended Registry Attributes (ERAs)
  constructing login context with 6-329

F
file
  closing 6-34
  limiting size 6-54
  opening 6-48
  writing to 6-35
File Group Class, ACL permissions, setting 6-191
file, audit
  closing 6-34
  limiting size 6-54
  opening 6-48
  writing to 6-35
FINALLY 3-12
FINALLY macro 3-12
foreign user, adding permissions to ACL for 6-14
formatting audit record 6-51
freeing
  space used by ACL entries 6-20

G
GDS
  See Global Directory Service (GDS)
general thread cancelation 3-65
generic security interface 6-71
Generic Security Service Application Programming Interface (GSSAPI)
  buffer
    releasing storage 6-117
  context
    accepting 6-80
    acceptor 6-71
    checking validity time 6-91
    deleting 6-93
    export 6-133
    extracting DCE credential from 6-134, 6-136
    import 6-137
    indicator 6-71
    initiating 6-103
    processing 6-116
    registering acceptor identity 6-139
    returning information about 6-107
  credential
    acquiring 6-84
    adding 6-86
    change ownership of DCE credential login context 6-140
    creating credential from DCE login context 6-138
    obtaining DCE login context associated with 6-132
    obtaining information about 6-109
    releasing local data structures 6-118
    returning information for security mechanism 6-111
    returning Kerberos 6-142
  data types 6-75
  DCE extensions 6-71
  displaying status codes 6-95
  duplicating internal name 6-97
  error handling 6-73
  internal name
    reducing to mechanism name 6-89
  Kerberos extensions 6-71
  mechanism availability
    determining 6-102
    returning 6-111, 6-113, 6-114
  messages
    determining largest 6-130
    encrypting 6-128
    encryption signature, generating 6-98
    encryption signature, verifying 6-126
    signing 6-128
    unwrapping 6-124
Generic Security Service Application Programming Interface (GSSAPI) (continued)

names
- comparing 6-90
- internal, converting printable name to 6-100
- internal, displaying 6-94
- internal, releasing storage for 6-119

object ID (OID)
- adding 6-88
- checking 6-123
- converting to 6-122
- converting to string representation 6-115
- creating new set of 6-92
- releasing storage 6-120, 6-121

overview 6-71

routines
- gss_accept_sec_context 6-80
- gss_acquire_cred 6-84
- gss_add_cred 6-86
- gss_add_oid_set_member 6-88
- gss_canonicalize_name 6-89
- gss_compare_name 6-90
- gss_context_time 6-91
- gss_create_empty_oid_set 6-92
- gss_delete_sec_context 6-93
- gss_display_name 6-94
- gss_display_status 6-95
- gss_duplicate_name 6-97
- gss_import_name 6-100
- gss_indicate_mechs 6-102
- gss_init_sec_context 6-103
- gss_inquire_context 6-107
- gss_inquire_cred 6-109
- gss_inquire_cred_by_mech 6-111
- gss_inquire_names_for mech 6-113
- gss_krb5_get_ccache 6-142
- gss_krb5_get_tkt_flags 6-143
- gss_oid_to_str 6-115
- gss_process_context_token 6-116
- gss_release_buffer 6-117
- gss_release_cred 6-118
- gss_release_name 6-119
- gss_release_oid 6-120
- gss_release_oid_set 6-121
- gss_seal 6-128
- gss_sign 6-98
- gss_str_to_oid 6-122
- gss_test_oid_set_member 6-123
- gss_unseal 6-124
- gss_unwrap 6-124
- gss_verify 6-126
- gss_verify_mic 6-126
- gss_wrap 6-128
- gss_wrap_size_limit 6-130
- gssdce_add_oid_set_member 6-88

Generic Security Service Application Programming Interface (GSSAPI) (continued)
routines (continued)
- gssdce_create_empty_oid_set 6-92
- gssdce_cred_to_login_context 6-132
- gssdce_export_cred 6-133
- gssdce_extract_creds_from_sec_context 6-134
- gssdce_extract_PAC_from_cred 6-135
- gssdce_extract_PAC_from_sec_context 6-136
- gssdce_import_cred 6-137
- gssdce_login_context_to_cred 6-138
- gssdce_register_acceptor_identity 6-139
- gssdce_set_cred_context_ownership 6-140
- gssdce_test_oid_set_member 6-123

Global Directory Service (GDS)
- comparison to Cell Directory Services (CDS) 4-2
- directory information tree 4-2
- xds definitions file 4-41

global ID
- from cell and group UUID 6-237, 6-241
- from cell and principal UUID 6-239, 6-243
- parsing into cell/group, names/UUIDs 6-247, 6-249, 6-253, 6-255
- translating into cell and principal UUID 6-247, 6-249, 6-253, 6-255

global name 2-13
- from cell and group UUID 6-237, 6-241
- from cell and principal UUID 6-239, 6-243
- parsing into cell/group, names/UUIDs 6-247, 6-249, 6-253, 6-255
- translating into cell and principal UUID 6-247, 6-249, 6-253, 6-255

glossary X-1

GMT 5-10
- Greenwich Mean Time (GMT)
  - converting
    - from binary 5-14
    - from tm structure 5-32
    - to binary timestamp 5-51
  - converting from binary to 5-14
  - getting timezone offset 5-10
  - setting 5-34
  - time zone label 5-34

group entry (NSI)
- inquiry context
  - creating for viewing 2-194
  - removing 2-196
- member name
  - adding 2-192
  - removing 2-200
  - viewing individually 2-198

group list, login context, changing 6-320
- group UUID, generating global name from 6-237, 6-241
- group, adding permissions for 6-15
GSSAPI (continued)
context (continued)
returning information about 6-107
credential
acquiring 6-84
adding 6-86
change ownership of DCE credential login context 6-140
creating credential from DCE login context 6-138
obtaining DCE login context associated with 6-132
obtaining information about 6-109
releasing local data structures 6-118
returning information for security mechanism 6-111
returning Kerberos 6-142
data types 6-75
DCE extensions 6-71
displaying status codes 6-95
duplicating internal name 6-97
error handling 6-73
internal name
  reducing to mechanism name 6-89
Kerberos extensions 6-71
mechanism availability
determining 6-102
returning 6-111, 6-113, 6-114
messages
determining largest 6-130
encrypting 6-128
encryption signature, generating 6-98
encryption signature, verifying 6-126
signing 6-128
unwrapping 6-124
names
  comparing 6-90
internal, converting printable name to 6-100
internal, displaying 6-94
internal, releasing storage for 6-119
object ID (OID)
adding 6-88
checking 6-123
converting to 6-122
converting to string representation 6-115
creating new set of 6-92
releasing storage 6-120, 6-121
overview 6-71
routines
  gss_accept_sec_context 6-80
  gss_acquire_cred 6-84
  gss_add_cred 6-86
  gss_add_oid_set_member 6-88
  gsscanonicalize_name 6-89
  gss_compare_name 6-90
  gss_context_time 6-91
  gss_create_empty_oid_set 6-92
  gss_delete_sec_context 6-93
  gss_display_name 6-94
  gss_display_status 6-95
  gss_export_cred 6-100
  gss_init_sec_context 6-103
  gss_inquire_context 6-107
  gss_inquire_cred 6-109
  gss_inquire_cred_by_mech 6-111
  gss_inquire_mechs_for_name 6-113
  gss_inquire_names_for_mech 6-114
  gss_krb5_get_ccache 6-142
  gss_krb5_get_tkt_flags 6-143
  gss_oid_to_str 6-115
  gss_process_context_token 6-116
  gss_release_buffer 6-117
  gss_release_cred 6-118
  gss_release_name 6-119
  gss_release_oid 6-120
  gss_release_oid_set 6-121
  gss Seal 6-128
  gss_sign 6-98
  gss_str_to_oid 6-122
  gss_test_oid_set_member 6-123
  gss_unseal 6-124
  gss_unwrap 6-124
  gss_verify 6-126
  gss_verify_mic 6-126
  gss_wrap 6-128
  gss_wrap_size_limit 6-130
GSSAPI
buffer
  releasing storage 6-117
context
  accepting 6-80
  acceptor 6-71
  checking validity time 6-91
  deleting 6-93
  export 6-133
  extracting DCE credential from 6-134, 6-136
  import 6-137
  indicator 6-71
  initiating 6-103
  processing 6-116
  registering acceptor identity 6-139
GSSAPI (continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gss_duplicate_name</td>
<td>6-97</td>
</tr>
<tr>
<td>gss_get_mic</td>
<td>6-98</td>
</tr>
<tr>
<td>gss_import_name</td>
<td>6-100</td>
</tr>
<tr>
<td>gss_indicate_mechs</td>
<td>6-102</td>
</tr>
<tr>
<td>gss_inquire_context</td>
<td>6-103</td>
</tr>
<tr>
<td>gss_inquire_cred</td>
<td>6-109</td>
</tr>
<tr>
<td>gss_inquire_cred_by_mech</td>
<td>6-111</td>
</tr>
<tr>
<td>gss_inquire_mechs_for_name</td>
<td>6-113</td>
</tr>
<tr>
<td>gss_inquire_names_for_mech</td>
<td>6-114</td>
</tr>
<tr>
<td>gss_krb5_get_ccache</td>
<td>6-142</td>
</tr>
<tr>
<td>gss_krb5_get_tkt_flags</td>
<td>6-143</td>
</tr>
<tr>
<td>gss_oid_to_str</td>
<td>6-115</td>
</tr>
<tr>
<td>gss_process_context_token</td>
<td>6-116</td>
</tr>
<tr>
<td>gss_release_buffer</td>
<td>6-117</td>
</tr>
<tr>
<td>gss_release_cred</td>
<td>6-118</td>
</tr>
<tr>
<td>gss_release_name</td>
<td>6-119</td>
</tr>
<tr>
<td>gss_release_oid</td>
<td>6-120</td>
</tr>
<tr>
<td>gss_release_oid_set</td>
<td>6-121</td>
</tr>
<tr>
<td>gss_seal</td>
<td>6-128</td>
</tr>
<tr>
<td>gss_sign</td>
<td>6-98</td>
</tr>
<tr>
<td>gss_str_to_oid</td>
<td>6-122</td>
</tr>
<tr>
<td>gss_test_oid_set_member</td>
<td>6-123</td>
</tr>
<tr>
<td>gss_unseal</td>
<td>6-124</td>
</tr>
<tr>
<td>gss_unwrap</td>
<td>6-124</td>
</tr>
<tr>
<td>gss_verify</td>
<td>6-126</td>
</tr>
<tr>
<td>gss_verify_mic</td>
<td>6-126</td>
</tr>
<tr>
<td>gss_wrap</td>
<td>6-128</td>
</tr>
<tr>
<td>gss_wrap_size_limit</td>
<td>6-130</td>
</tr>
<tr>
<td>gssdce_add_oid_set_member</td>
<td>6-88</td>
</tr>
<tr>
<td>gssdce_create_empty_oid_set</td>
<td>6-92</td>
</tr>
<tr>
<td>gssdce_cred_to_login_context</td>
<td>6-132</td>
</tr>
<tr>
<td>gssdce_export_cred</td>
<td>6-133</td>
</tr>
<tr>
<td>gssdce_extractcreds_from_sec_context</td>
<td>6-134</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_crd</td>
<td>6-135</td>
</tr>
<tr>
<td>gssdce_extract_PAC_from_sec_context</td>
<td>6-136</td>
</tr>
<tr>
<td>gssdce_import_crd</td>
<td>6-137</td>
</tr>
<tr>
<td>gssdce_login_context_to_crd</td>
<td>6-138</td>
</tr>
<tr>
<td>gssdce_register_acceptor_identity</td>
<td>6-139</td>
</tr>
<tr>
<td>gssdce_set_cred_context_ownership</td>
<td>6-140</td>
</tr>
<tr>
<td>gssdce_test_oid_set_member</td>
<td>6-123</td>
</tr>
</tbody>
</table>

Index

- **GSSAPI (continued)**
- **gssdce_test_oid_set_member API**  6-123

**H**

handle, binding

**ACL**

- getting by network address  6-190
- getting for object  6-189
- changing to individual components  2-309
- client authorization, obtaining  2-64, 2-67
- client, copying  2-58
- code set tags, placing  2-87
- communication time limits, determining  2-130
- context, lookup
  - creating  2-169
  - removing conditions  2-171
  - showing compatible servers  2-173
- converting
  - client to server  2-76
  - to string representation  2-84
- creating
  - from string representation  2-62
  - fully bound server binding handle  2-111
  - creating from components of string binding  2-307
  - creating tower vector from  2-314
  - from tower representation  2-312
- import conditions
  - creating  2-161
- import context
  - removing name service  2-163
  - modifying  2-126
- name service expiration, setting  2-215
- object
  - getting by network address  6-190
  - obtaining compatible server handle  2-165
  - obtaining global name  2-167
- partially bound
  - nil object UUID  2-112
  - non-nil object UUID  2-112
- registering with name service database  2-158
- releasing ACL  6-204
- removing
  - from endpoint map  2-128
  - from name service database entry  2-178
  - multiple  2-204
- replacing object UUID in server handle  2-83
- resetting for server  2-74
- returning compatible server  2-176
- returning object UUID  2-73
- server authorization, obtaining  2-70
- server handles, list of  2-251
- server, copying  2-58
- setting authorization for server  2-170
- setting timeout value  2-146
- storage, freeing  2-60, 2-86
handle, thread
  passing to spawned threads  2-299
handler, thread cleanup  3-25, 3-26
hash value for UUID, creating  2-321
header
  retrieving from backing store  8-57
header files
  supported
    xds.h   4-27
    xdsbdcp.h   4-35
    xdscds.h   4-39
    xdsqds.h   4-41
    xdsmdup.h   4-44
    xdsap.h   4-47
    xms.h   4-50
    xmsga.h   4-59
    xom.h   4-95
    xomext.h   4-101
header files, supplied  A-1
header, audit record, freeing  6-40
header, audit record, obtaining  6-42
host
  address
    elements, continuous viewing  2-122, 2-124
    removing information  2-126, 2-128
    removing search conditions  2-121
    setting search conditions  2-118
attributes
    obtaining from dced  8-82
    binding handle, freeing registry  6-482
binding handles
  listing compatible  2-173
  obtaining list of  2-251
  requesting compatible  2-167
  retrieving compatible  2-165
  returning compatible  2-176
  binding, starting audit with  6-65
changing keytable  8-126
configuration data
  obtaining from dced  8-83
configure dced  8-156
creating keytable  8-128
dced start security validation status  8-151
deleting dced configuration  8-158
deleting keytable  8-130
disabling authenticated RPC  8-88
disabling service  8-80
disabling specific service  8-160
enabling authenticated RPC  8-87
enabling service  8-81
enabling specific service  8-162
endpoint
  activation  2-126
  adding to endpoint map  2-105, 2-108
  registering  2-98, 2-101
  removing from endpoint map  2-114
host (continued)
  get list of entries  8-138
  getting next key  8-132
information
  deleting  8-112
  reading  8-114
  replacing  8-116
initialize keytable cursor  8-133
interface
  registering  2-262
  registering for use  2-158
  removing from registry  2-264
modifying dced configuration  8-164
password
  adding to keytable  8-124
  read all dced data  8-144
  read dced object  8-141
  registering with DCE  8-85
release cursor from dced data read  8-148
release keytable cursor  8-135
release list of entries  8-140
release storage from dced data read  8-146
remove key  8-136
specifying RPC protocol sequences  8-90
start security validation of dced  8-149
starting DCE-configured  8-166
stop dced security validation  8-152
stopping DCE-configured  8-168
stub, rdacif  6-162
trigger, passing security attributes to  6-214
unregistering with DCE  8-89
UUIDs, configuration and execution
  obtaining from dced  8-84
  validation of host secd  8-154
workload balancing
  get AENAME  2-151
  set AENAME  2-153
host dced entry name, obtaining  8-25
host profile name, obtaining  8-39
ID Mapping API
  global name
    generating from cell and group UUID  6-237, 6-241
    generating from cell and principal UUID  6-239, 6-243
    parsing into components  6-247, 6-249, 6-253, 6-255
    translating into cell and principal UUID  6-247, 6-249, 6-253, 6-255
id_base.h  A-4
id_base.idl  A-9
id_epac.h  A-4
ID, global
  from cell and group UUID  6-237, 6-241
  from cell and principal UUID  6-239, 6-243
  parsing into cell/group, names/UUIDs  6-247, 6-249, 6-253, 6-255
ID, global (continued)
  translating into cell and principal UUID 6-247, 6-249, 6-253, 6-255
ID, UNIX
  from name of registry PGO item 6-456
  from PGO item UUID 6-452
  getting PGO element by item name
    getting by UUID 6-450
    PGO item name, converting to 6-462
identification, creating new UUID 2-317
identifier, thread 3-62
identifier, thread, comparing 3-42
identifying
  IDL encoding operation 2-52
identity information, security, getting 6-229
IDL
  See Interface Definition Language (IDL)
IDL APIs 2-38—2-57
IDL base data types 2-1
IDL-to-C Type Mappings 2-1
idl_es_decode_buffer API 2-39
  returning
    buffer decoding handle to IDL 2-39
idl_es_decode_incremental API 2-41
  returning
    incremental decoding handle to IDL 2-41
idl_es_encode_dyn_buffer API 2-43
  returning
    dynamic buffer encoding handle to IDL 2-43
idl_es_encode_fixed_buffer API 2-45
  returning
    fixed buffer encoding handle to IDL 2-45
idl_es_encode_incremental API 2-47
  returning
    incremental encoding handle to IDL 2-47
idl_es_handle_free API 2-49
  freeing
    IDL encoding handle 2-49
idl_es_inq_attrs API 2-50
idl_es_inq_encoding_id API 2-52
idl_es_set_attrs API 2-54
idl_es_set_transfer_syntax API 2-56
idl_es.h A-4
idlbase.h A-4
idlddefs.h A-4
import conditions
  creating 2-161
import context
  removing name service 2-163
importing interface, removing context 2-163
information, audit event
  adding to audit record 6-53
  freeing 6-39
  obtaining 6-41
initializing
  Access Control List (ACL) 6-21
initializing single thread routine 3-61
interface
  directory
    modifying 4-24
  XDS, initializing 4-10
Interface Definition Language (IDL)
  base data types 2-1
  encoding services
    getting flags from 2-50
    identifying operation 2-52
    setting flags from 2-54
    specifying handle for 2-56
  IDL-to-C data-type mappings 2-1
    rdacil 6-162
interface handle 2-13
interface identifier
  data structure 2-14
  obtaining a listing 2-134
  returning for interface specification 2-117
  vector 2-15
interface lookup (NSI)
  removing conditions 2-171
  setting conditions 2-169
interface, rules for removing 2-264
intermediary, delegation
  obtaining privilege attribute handle for 6-222
iovector.h A-4
iovector.idl A-9
iteration
  backing store, preparing for 8-63
  key, returning next during 8-60, 8-61, 8-62
  state, freeing 8-59
K
Kerberos
  address
    array of, copying 7-63
    comparing 7-10
    copying to new structure 7-62
    determining presence in address list 7-11
    releasing storage assigned to 7-73
    releasing storage assigned to array of 7-74
  AP_REP message
    releasing storage from 7-75
  authentication context
    allocating initial vector in 7-26
    creating 7-25
    generating addresses 7-13
    releasing 7-12
    retrieving encryption key from 7-18
    returning address of initial vector 7-17
    returning addresses stored in 7-14
    returning authenticator from 7-15
    returning current flags 7-16
Kerberos (continued)
  authentication context (continued)
    returning local message from  7-19
    returning local subsession key from  7-20
    returning ports stored in  7-24
    returning remote message from  7-21
    returning remote subsession key from  7-23
    returning replay cache for  7-21
    setting address values in  7-29
    setting checksum type  7-27
    setting flags  7-30
    setting initial vector  7-31
    setting network ports in  7-32
    setting replay cache for  7-33
    setting user key in  7-34
    zeroing initial vector in  7-26
  authentication data
    array of, releasing storage from  7-76
  authenticator
    contents of, releasing storage assigned to  7-78
    converting to replay entry  7-35
    copying  7-65
    releasing storage assigned to  7-77
  checksum
    copying  7-66
    releasing storage assigned to  7-79
    setting  7-27, 7-28
  compatibility mode
    setting  7-133, 7-134
  configuration
    files  7-4
    krb5krb5.conf  7-7
    requests  7-169
  context
    creating  7-117
    releasing  7-80
  credentials
    array of, releasing storage assigned to  7-92
    contents of, releasing storage assigned to  7-81
    copying  7-87
    releasing storage assigned to  7-82
  credentials cache
    closing  7-42
    default name, returning  7-44
    default, resolving  7-43
    defining new  7-53
    deleting  7-45
    description  7-3
    ending reading of  7-46
    generating new  7-47
    initializing  7-51
    name, resolving  7-56
    name, returning  7-48
    removing entry from  7-54
    retrieving entries  7-60
    retrieving set  7-57
    returning  6-142

Kerberos (continued)
  credentials cache (continued)
    returning next entry from  7-52
    returning principal associated with  7-49
    setting processing flags for  7-59
    storing  7-61
    type, returning  7-50
  data object
    copying  7-68
    releasing storage assigned to  7-83
  data structures
    array of, copying  7-64
    description  7-1
    encrypted ticket part
      releasing storage assigned to  7-84
  encryption types
    default, setting  7-170, 7-172
    returning  7-105
    returning default  7-107
  error code
    returning  7-175
  error message
    releasing storage assigned to  7-85
  files, using  7-2
  host list
    releasing storage assigned to  7-90
  interface routines
    krb5_address_compare  7-10
    krb5_address_search  7-11
    krb5_auth_con_free  7-12
    krb5_auth_con_genaddrs  7-13
    krb5_auth_con_getaddrs  7-14
    krb5_auth_con_getauthenticator  7-15
    krb5_auth_con_getflags  7-16
    krb5_auth_con_getivector  7-17
    krb5_auth_con_getkey  7-18
    krb5_auth_con_getlocalseqnumber  7-19
    krb5_auth_con_getlocalsubkey  7-20
    krb5_auth_con_getports  7-24
    krb5_auth_con_getrcache  7-21
    krb5_auth_con_getremoteseqnumber  7-22
    krb5_auth_con_getremotesubkey  7-23
    krb5_auth_con_init  7-25
    krb5_auth_con_initivector  7-26
    krb5_auth_con_set_req_cksumtype  7-27
    krb5_auth_con_set_safe_cksumtype  7-28
    krb5_auth_con_setaddrs  7-29
    krb5_auth_con_setflags  7-30
    krb5_auth_con_setivector  7-31
    krb5_auth_con_setports  7-32
    krb5_auth_con_setrcache  7-33
    krb5_auth_con_setuserkey  7-34
    krb5_auth_to_rep  7-35
    krb5_build_principal  7-36
    krb5_build_principal_ext  7-37
    krb5_build_principal_ext_va  7-38
    krb5_build_principal_va  7-40
Kerberos (continued)

interface routines (continued)

    krb5_cc_close 7-42
    krb5_cc_default 7-43
    krb5_cc_default_name 7-44
    krb5_cc_destroy 7-45
    krb5_cc_end_seq_get 7-46
    krb5_cc_generate_new 7-47
    krb5_cc_get_name 7-48
    krb5_cc_get_principal 7-49
    krb5_cc_get_type 7-50
    krb5_cc_initialize 7-51
    krb5_cc_next_cred 7-52
    krb5_cc_register 7-53
    krb5_cc_remove_cred 7-54
    krb5_cc_resolve 7-56
    krb5_cc_retrieve_cred 7-57
    krb5_cc_set_flags 7-59
    krb5_cc_start_seq_get 7-60
    krb5_cc_store_cred 7-61
    krb5_copy_address 7-62
    krb5_copy_addresses 7-63
    krb5_copy_authdata 7-64
    krb5_copy_authenticator 7-65
    krb5_copy_checksum 7-66
    krb5_copy_creds 7-67
    krb5_copy_data 7-68
    krb5_copy_keyblock 7-69
    krb5_copy_keyblock_contents 7-70
    krb5_copy_principal 7-71
    krb5_copy_ticket 7-72
    krb5_free_address 7-73
    krb5_free_addresses 7-74
    krb5_free_ap_rep_enc_part 7-75
    krb5_free_context 7-76
    krb5_free_keyblock 7-77
    krb5_free_keyblock_contents 7-78
    krb5_free_creds 7-79
    krb5_free_checksum 7-80
    krb5_free_credential_contents 7-81
    krb5_free_cred_data 7-82
    krb5_free_enc_tkt_part 7-83
    krb5_free_enc_ticket 7-84
    krb5_free_error 7-85
    krb5_free_host_realm 7-86
    krb5_free_keyblock 7-87
    krb5_free_Keyblock 7-88
    krb5_free_keyblock_contents 7-89
    krb5_free krbhst 7-90
    krb5_free_principal 7-91
    krb5_free_tkt_creds 7-92
    krb5_free_ticket 7-93
    krb5_free_tickets 7-94
    krb5_gen_replay_name 7-95
    krb5_generate_seq_number 7-96
    krb5_generate_subkey 7-97
    krb5_get_crd_from_kdc 7-98

Kerberos (continued)

interface routines (continued)

    krb5_get_cred_from_kdc_renew 7-99
    krb5_get_cred_from_kdc_validate 7-100
    krb5_get_cred_via_tkt 7-101
    krb5_get_credentials 7-102
    krb5_get_credentials_renew 7-103
    krb5_get_credentials_validate 7-104
    krb5_get_default_in_tkt_ktypes 7-105
    krb5_get_default_realm 7-106
    krb5_get_default_tgs_ktypes 7-107
    krb5_get_host_realm 7-108
    krb5_get_in_tkt_with_keytab 7-109
    krb5_get_in_tkt_with_password 7-111
    krb5_get_in_tkt_with_krbhst 7-113
    krb5_get_krbhst 7-115
    krb5_get_server_cache 7-116
    krb5_init_context 7-117
    krb5_kt_add_entry 7-118
    krb5_kt_close 7-119
    krb5_kt_default 7-120
    krb5_kt_default_name 7-121
    krb5_kt_end_seq_get 7-122
    krb5_kt_free_entry 7-123
    krb5_kt_get_entry 7-124
    krb5_kt_get_name 7-125
    krb5_kt_get_type 7-126
    krb5_kt_next_entry 7-127
    krb5_kt_read_service_key 7-128
    krb5_kt_register 7-129
    krb5_kt_remove_entry 7-130
    krb5_kt_resolve 7-131
    krb5_kt_start_seq_get 7-132
    krb5_md4_crypto_compat_ctl 7-133
    krb5_md5_crypto_compat_ctl 7-134
    krb5_mk_error 7-135
    krb5_mk_priv 7-136
    krb5_mk_rep 7-137
    krb5_mk_rep_extended 7-140
    krb5_mk_safe 7-141
    krb5_os_hostaddr 7-142
    krb5_os_localaddr 7-143
    krb5_parse_name 7-144
    krb5_principal_compare 7-145
    krb5_random_confounder 7-146
    krb5_rc_close 7-147
    krb5_rc_default 7-148
    krb5_rc_default_name 7-149
    krb5_rc_destroy 7-150
    krb5_rc_expunge 7-151
    krb5_rc_free_entry_contents 7-152
    krb5_rc_get_lifespan 7-153
    krb5_rc_initialize 7-154
    krb5_rc_get_type 7-155
    krb5_rc_recover 7-157

Index X-45
Kerberos (continued)

interface routines (continued)

krb5_rc_register_type 7-158
krb5_rc_resolve 7-159
krb5_rc_store 7-160
krb5_rd_error 7-161
krb5_rd_priv 7-162
krb5_rd_rep 7-164
krb5_rd_req 7-165
krb5_rd_safe 7-167
krb5_realm_compare 7-168
krb5_set_config_files 7-169
krb5_set_default_in_tkt_ktypes 7-170
krb5_set_default_realm 7-171
krb5_set_default_tgs_ktypes 7-172
krb5_sname_to_principal 7-173
krb5_svc_get_level 7-174
krb5_svc_get_msg 7-175
krb5_timeofday 7-176
krb5_unparse_name 7-177
krb5_unparse_name_ext 7-178
krb5_us_timeofday 7-179

KDC hosts
returning list of 7-115

KDC reply
releasing storage assigned to 7-87

KDC server
obtaining service ticket from 7-98, 7-101
renewing service ticket from 7-99
validating service ticket from 7-100

key table
adding new entry to 7-118
closing 7-119
default, resolving 7-120
default, returning 7-121
defining type 7-129
description 7-4
ending reading of 7-122
getting ticket with 7-109
releasing storage from 7-123
removing entry from 7-130
resolving name 7-131
retrieving entries sequentially 7-132
retrieving service key from 7-128
returning entry from 7-124
returning name 7-125
returning next entry from 7-127
returning type 7-126

keyblock
contents of, releasing storage assigned to 7-89
copying 7-69
copying contents of 7-70
releasing storage assigned to 7-88

krb5/krb.conf 7-4
krb5/krb5.conf 7-5

MD5 checksum
setting compatibility mode for 7-133

Kerberos (continued)

MD5 checksum
setting compatibility mode for 7-134
message
AP_REP, creating 7-137
AP_REP, processing 7-164
AP_REP, releasing storage of 7-75
AP_REQ, creating 7-138, 7-140
AP_REQ, processing 7-165
KRB_ERROR, creating 7-135
KRB_ERROR, processing 7-161
KRB_PRIV, creating 7-136
KRB_PRIV, processing 7-162
KRB_SAFE, creating 7-141
KRB_SAFE, processing 7-167
releasing storage assigned to 7-85

network addresses
returning 7-142, 7-143

principal
building 7-36, 7-37, 7-38, 7-40
comparing 7-145
comparing realms of 7-168
converting service name to 7-173
creating to text 7-177, 7-178
copying 7-71
creating from text string 7-144
releasing storage assigned to 7-91
returning 7-49

random sequence number
generating 7-96

realm
comparing 7-168
default, setting 7-171
purpose of 7-1
returning default 7-106
returning hosts for 7-115

realm list
releasing storage assigned to 7-86

realm name
generating 7-108

release level
returning 7-174

replay cache
closing 7-147
default name, returning 7-149
default, resolving 7-148
deleting 7-150
deleting expired entries from 7-151
description 7-3
generating for server use 7-116
initializing 7-156
name, resolving 7-159
name, returning 7-154
recovering 7-157
releasing storage from 7-152
returning Authenticator lifespan 7-153
storing new entry in 7-160
Kerberos (continued)
replay cache (continued)
type, defining new 7-158
type, returning 7-155
replay cache name
generating 7-95
replay entry
converting authenticator to 7-35
reply, KDC
releasing storage assigned to 7-87
service name
converting to principal 7-173
service ticket
obtaining 7-102
obtaining from KDC server 7-98, 7-101
renewing from KDC server 7-99
validating from KDC server 7-100
services 7-4
session key
going initial ticket from 7-113
subsession key
getting initial ticket from 7-111
terminology 7-2
text password
getting initial ticket from 7-111
ticket
array of, releasing storage assigned to 7-94
copying 7-72
going initial 7-111, 7-113
initial, getting 7-109
releasing storage assigned to 7-93
renewing 7-103
validating 7-104
ticket flags
returning 6-143
time
returning 7-176, 7-179
key
adding for host 8-124
changing for host 8-126
creating for host 8-128
deleting for host 8-130
getting next 8-132
initialize cursor 8-133
release cursor 8-135
remove 8-136
key data value, creating 3-47
key management API (continued)
inserting local key value 6-281
key
deleting by type and version 6-263
deleting from local key storage 6-261
retrieving from local storage 6-270
next key
generating by version 6-274
generating from local storage 6-272
obsolete keys
deleting 6-266
key storage, local
deleting keys from 6-261
getting next key from 6-272
retrieving keys from 6-270
key values, thread specific 3-45
key version number 6-258, 6-261, 6-263, 6-268, 6-270, 6-272, 6-274, 6-281
keymgmt.h A-5
keymgmt.idl A-9
keytab
See also keytable
adding keys to 8-124
keytable
changing keys in 8-126
creating 8-128
deleting 8-130
getting next key 8-132
initialize cursor 8-133
release cursor 8-135
remove key 8-136
keytable file (security) 6-338
krb5_address_compare API 7-10
krb5_address_search API 7-11
krb5_auth_con_free API 7-12
krb5_auth_con_genaddrs API 7-13
krb5_auth_con_getaddrs API 7-14
krb5_auth_con_getauthenticator API 7-15
krb5_auth_con_getflags API 7-16
krb5_auth_con_getivector API 7-17
krb5_auth_con_getkey API 7-18
krb5_auth_con_getlocalseqnumber API 7-19
krb5_auth_con_getremotesubkey API 7-20
krb5_auth_con_getports API 7-24
krb5_auth_con_getrcauthenticator API 7-21
krb5_auth_con_getremotesubkey API 7-22
krb5_auth_con_getremotesubkey API 7-23
krb5_auth_con_init API 7-25
krb5_auth_con_initvector API 7-26
krb5_auth_con_set_req_cksumtype API 7-27
krb5_auth_con_set_safe_cksumtype API 7-28
krb5_auth_con_setaddrs API 7-29
krb5_auth_con_setflags API 7-30
krb5_auth_con_setvector API 7-31
krb5_auth_con_setports API 7-32
Index

M
management, key
automatic key change prior to expiration 6-278
changing principal directory object key 6-258
cursor
   locating in local key store 6-276
   releasing initialized storage 6-280
data types 6-170
freeing storage used by key 6-265
generating random key, specific type 6-268
inserting local key value 6-281
key
   deleting by type and version 6-263
   deleting from local key storage 6-261
   retrieving from local storage 6-270
next key
   getting by version 6-274
   getting from local storage 6-272
obsolete keys
deleting 6-266
management, storage
allocating
   for stubs 2-276
allocating storage within scheme for stubs 2-290
character string storage allocation, freeing 2-311
deleting key 6-261
deleting key by manager type and version 6-263
deleting keys from local storage 6-261
disabling storage for stub 2-296
enabling allocation for stubs 2-297
enabling storage management for spawned threads 2-299
login account
   changing name details 6-366
defining name 6-354
deleting from registry 6-358
replacing all data 6-368
Login API
data types 6-171
Login APIs 6-284—6-351
login context
   constructing with ERAs 6-329
   creating network credentials 6-327
   exporting to other process 6-304
   freeing storage 6-326
   removing 6-322
   renewing ticket-granting authentication 6-324
   validating and certifying 6-335
   validating client 6-343
   validating identity prior to use 6-345
login credentials, network information, getting 6-318
login data, registry API, getting 6-424
lookup, interface (NSI)
   removing conditions 2-171
   setting conditions 2-169

L
ldbase.h A-5
ldbase.idl A-9
LDAP (Lightweight Directory Access Protocol) 4-1
ldrall.h A-5
ldrxa.h A-5
ldrxb.h A-5
leaf entry, adding to directory 4-4
length, audit record, obtaining 6-43
Lightweight Directory Access Protocol (LDAP) 4-1
local cell name, obtaining 8-31
local host name, obtaining 8-35
local key store
   inserting key in 6-281
   locating cursor in 6-276
local time
   converting
      from binary to 5-16
      to binary 5-53
lock, advisory
   applying on a backing store 8-64
locking a mutex 3-52, 3-54
login
   identity
      setting up 6-333
management, storage (continued)
establishing environment for stubs 2-281
exchanging current allocation for stubs 2-288
exchanging storage allocation with client caller 2-305
freeing
  - binding handle 2-86
  - client stub storage 2-278
  - id structures 2-116
  - statistics vector storage 2-148
  - storage used by binding handle 2-60
  - vector storage 2-86
freeing allocated storage for stubs 2-298
freeing key 6-265
freeing space used by ACL entries 6-20
freeing storage allocated for stubs 2-282
freeing storage used to store ID structures 2-116
freeing stub storage 2-280
obtaining thread for stub environment 2-283, 2-286
overriding stub defaults 2-301
protocol sequence, freeing vector storage 2-246
reclaiming
  - client resources 2-279
  - client stub context handle storage 2-295
releasing initialized key storage 6-280
releasing storage for client stub 2-294
setting allocation for stubs 2-284
manager
  - ACL, writing 6-145
manager entry point vector
    - description 2-15
    - determining 2-253
manager routines, rules for calling 2-263
manager type
  - ACL permissions, listing 6-201
  - ACL print permission bits 6-199
  - key, deleting by version 6-263
manager types protecting object, listing 6-147
manager, ACL, registering type of 6-22
map, endpoint
  - element search parameters
    - removing 2-121
    - setting 2-118
  - freeing storage used to store ID structures 2-116
  - server address information
    - activating 2-126
    - adding 2-101, 2-105, 2-108
    - changing 2-98
    - removing 2-114, 2-128
    - replacing 2-101
  - server binding handle endpoint, removing 2-74
  - viewing server endpoint elements 2-122, 2-124
Mapping API
  - global name
    - generating from cell and group UUID 6-237, 6-241
    - generating from cell and principal UUID 6-239, 6-243
    - parsing into components 6-247, 6-249, 6-253, 6-255
    - translating into cell and principal UUID 6-247, 6-249, 6-253, 6-255
marshall.h A-5
members, registry PGO, group or organization 6-445
membership group or organization, registry, verifying 6-453
memory management
  - allocating
    - for stubs 2-276
  - allocating storage within scheme for stubs 2-290
  - character string storage allocation, freeing 2-311
  - deleting key 6-261
  - deleting key by manager type and version 6-263
  - deleting keys from local storage 6-261
  - disabling storage for stub 2-296
  - enabling allocation for stubs 2-297
  - enabling storage management for spawned threads 2-299
  - establishing environment for stubs 2-281
  - exchanging current allocation for stubs 2-288
  - exchanging storage allocation with client caller 2-305
  - freeing
    - binding handle 2-86
    - client stub storage 2-278
    - id structures 2-116
    - statistics vector storage 2-148
    - storage used by binding handle 2-60
    - vector storage 2-86
  - freeing allocated storage for stubs 2-298
  - freeing key 6-265
  - freeing space used by ACL entries 6-20
  - freeing storage allocated for stubs 2-282
  - freeing stub storage 2-280
  - obtaining thread for stub environment 2-283, 2-286
  - overriding stub defaults 2-301
  - protocol sequence, freeing vector storage 2-246
  - reclaiming
    - client resources 2-279
    - client stub context handle storage 2-295
  - releasing initialized key storage 6-280
  - releasing storage for client stub 2-294
  - setting allocation for stubs 2-284
message
  - determining largest to wrap 6-130
  - encryption
    - in GSSAPI 6-128
    - signature, generating in GSSAPI 6-98
Message Handling System (MHS)
  - Directory User Package 4-44
  - Message Store General attributes 4-59
  - object and attribute definitions 4-50
misc.h A-5
misc.idl A-9
modifying
  - IDL encoding operation 2-50, 2-54, 2-56
mutex (mutual exclusivity), thread
  - attribute object, creating 3-56
  - attribute object, deleting 3-57
mutex (mutual exclusivity), thread (continued)
creating 3-51
creating mutex attribute object 3-56
deleting 3-50
deleting mutex attribute object 3-57
global
locking 3-49
unlocking 3-70
global, locking 3-49
global, unlocking 3-70
locking 3-52, 3-54
recursive 3-49
type, getting 3-58
type, setting 3-59
unlocking 3-55

Name
name
global
from cell and group UUID 6-237, 6-241
from cell and principal UUID 6-239, 6-243
parsing into components 6-247, 6-249, 6-253, 6-255
translating into cell and principal UUID 6-247, 6-249,
6-253, 6-255
name for PGO item, getting by UUID 6-450
name service database
adding member to group entry name 2-192
binding handle, retrieving compatible entry
creating 2-207
deleting 2-209
entry name 2-211
entry name, obtaining expanded version 2-180
entry name, resolving 2-182
global expiration age
changing 2-225
global, getting 2-217
group attribute, deleting from entry 2-190
interface lookup
See interface lookup (NSI)
local expiration age, setting 2-215
profile attribute
adding element 2-229
removing from entry 2-227
profile elements
remove viewing conditions 2-234
removing 2-238
set viewing conditions 2-231
viewing individually 2-236
removing
import context 2-163
multiple binding handles 2-204
object UUID binding handle 2-178
showing compatible servers 2-173

Name Service Interface (NSI)
code sets array, freeing 2-213
code sets, getting supported 2-247
code sets, maximum bytes used for encoding 2-249
code sets, reading for server 2-219
database, removing import context 2-163
database, retrieving compatible binding handle 2-165
environment variables 2-8
group attribute, removing member 2-200
import context, adding evaluation routine 2-202
permissions 2-19
crpc server entry, adding attribute 2-223
rcrp server entry, removing attribute 2-221

Name Service, DCE (NSI)
code sets array, freeing 2-213
code sets, getting supported 2-247
code sets, maximum bytes used for encoding 2-249
code sets, reading for server 2-219
database, removing import context 2-163
database, retrieving compatible binding handle 2-165
environment variables 2-8
group attribute, removing member 2-200
import context, adding evaluation routine 2-202
permissions 2-19
crpc server entry, adding attribute 2-223
crpc server entry, removing attribute 2-221

namespace 4-2
routines
listing 4-2

nbasis.h A-5
nbasis.idl A-9
ncastat.h A-5
ncastat.idl A-9
ndr_rep.h A-5
ndr.ld A-5
ndr.ld.idl A-9
ndrtypes.h A-5
nested remote procedure calls 2-76
Network Authentication Service, certifying ID 6-291
network credentials, login context 6-327, 6-331
creating 6-327
network identity
binding handle, getting for login 6-311
setting up client login 6-333
network identity, getting binding handle 6-311
network information
getting login context 6-318
login data, freeing storage 6-308
network information, freeing storage 6-308
network information, principal, getting 6-318
network login prior to use 6-345
nil UUID 2-318

NSI
See Name Service, DCE (NSI)
O

object
  adding permissions for 6-17
OM
attribute values, extracting 4-26
  copying 4-64
private
  copying as public object 4-72
  copying attribute value strings 4-66
  copying attribute values 4-77
  creating 4-68
  deleting 4-70
  deleting attribute values 4-82
  determining class 4-75
extracting OM type 4-86
  reading a string 4-80
writing a string 4-84
public
  copying attribute values 4-77
  creating new 4-72
  deleting service-generated 4-70
determining class 4-75
object ACL, getting 6-155
object identifier (OI)
  value
    initializing OM descriptor with 4-90
object type
  assigning 2-244
determining class 2-240
  registering inquiry function 2-242
object UUID
  creating inquiry context for viewing 2-184
  removing inquiry context for viewing NSI objects 2-186
  viewing one at a time 2-188
object, directory
  ACL access for noncalling process 6-209
obsolete keys, deleting 6-266
OM (Object Management)
  descriptor, initializing 4-89
  descriptor, initializing with OI value 4-90
  object attribute values, extracting 4-26
  object, converting from descriptor to string format 4-91
object, converting from string to descriptor format 4-93
type, extracting 4-86
om_copy API 4-64
om_copy_value API 4-66
om_create API 4-68
om_delete API 4-70
om_get API 4-72
om_instance API 4-75
om_put API 4-77
om_read API 4-80
om_remove API 4-82
om_write API 4-84
omX_extract API 4-86
omX_fill API 4-89
omX_fill_oid API 4-90
omX_object_to_string API 4-91
omX_string_to_object API 4-93
one-way authentication 2-136
open connection, binding 6-486
organization authentication policy 6-464
organization of book xxv
oride_base.h A-5
oride_base.idl A-9

P

passwd.h A-5
passwd.idl A-9
password 6-348
  adding for host 8-124
  changing for host 8-126
  creating for host 8-128
  deleting for host 8-130
  getting next 8-132
  initialize cursor 8-133
  policy, establishing 6-351
  release cursor 8-135
  remove 8-136
password, registry login account, changing 6-364
passwords, generating 6-349
passwords, obtaining validation type 6-350
permissions
  ACL
    file group class 6-191
    listing for object 6-192
  adding for ACL entry 6-13, 6-16
  adding for foreign user 6-14
  adding for group 6-15
  adding for object 6-17
  adding for unauthorized entry 6-18
  adding for user 6-19
  object 6-201
  obtaining 6-6
  of principal to an object, determining 6-8
permissions, adding 6-15
person to registry PGO, adding 6-430
person, group, organization, registry, getting by name 6-441
person, group, organization, registry, getting by UNIX ID 6-443
person, group, organization, registry, getting by UUID 6-439
PGO
  adding to registry account 6-428
  deleting member 6-434
element name
  changing 6-457
  getting by name 6-441
  getting by UNIX ID 6-443
PGO (continued)
element name (continued)
  getting by UUID 6-439
  identifying by UNIX number 6-436
  item name
    getting from UNIX ID 6-462
    listing next item in registry 6-448
    members list 6-445
    registry account name for, deleting 6-432
    replacing item data 6-459

pgo.h A-5
pgo.idl A-9
policy, password 6-351
policy.h A-5
policy.idl A-9

POSIX semantics
  of manager types, listing 6-149

predicate, threads, definition 3-29

principal
  determining access to an object 6-8
  directory object key, changing 6-258
  inquiring for an RPC caller 6-10
  security key, changing before expiration 6-278
  UUID, generating global name from 6-239, 6-243
  UUID, translating global name into 6-247, 6-249, 6-253, 6-255

Principal, Group, Organization (PGO)
  adding to registry account 6-428
  deleting member 6-434
  element name
    changing 6-457
    getting by name 6-441
    getting by UNIX ID 6-443
    getting by UUID 6-439
    identifying by UNIX number 6-436
  item name
    getting from UNIX ID 6-462
    listing next item in registry 6-448
    members list 6-445
    registry account name for, deleting 6-432
    replacing item data 6-459

principle name, determining 2-136

print strings, ACL, getting 6-151

priv_attr_trig.h A-6

Privilege Attribute Certificate (EPAC)
  See Extended Privilege Attribute Certificate (EPAC)

privilege attribute handle
  getting identity information 6-229
  obtaining EPAC 6-232
  obtaining required restrictions 6-230
  obtaining target restrictions 6-231
  privilege attribute handle
    obtaining, for a delegation intermediary 6-222

Privilege Attribute, Extended (EPA) 6-167

Privilege Certificate, Extended Attribute (continued)
  obtaining from privilege attribute handle 6-232
  reading 6-145

privileges
  ACL
    file group class 6-191
    listing for object 6-192
    adding for ACL entry 6-13, 6-16
    adding for foreign user 6-14
    adding for group 6-15
    adding for object 6-17
    adding for unauthorized entry 6-18
    adding for user 6-19
    object 6-201
    obtaining 6-6
    of principal to an object, determining 6-8

privileges of login name, updating 6-356

processor, threads 3-71

profile attribute, removing 2-227

profile elements, NSI
  adding 2-229
  remove viewing conditions 2-234
  removing 2-238
  setting viewing conditions 2-231
  viewing individually 2-236

project list, getting from registry account 6-360

properties, registry database
  getting 6-470
  setting 6-472

protection level for authentication, setting 2-132

protocol sequence
  determining support 2-156
  freeing storage 2-246
  ncacn_ip_tcp 2-16, 2-18
  ncadg_ip_udp 2-16, 2-18
  obtaining vector 2-155
  registering 2-266, 2-268
  specifying with endpoint 2-272

protocol sequence vector 2-16

pthread_attr_create API 3-20
pthread_attr_delete API 3-21
pthread_attr_getstacksize API 3-22
pthread_attr_setstacksize API 3-23
pthread_cancel API 3-24
pthread_cleanup_pop API 3-25
pthread_cleanup_push API 3-26
pthread_cond_broadcast API 3-27
pthread_cond_destroy API 3-28
pthread_cond_init API 3-29
pthread_cond_signal API 3-31
pthread_cond_timedwait API 3-32
pthread_cond_wait API 3-34
pthread_condattr_create API 3-36
pthread_condattr_delete API 3-37
pthread_create API 3-38
pthread_delay_np API 3-40
pthread_detach API 3-41
pthread_equal API 3-42
pthread_exit API 3-43
pthread_get_expiration_np API 3-44
pthread_getspecific API 3-45
pthread_join API 3-46
pthread_keycreate API 3-47
pthread_lock_global_np API 3-49
pthread_mutex_destroy API 3-50
pthread_mutex_init API 3-51
pthread_mutex_init API 3-51
pthread_mutex_lock API 3-52
pthread_mutex_trylock API 3-54
pthread_mutex_unlock API 3-55
pthread_mutexattr_create API 3-56
pthread_mutexattr_delete API 3-57
pthread_mutexattr_getkind_np API 3-58
pthread_mutexattr_setkind_np API 3-59
pthread_once API 3-61
pthread_self API 3-62
pthread_setsasynccancel API 3-63
pthread_setscancel API 3-65
pthread_setspecific API 3-67
pthread_signal_to_cancel_np API 3-68
pthread_testcancel API 3-69
pthread_unlock_global_np API 3-70
pthread_yield API 3-71
record (continued)
formatting 6-51
header, freeing 6-40
header, getting 6-42
length, obtaining 6-43
reading 6-45
record, audit
adding event information to 6-53
discarding 6-38
formatting 6-51
header, freeing 6-40
header, getting 6-42
length, obtaining 6-43
reading 6-45
registering
ACL manager object type 6-22
registry account for login name, adding 6-354
registry account login data, getting 6-426
Registry API
Access Control List (ACL)
retrieving ACL types of schema objects 6-400
account data
obtaining 6-362
replacing all 6-368
account login data
obtaining 6-426
ACL strings
obtaining, for an object 6-389
administrative account
updating 6-356
attribute instances
creating and updating 6-412
updating from control instances 6-409
attribute routine parameter
cursor, allocating resources to 6-393
cursor, initializing 6-394
cursor, releasing 6-396
cursor, resetting 6-397
attributes
obtaining information about 6-373, 6-374, 6-376, 6-377, 6-378, 6-380
reading, expanded 6-382
reading, for specific object 6-385, 6-387
authentication policy
obtaining 6-417
obtaining effective 6-415
setting 6-419
bind connection to query site 6-487
bind connection to update site 6-489
bind open connection 6-486
binding to
to bind open connection 6-486
cell registry 6-421
freeing binding handle 6-482
getting binding handle information 6-480
registry query site 6-476
registry site 6-474
record
adding event information to 6-53
discarding 6-38
query site, registry, binding to 6-476
query, attribute trigger 6-211
-racl APIs 6-145–6-162
-rdaclif.idl file 6-162
read only status query (registry) 6-485
RAISE 3-7
RAISE macro 3-7
rdacl APIs 6-145–6-162
rdacl_get_access API 6-145
rdacl_get_manager_types API 6-147
rdacl_get_mgr_types_semantics API 6-149
rdacl_get_printstring API 6-151
rdacl_get_referral API 6-153
rdacl_lookup API 6-155
rdacl_replace API 6-157
rdacl_test_access API 6-159
rdacl_test_access_on_behalf API 6-161
rdaclbase.h A-6
rdacl++.h A-6
rdacl++.idl A-9
rdacl++.idl file 6-162
rdacl++.idl0.h A-6
read only status query (registry) 6-485
registry account for login name, adding 6-354
registry account login data, getting 6-426
record
adding event information to 6-53
discarding 6-38
Registry API (continued)

binding to (continued)
  registry update site 6-478
client account name
  adding 6-428
constants 6-187
data types 6-172
database cursor location
  resetting 6-422
database properties
  getting 6-470
  setting 6-472
login account name
  changing details 6-366
  defining 6-354
  deleting 6-358
name
  getting 6-483
password
  changing 6-364
  UNIX, for group ID 6-497
  UNIX, for group name 6-495
person
  adding to group or organization 6-430
  Principal, Group, Organization (PGO)
    adding person to group or organization 6-430
    deleting account name 6-432
    deleting member of group or organization 6-434
    element name from UNIX ID 6-462
    getting element by name 6-441
    getting element by UNIX ID 6-443
    getting element by UUID 6-439
    getting item name from UUID 6-450
    getting UUID for item from UNIX ID 6-461
    item UUID, getting UNIX ID from 6-452
    item, getting UNIX ID from name of 6-456
    item, identifying by UNIX number 6-436
    item, name, changing 6-457
    item, named, getting UUID from 6-455
    listing group or organization members 6-445
    listing next item in database 6-448
    replacing item data 6-459
project list
  obtaining from account 6-360
querying read only status 6-485
schema entry
  creating 6-391
  deleting 6-398
  read by name 6-404
  read by UUID 6-402
  reading a number of 6-405
  updating 6-407
UNIX group entry
  getting for UNIX group name 6-493
UNIX ID
  getting 6-491
Registry API (continued)

updates
  blocking until replicas match 6-499
user information
  replacing 6-371
validating membership in 6-453
Registry APIs 6-354—6-499
defining login account name 6-354
registry site, binding to 6-474
registry update site server handle, binding to 6-478
relative binary time
  comparing
    two 5-24, 5-27
  converting
    to absolute value 5-4
    to tm structure 5-62
relative binary timestamp 5-4
relative time, definition 5-1
Remote Procedure Call (RPC)
ACL permissions 2-19
arguments, frequently used
  binding 2-20
  name 2-21
  name_syntax 2-21
  status 2-21
  string 2-21
  UUID 2-22
binding handle, routines for obtaining 2-20
caller, inquiring security principal and group 6-10
client binding handles, listing 2-9
daemon
  adding entries 8-104
  adding key to host keytable 8-124
  binding to 8-97
  changing key to host keytable 8-126
  configure dced 8-156
  creating host information 8-110
  creating host keytable 8-128
  deleting configuration 8-158
  deleting host information 8-112
  deleting host keytable 8-130
  disabling specific service 8-160
  enabling specific service 8-162
  get list of entries 8-138
  getting next key 8-132
  initialize cursor 8-133
  list of entries, initializing cursor 8-118
  listing entries for a service 8-106
  modifying configuration 8-164
  name to UUID 8-120
  read all dced data 8-144
  read dced object 8-141
  reading host information 8-114
  release cursor 8-135
  release cursor from data read 8-148
  release list of entries 8-140
  release storage from data read 8-146
Remote Procedure Call (RPC) (continued)
daemon (continued)
  remove key 8-136
  removing entries for a service 8-108
  replacing host information 8-116
  security validation status 8-151
  start security validation 8-149
  stopping DCE-configured server 8-166
  stopping security validation 8-152
  unbinding from 8-99
  UUID to name 8-122
  validation of host secid 8-154

data types and structures
  binding handle 2-9
  binding vector 2-10
  boolean32 2-11
  code set 2-11
  code sets array 2-12
  conversion type 2-12
  endpoint map enquiry handle 2-13
  global name 2-13
  IDL encoding service handle 2-13
  interface handle and specification 2-13
  interface identifier (id) 2-14
  interface identifier vector 2-15
  manager entry point vector 2-15
  name service handle 2-15
  protocol sequence 2-16
  protocol sequence vector 2-16
  statistics vector 2-17
  string binding 2-17
  string UUID 2-19
  unsigned character string 2-19
  UUID vector 2-19
dced binding
  setting authentication and authorization information 8-102

environment variables
  RPC_DEFAULT_ENTRY 2-8
  RPC_DEFAULT_ENTRY_SYNTAX 2-8
  for DCE Security Service 6-163
  group
    adding member name 2-192
    creating inquiry context for viewing 2-194
    removing inquiry context for viewing 2-196
    viewing individually 2-198
  name syntaxes
    valid 2-21
  nested 2-76
  nonstub support routines, listing 2-3
  runtime
    authentication information, registering 2-258
    interface specification, registering protocol sequence 2-274
    protocol sequence and endpoint, registering 2-272
    registering protocol sequence 2-268
Remote Procedure Call (RPC) (continued)
runtime (continued)
  registering server interface 2-262
  registering single protocol sequence 2-270
  supported protocol sequences registering 2-266
  runtime services 2-2
  server binding handles, listing 2-9
  start server listening for 2-255
  stub support routines, listing 2-2
remote procedure calls
  is server listening 2-140
  stopping server listen for 2-149
replicating registry 6-499
required security restrictions, getting 6-230
RERAISE 3-11
RERAISE macro 3-11
restrictions, delegation, getting from privilege attribute handle 6-221
rgybase.h A-6
rgybase.idl A-9
rgynbase.h A-6
rgynbase.idl A-9
routine processing error, determining 8-77
routines, auditing 6-29
RPC
  See Remote Procedure Call (RPC)
RPC APIs 2-58—2-314
  rpc_binding_copy API 2-58
  rpc_binding_free API 2-60
  rpc_binding_from_string_binding API 2-62
  rpc_binding_inq_auth_caller API 2-64
  rpc_binding_inq_auth_client API 2-67
  rpc_binding_inq_auth_info API 2-70
  rpc_binding_inq_object API 2-73
  rpc_binding_reset API 2-74
  rpc_binding_server_from_client API 2-76
  rpc_binding_set_auth_info API 2-79
  rpc_binding_set_object API 2-83
  rpc binding from string binding API 2-84
  rpc_binding_vector_free API 2-86
  rpc_cs_binding_set_tags API 2-87
  rpc_cs_char_set_compat_check API 2-91
  rpc_cs_eval_with_universal API 2-91
  rpc_cs_eval_without_universal API 2-93
  rpc_cs_get_tags API 2-95
  RPC_DEFAULT_ENTRY 2-8
  RPC_DEFAULT_ENTRY_SYNTAX 2-8
  rpc_ep_register API 2-98
  rpc_ep_register_no_replace API 2-105
  rpc_ep_register_no_replace_wlb API 2-108
  rpc_ep_register_wlb API 2-101
  rpc_ep_resolve_binding API 2-111
  rpc_ep_unregister API 2-114
  rpc_get_ae_name API 2-151
  rpc_if_id_vector_free API 2-116
  freeing storage used to store ID structures 2-116
Index
S

deleting 6-398
read by name 6-404
read by UUID 6-402
reading a number of 6-405
updating 6-407
sec_acl routines 6-189—6-210
sec_acl_bind API 6-189
sec_acl_bind_to_addr API 6-190
sec_acl_calc_mask API 6-191
sec_acl_get_access API 6-192
sec_acl_get_error_info API 6-194
sec_acl_get_manager_types API 6-195
sec_acl_get_mgr_types_semantics API 6-197
sec_acl_get_printstring API 6-199
sec_acl_list, replacing ACL with 6-157
sec_acl_lookup API 6-201
sec_acl_release API 6-203
sec_acl_release_handle API 6-204
sec_acl_replace API 6-205
replacing
ACL 6-205
sec_acl_test_access API 6-207
sec_acl_test_access_on behalf API 6-209
sec_attr routines 6-211—6-216
sec_attr_base.h A-7
sec_attr_tools.h A-7
sec_attr_trig_query API 6-211
sec_attr_trig_update API 6-214
sec_attr_trig.h A-7
sec_base.h A-7
sec_base.idl A-9
sec_cred_free_attr_cursor API 6-216
sec_cred_free_cursor API 6-217
sec_cred_free_pa_handle API 6-218
sec_cred_get_authz_session_info API 6-219
sec_cred_get_client_princ_name API 6-220
sec_cred_get_deleg_restrictions API 6-221
delegation
restrictions, getting from privilege attribute
handle 6-221
sec_cred_get_delegate API 6-222
sec_cred_get_delegation_type API 6-224
sec_cred_get_extended_attr API 6-225
sec_cred_get_initiator API 6-227
sec_cred_get_opt_restrictions API 6-228
sec_cred_get_pa_data API 6-229
sec_cred_get_req_restrictions API 6-230
sec_cred_get_tgt_restrictions API 6-231
sec_cred_get_v1_pac API 6-232
sec_cred_initialize_attr_cursor API 6-233
sec_cred_initialize_cursor API 6-234
sec_cred_is_authenticated API 6-235
sec_cred.h A-7
sec_id routines 6-237—6-256
sec_id_gen_group API 6-237
sec_id_gen_name API 6-239
sec_id_global_gen_group API 6-241
sec_id_global_gen_name API 6-243
sec_id_global_is_group_member API 6-245
sec_id_global_parse_group API 6-247
sec_id_global_parse_name API 6-249
sec_id_is_group_member API 6-251
sec_id_parse_group API 6-253
sec_id_parse_name API 6-255
sec_key_mgmt_change_key API 6-258
sec_key_mgmt_delete_key API 6-261
sec_key_mgmt_delete_key_type API 6-263
sec_key_mgmt_free_key API 6-265
sec_key_mgmt_garbage_collect API 6-266
sec_key_mgmt_gen_rand_key API 6-268
sec_key_mgmt_get_key API 6-270
sec_key_mgmt_get_next_key API 6-272
sec_key_mgmt_get_next_kvno API 6-274
sec_key_mgmt_initialize_cursor API 6-276
sec_key_mgmt_manage_key API 6-278
sec_key_mgmt_release_cursor API 6-280
sec_key_mgmt_set_key API 6-281
sec_login_become_delegate API 6-284
sec_login_become_impersonator API 6-287
sec_login_become_initiator API 6-289
sec_login_certify_identity API 6-291
sec_login_context_token_owner API 6-293
sec_login_create_context_token API 6-294
sec_login_cred_get_delegate API 6-296
sec_login_cred_get_initiator API 6-298
sec_login_cred_init_cursor API 6-299
sec_login_delete_context_token API 6-300
sec_login_disable_delegation API 6-301
sec_login_expand_context_token API 6-302
sec_login_export_context API 6-304
sec_login_export_context_data API 6-306
sec_login_free_net_info API 6-308
sec_login_get_current_context API 6-309
sec_login_get_expiration API 6-311
sec_login_get_groups API 6-313
security (continued)

delegation (continued)
  obtaining privilege attributes for 6-227
  obtaining privilege attributes of intermediary 6-296
  obtaining type 6-224
  server 6-284, 6-287

generic interface (GSSAPI) 6-71

identity
  Network Authentication Service, certifying ID 6-291
  identity information from privilege attribute handle 6-229

initializing 6-234

interface
  using other than DCE secure RPC 6-71
  using peer-to-peer communications 6-71

key, deleting by version 6-263
key, getting next by version number 6-274
key, getting next from local storage 6-272
key, random generation by type 6-268
key, retrieving from local storage 6-270

network information
  freeing storage used by 6-308
  passing to trigger server 6-214

password
  deleting 6-348
  establishing policy 6-351
  generating 6-349
  obtaining validation type 6-350

password handle
  freeing 6-348

principal
  determining group member 6-245, 6-251
  obtaining name from credential 6-220

privilege attribute handle
  getting identity information 6-229
  getting restrictions from 6-228
  obtaining EPAC 6-232
  obtaining required restrictions 6-230
  obtaining target restrictions 6-231

protection
  on object, listing types of ACLs 6-147, 6-149

restrictions
  obtaining from privilege handle 6-228
  required, getting from privilege attribute handle 6-230
  target, getting from privilege attribute handle 6-231

security (GSSAPI)

buffer
  releasing storage 6-117

credential
  authenticating 6-235
  credential, initializing 6-234

cursor
  initializing 6-299
  cursor, attribute
    initializing 6-233
  cursor, credential 6-234
delegation
  creating login context 6-289
  disabling 6-301
  obtaining information about initiator 6-298
  obtaining privilege attribute handle for intermediary 6-222

context
  accepting 6-80
  acceptor 6-71
  checking validity time 6-91
  deleting 6-93
  export 6-133
  extracting DCE credential from 6-134, 6-136
  import 6-137
  indicator 6-71
  initiating 6-103
security (GSSAPI) (continued)
context (continued)
  processing 6-116
  registering acceptor identity 6-139
  returning information about 6-107
credential
  acquiring 6-84
  adding 6-86
  change ownership of DCE credential login context 6-140
  creating credential from DCE login context 6-138
  obtaining DCE login context associated with 6-132
  obtaining information about 6-109
  releasing local data structures 6-118
  returning information for security mechanism 6-111
  returning Kerberos 6-142
data types 6-75
DCE extensions 6-71
displaying status codes 6-95
duplicating internal name 6-97
error handling 6-73
  internal name
    reducing to mechanism name 6-89
Kerberos extensions 6-71
mechanism availability
  determining 6-102
  returning 6-111, 6-113, 6-114
messages
  determining largest 6-130
  encrypting 6-128
  encryption signature, generating 6-98
  encryption signature, verifying 6-126
  signing 6-128
  unwrapping 6-124
names
  comparing 6-90
  internal, converting printable name to 6-100
  internal, displaying 6-94
  internal, releasing storage for 6-119
object ID (OID)
  adding 6-88
  checking 6-123
  converting to 6-122
  converting to string representation 6-115
  creating new set of 6-92
  releasing storage 6-120, 6-121
overview 6-71
routines
  gss_accept_sec_context 6-80
  gss_acquire_cred 6-84
  gss_add_cred 6-86
  gss_add_oid_set_member 6-88
  gss_canonicalize_name 6-89
  gss_compare_name 6-90
  gss_context_time 6-91
  gss_create_empty_oid_set 6-92
  gss_delete_sec_context 6-93
security (GSSAPI) (continued)
routines (continued)
gss_display_name 6-94
  gss_display_status 6-95
  gss_duplicate_name 6-97
  gss_get_mic 6-98
  gss_import_name 6-100
  gss_indicate_mechs 6-102
  gss_init_sec_context 6-103
  gss_inquire_context 6-107
  gss_inquire_cred 6-109
  gss_inquire_cred_by_mech 6-111
  gss_inquire_mechs_for_name 6-113
  gss_inquire_names_for_mech 6-114
  gss_krb5_get_tkt_flags 6-143
  gss_oid_to_str 6-115
  gss_process_context_token 6-116
  gss_release_buffer 6-117
  gss_release_cred 6-118
  gss_release_name 6-119
  gss_release_oid 6-120
  gss_release_oid_set 6-121
  gss_seal 6-128
  gss_sign 6-98
  gss_str_to_oid 6-122
  gss_test_oid_set_member 6-123
  gss_unseal 6-124
  gss_unwrap 6-124
  gss_verify 6-126
  gss_verify_mic 6-126
  gss_wrap 6-128
  gss_wrap_size_limit 6-130
  gssdce_add_oid_set_member 6-88
  gssdce_create_empty_oid_set 6-92
  gssdce_cred_to_login_context 6-132
  gssdce_export_cred 6-133
  gssdce_extract_creds_from_sec_context 6-134
  gssdce_extract_PAC_from_credential 6-135
  gssdce_extract_PAC_from_sec_context 6-136
  gssdce_import_cred 6-137
  gssdce_login_context_to_cred 6-138
  gssdce_register_acceptor_identity 6-139
  gssdce_set_cred_context_ownership 6-140
  gssdce_test_oid_set_member 6-123
standard 6-71
security attributes
  obtaining for a delegation intermediary 6-222
  passing to trigger server 6-214
  reading 6-211
security authorization
  accessing services 6-163
  authentication information, registering 2-258
  authentication protection level, setting 2-132
  checking
    client 6-12
Index X-61
security authorization (continued)
controlling remote access 2-142
obtaining information from client 2-67
obtaining information from handle for client 2-64
obtaining information from server 2-70
server principal name, getting 2-136
setting for server binding handle 2-79

security credentials
authentication of, checking 6-12
cursor, initializing 6-234
extracting EPAC from 6-135, 6-136
freeing attribute cursor 6-216
freeing cursor 6-217
freeing privilege attribute handle 6-218
GSSAPI, acquiring 6-84
obtaining 6-5
obtaining client 6-219
obtaining principal name from 6-220

security permissions
ACL
file group class 6-191
listing for object 6-192
adding for ACL entry 6-13, 6-16
adding for foreign user 6-14
adding for group 6-15
adding for object 6-17
adding for unauthorized entry 6-18
adding for user 6-19
object 6-201
obtaining 6-6
of principal to an object, determining 6-8

determining access to an object 6-8
directory object key, changing 6-258
inquiring for an RPC caller 6-10
security key, changing before expiration 6-278
UUID, generating global name from 6-239, 6-243
UUID, translating global name into 6-247, 6-249, 6-253, 6-255

security Registry API (continued)
attribute routine parameter (continued)
cursor, releasing 6-396
cursor, resetting 6-397
attributes
obtaining information about 6-373, 6-374, 6-376, 6-377, 6-378, 6-380
reading, expanded 6-382
reading, for specific object 6-385, 6-387
authentication policy
obtaining 6-417
obtaining effective 6-415
setting 6-419
bind connection to query site 6-487
bind connection to update site 6-489
bind open connection 6-486
binding to
bind open connection 6-486
cell registry 6-421
freeing binding handle 6-482
getting binding handle information 6-480
registry query site 6-476
registry site 6-474
registry update site 6-478
client account name
adding 6-428
data types 6-172
database cursor location
resetting 6-422
database properties
getting 6-470
setting 6-472
login account name
changing details 6-366
defining 6-354
deleting 6-358
name
getting 6-483
password
changing 6-364
UNIX, for group ID 6-497
UNIX, for group name 6-495
person
adding to group or organization 6-430
Principal, Group, Organization (PGO)
adding person to group or organization 6-430
deleting account name 6-432
deleting member of group or organization 6-434
element name from UNIX ID 6-462
generating element by name 6-441
generating element by UNIX ID 6-443
generating element by UUID 6-439
generating item name from UUID 6-450
getting UUID for item from UNIX ID 6-461
item, getting UNIX ID from 6-452
item, identifying by UNIX number 6-436
security Registry API (continued)
Principal, Group, Organization (PGO) (continued)
  item, name, changing 6-457
  item, named, getting UUID from 6-455
  listing group or organization members 6-445
  listing next item in database 6-448
  replacing item data 6-459
project list
  obtaining from account 6-360
querying read only status 6-485
schema entry
  creating 6-391
  deleting 6-398
  read by name 6-404
  read by UUID 6-402
  reading a number of 6-405
  updating 6-407
UNIX group entry
  getting for UNIX group name 6-493
  UNIX ID
    getting 6-491
  updates
    blocking until replicas match 6-499
  user information
    replacing 6-371
  validating membership in 6-453
Security Service, DCE
overview 6-163
remote procedure call, types 6-163
unsupported APIs, listing 6-163
semantics, POSIX
  of manager types, listing 6-149
sequence, protocol
  determining support 2-156
  freeing storage 2-246
  obtaining vector 2-155
  registering 2-266, 2-268
  specifying with endpoint 2-272
server
  address
    elements, continuous viewing 2-122, 2-124
    removing information 2-126, 2-128
    removing search conditions 2-121
    setting search conditions 2-118
  attributes
    obtaining from dced 8-82
  binding handle, freeing registry 6-482
  binding handles
    listing compatible 2-173
    obtaining list of 2-251
    requesting global name 2-167
    retrieving compatible 2-165
    returning compatible 2-176
  binding, starting audit with 6-65
  changing keytable 8-126
  configuration data
    obtaining from dced 8-83
server (continued)
  configure dced 8-156
  creating keytable 8-128
  dced start security validation status 8-151
  deleting dced configuration 8-158
  deleting keytable 8-130
  disabling authenticated RPC 8-88
  disabling service 8-80
  disabling specific service 8-160
  enabling authenticated RPC 8-87
  enabling service 8-81
  enabling specific service 8-162
  endpoint
    activation 2-126
    adding to endpoint map 2-105, 2-108
    registering 2-98, 2-101
    removing from endpoint map 2-114
  get list of entries 8-138
  getting next key 8-132
  information
    deleting 8-112
    reading 8-114
    replacing 8-116
    initialize keytable cursor 8-133
  interface
    registering 2-262
    registering for use 2-158
    removing from registry 2-264
    modifying dced configuration 8-164
    password
      adding to keytable 8-124
    read all dced data 8-144
    read dced object 8-141
    registering with DCE 8-85
    release cursor from dced data read 8-148
    release keytable cursor 8-135
    release list of entries 8-140
    release storage from dced data read 8-146
    remove key 8-136
    specifying RPC protocol sequences 8-90
    start security validation of dced 8-149
    starting DCE-configured 8-166
    stop dced security validation 8-152
    stopping DCE-configured 8-168
    stub, rdaclif 6-162
    trigger, passing security attributes to 6-214
    unregistering with DCE 8-89
  UUIDs, configuration and execution
    obtaining from dced 8-84
    validation of host seed 8-154
    workload balancing
      get AENAME 2-151
      set AENAME 2-153
server applications, waiting for incoming calls 2-255
server listening for remote procedure calls 2-149

Index X-63
server runtime statistics 2-138
server time to canceled call, orphaning 2-144
server, listening for remote procedure calls 2-140
service-generated object, deleting 4-70
signal mask, thread 3-68
signature, encryption (GSSAPI) 6-98
signing messages (GSSAPI) 6-128
sigwait API 3-72
site, update, referral 6-153
stack, thread cleanup 3-26
stacksize, thread attribute object 3-22
starting a thread 3-38
statistics vector 2-17
statistics vector, freeing storage 2-148
status codes, API
  obtaining message text for 8-77
storage
  releasing locally 6-203
storage allocation, disabling 2-296
storage management
  allocating
    for stubs 2-276
  allocating storage within scheme for stubs 2-290
  character string storage allocation, freeing 2-311
  deleting key 6-261
  deleting key by manager type and version 6-263
  deleting keys from local storage 6-261
  disabling storage for stub 2-296
  enabling allocation for stubs 2-297
  enabling storage management for spawned threads 2-299
  establishing environment for stubs 2-281
  exchanging current allocation for stubs 2-288
  exchanging storage allocation with client caller 2-305
freeding
  binding handle 2-86
  client stub storage 2-278
  id structures 2-116
  statistics vector storage 2-148
  storing used by binding handle 2-60
  vector storage 2-86
  freeing allocated storage for stubs 2-298
  freeing key 6-265
  freeing space used by ACL entries 6-20
  freeing storage allocated for stubs 2-282
  freeing stub storage 2-280
  obtaining thread for stub environment 2-283, 2-286
  overriding stub defaults 2-301
  protocol sequence, freeing vector storage 2-246
reclaiming
  client resources 2-279
  client stub context handle storage 2-295
  releasing initialized key storage 6-280
  releasing storage for client stub 2-294
  setting allocation for stubs 2-284
store, backing
  See backing store
store, local key
  inserting key in 6-281
  locating cursor in 6-276
string binding 2-17
string key match 8-27
string text, using to create binding handle 2-307
string UUID 2-19
Strong Authentication Package 4-47
stub
  allocating storage within scheme 2-290
  authenticating client identity from client 2-292
  disabling storage for 2-296
  enabling storage allocation for 2-297
  establishing storage management environment 2-281
  exchanging current storage allocation 2-288
  freeing allocated storage for 2-282
  freeing storage 2-280
  freeing storage allocated for 2-298
  obtaining thread for storage management environment 2-283, 2-286
  overriding storage management defaults 2-301
  reclaiming context handle storage 2-295
  releasing storage for 2-294
  setting storage allocation 2-284
stub, client
  exchange storage allocation with client caller 2-305
  freeing storage 2-278
  memory management, overriding defaults 2-301
  storage used by, releasing 2-294
stubbase.h A-8
successive extraction of security key by version 6-274
successive key extraction from local storage 6-272
supplied
  ACF files A-8
  IDL files A-8
supplied header files A-1
system time, DCE, retrieving 5-30, 5-31
T
target security restrictions, getting 6-231
TDF 5-1
text string, as arbitrary time zone 5-12
thread attribute object
  changing stacksize 3-23
  creating 3-20
  creating on condition variable 3-36
  deleting 3-21
  deleting on condition variable 3-37
  getting stacksize 3-22
  mutex, creating 3-56
  mutex, deleting 3-57
thread cancellation
  actions taken upon 3-38
thread cancelation (continued)
  asynchronous 3-63
delivering request 3-69
destructor 3-39
general 3-65
signal mask method 3-68
terminating execution 3-24

thread cleanup
  handler, establishing 3-26
  handler, running 3-25
  stack 3-26

thread condition variable
  creating 3-29
creating attributes object on 3-36
deleting 3-28
deleting attributes object on 3-37
waiting 3-34
waiting for signal 3-32
waiting threads, waking all 3-27
waking all waiting threads 3-27
waking one thread 3-31

thread mutex
  attribute object, creating 3-56
  attribute object, deleting 3-57
creating 3-51
creating mutex attribute object 3-56
deleting 3-50
deleting mutex attribute object 3-57
global
  locking 3-49
  unlocking 3-70
global, locking 3-49
global, unlocking 3-70
locking 3-52, 3-54
recursive 3-49
type, getting 3-58
type, setting 3-59
unlocking 3-55

thread(s) (continued)
cleanup routines 3-3
comparing two exception objects 3-16
condition variable
  See thread condition variable
condition variable, definition 3-29
creating 3-38
defining exception actions 3-12
delay interval, setting 3-40
deletion, marking for 3-41
destructor 3-39
enabling package 3-5
errno variable 3-5
exception handling routines 3-4
expiration time, getting 3-44
functional groups
  See also threads functional groups
  list 3-1
handle
  obtaining for storage management
    environment 2-283, 2-286
    passing to spawned threads 2-299
    setting 2-303
identifier, obtaining, of current 3-62
identifiers, comparing 3-42
key data value, creating 3-47
key values, data associated with 3-45
macros
  See threads macros
management 3-1
mutex 3-2
object, creating 3-38
object, definition 3-38
object, exception 3-6
overview 3-1
processor, releasing 3-71
raising current exception 3-11
raising status exception 3-17
reason code, retrieving exception 3-14
routines
  unsupported 3-1
setting exception status value 3-18
signal mask cancelation method 3-68
signal, causing wait for 3-32
sigwait routine 3-4
single thread routine, initializing 3-61
spawned
  enabling storage management 2-299
  setting handle for 2-303
specific data, setting 3-67
starting 3-38
status value, retrieving exception 3-15
status, creating exception 3-15
status, obtaining information 3-5
status, specifying exception 3-17
terminating calling thread 3-43

and z/OS abends 3-14
asynchronous cancelation 3-63
attribute object 3-2
  See also thread attribute object
attribute object, changing stacksize 3-23
attribute object, creating 3-20
attribute object, deleting 3-21
attribute object, getting stacksize 3-22
calling thread, terminating 3-43
cancelation
  See thread cancelation
cancelation actions 3-38
cancelation routines 3-3
catching current exception 3-13
catching exceptions 3-9, 3-10
classes, exception macro 3-12
cleanup
  See thread cleanup

Index  X-65
thread(s) (continued)
  termination, wait for 3-46
  thread-specific data routines 3-3
  time delay interval, setting 3-40
  variable condition 3-3
  waiting 3-34
  waiting for asynchronous signal 3-72
  waking one thread 3-31

threads exceptions
  catching a particular 3-9
  catching all 3-10
  catching current 3-13
  clauses 3-12
  comparing two objects 3-16
  defining actions 3-12
  defining scope for 3-8
  enabling exception-returning interface 3-5
  GOTO, nonlocal, use of 3-10
  object, initializing 3-6
  overview 3-5
  raising current 3-11
  raising status 3-17
  reason code, retrieving 3-14
  RETURN, use of 3-10
  searching for 3-7
  setting status value 3-18
  status value, retrieving 3-15
  status, creating 3-15

threads functional groups
  attribute object routines 3-2
  cancelation routines 3-3
  cleanup routines 3-3
  exception handling routines 3-4, 3-5
  management routines 3-1
  mutex routines 3-2
  priority routines 3-3
  scheduling routines 3-3
  sigwait routine 3-4
  thread-specific data routines 3-3
  variable condition routines 3-3

threads macros
  CATCH 3-9
  CATCH_ALL 3-10
  comparing two objects 3-16
  exc_first_catch 3-13
  exc_raise_status 3-17
  exc_set_status 3-18
  EXCEPTION_INIT 3-6
  FINALLY 3-12
  RAISE 3-7
  RERAISE 3-11
  TRY-ENDTRY 3-8

time (continued)
  bounds, event 5-22
  coordinated universal (UTC)
    definition 5-1
    interval, deriving from two binary timestamps 5-64
    interval, UTC
      deriving from two binary timestamps 5-64
  local
    converting from binary to 5-16
  relative
    converting to binary 5-44
    converting to relative binary 5-55
    converting to text string 5-18
    converting to timespec structure 5-19
    definition 5-1
  system, DCE, retrieving 5-30, 5-31
  zone
    converting binary to arbitrary text string 5-12
    converting binary to GMT 5-14
    converting binary to local 5-16
    converting relative to text string 5-18
    information, getting 5-10

time differential factor (TDF)
  definition 5-1
  using to convert binary to tm structure 5-8

Time Services, Distributed (DTS)
  APIs
    listing by function 5-2
    summary 5-2
    time calculating 5-4, 5-6, 5-57, 5-59, 5-66
    time comparing 5-24, 5-27
    time converting 5-8, 5-12, 5-14, 5-16, 5-18, 5-19,
      5-21, 5-32, 5-37, 5-41, 5-44, 5-46, 5-48, 5-49, 5-51,
      5-53, 5-55, 5-62
    time manipulating 5-22, 5-61, 5-64
    time retrieving 5-30, 5-31
    timezone information 5-10
  terms 5-1

timeout
  determining wait 2-144
  returning value 2-130

timespec relative time with TDF to binary 5-49

timespec structure
  converting
    from binary 5-21
    from relative binary 5-19
    to binary 5-48, 5-49

timespec, time interval, finding 5-19, 5-21

timestamp
  adding
    two binary 5-6
  binary
    converting to earliest, most likely, latest time 5-61
    deriving UTC time interval from two 5-64
  calculating
    difference between two 5-66
timestamp (continued)
  comparing
two relative binary or binary 5-24, 5-27
converting
  binary to arbitrary timezone text string 5-12
  binary to GMT 5-14
  binary to local 5-16
  binary to timespec structure 5-21
  binary to tm structure 5-8
  binary to UTC or GMT tm structure 5-32
  relative binary to absolute value 5-4
  relative binary to text string 5-18
  relative binary to timespec structure 5-19
relative binary
  multiplying by floating-point value 5-57
  multiplying by integer 5-59
timezone information 5-10
timezone information routines
  offset from GMT 5-39
tm structure
  converting
    from relative binary 5-62
    local to binary 5-53
    to binary timestamp 5-51
    to UTC or GMT 5-32
tower vector
  binding handle from 2-312
  creating from binding handle 2-314
  releasing memory 2-313
trail, audit
  closing 6-34
  limiting size 6-54
  opening 6-48
  writing to 6-35
tree, directory information (DIT), in CDS 4-2
trigger, attribute, query 6-211
TRY-ENDTRY 3-8
TRY-ENDTRY macro 3-8
twr.h A-8
twr.idl A-10

U
  unauthorized entry, adding permissions for 6-18
  unbinding from an ACL 6-204
  unbinding from Directory Services 4-22
  universal time, coordinated (UTC) 5-1
Universal Unique Identifier (UUID)
cell 6-237, 6-239, 6-241, 6-243, 6-247, 6-249, 6-253, 6-255
  comparing two 2-315
  context, lookup
    listing compatible 2-173
    removing conditions 2-171
    setting conditions 2-169
  converting
    binary to string 2-323
Universal Unique Identifier (UUID) (continued)
  converting (continued)
    string to binary 2-320
  creating
    hash value 2-321
    new 2-317
    nil UUID 2-318
  determining
    equality 2-319
    value 2-322
  finding ACL’s, by name 6-25
  finding ACL’s, by UUID 6-27
  for ACL, exchanging 6-205
  from named registry PGO item 6-455
  from registry PGO item UNIX ID 6-461
  generating global name from cell and group 6-237, 6-241
  generating global name from cell and principal 6-239, 6-243
  getting name for PGO item by 6-450
  getting PGO element by 6-439
  getting UNIX ID from PGO item 6-452
  group 6-237, 6-241, 6-245, 6-251
  group membership, determining 6-245, 6-251
  inquiry context
    creating for viewing NSI objects 2-184
    manager entry point vector, determining 2-253
    nil
    creating 2-318
    determining 2-322
    object
      creating inquiry context for viewing 2-184
      removing inquiry context for viewing NSI
      objects 2-186
      viewing one at a time 2-188
    object of binding handle, getting 2-73
    object of server binding handle. setting 2-83
    object type
      assigning 2-244
      determining 2-240
      registering inquiry function 2-242
    principal 6-239, 6-243, 6-247, 6-249, 6-253, 6-255
    registering with name service database 2-158
    removing from name service database entry 2-178
    removing multiple 2-204
    retrieving
      from item header in backing store 6-4
      starting audit with 6-68
      to name associated, of entry 8-122
    translating global name into cell and principal 6-247, 6-249, 6-253, 6-255
    vector data structure 2-19
    view object, setting conditions 2-184
    viewing one at a time 2-188
UNIX group entry
  for group ID 6-491
UNIX group entry (continued)
for group name 6-493

UNIX ID
from name of registry PGO item 6-456
from PGO item UUID 6-452
getting PGO element by 6-443
PGO item name, converting to 6-462

UNIX password
for registry group ID 6-497
for registry group name 6-495

unix.h A-8
unlocking a mutex 3-55
unsigned character string 2-19
update attribute trigger server 6-214
update registry, block while replicating 6-499
update site referral, internal 6-153
user registry account data, replacing 6-371
user, adding permissions for 6-19
user, foreign 6-14
utc_abstime API 5-4
utc_addtime API 5-6
utc_anytime API 5-8
utc_ascgmtmtime API 5-10
utc_ascgmtmzone API 5-12
utc_ascgmtlocaltime API 5-14
utc_ascgmtcreltime API 5-16
utc_ascgmttmtime API 5-18
utc_binreltime API 5-19
utc_bintime API 5-21
utc_boundtime API 5-22
utc_cmpintervaltime API 5-24
utc_cmpmtime API 5-27
utc_gettime API 5-30
utc_getusertime API 5-31
utc_gmtime API 5-32
utc_gmtzone API 5-34
utc_loc GMT API 5-37
tm structure
from binary to local time 5-37
utc_localzone API 5-39
utc_mkanytime API 5-41
tm structure
using TDF to binary timestamp 5-41
utc_mkascgmttime API 5-44
utc_mkascmtime API 5-46
utc_mk便秘 API 5-48
utc_mk便秘time API 5-49
utc_mkgmtime API 5-51
utc_mklocaltime API 5-53
utc_mktime API 5-55
utc_multime API 5-57
utc_multime API 5-59
utc_pointtime API 5-61
utc_reftime API 5-62
utc_spantime API 5-64

utc_subtime API 5-66
utc.h A-8
utc_TYPES.h A-8
utc_TYPES.idl A-10

UUID
See Universal Unique Identifier (UUID)

uuid_compare API 2-315
uuid_create API 2-317
uuid_create_nil API 2-318
uuid_equal API 2-319
uuid_from_string API 2-320
string UUID converting to binary 2-320
uuid_hash API 2-321
uuid_is_nil API 2-322
uuid_to_string API 2-323
uuid.h A-8

vector, manager entry point
determining 2-253

vector, tower
binding handle from 2-312
creating from binding handle 2-314
releasing memory 2-313

version number, key 6-258, 6-261, 6-263, 6-268, 6-270, 6-272, 6-274, 6-281

workload balancing
APIs
rpc_ep_register_no_replace_wlb 2-108
rpc_ep_register_wlb 2-101
rpc_get_ae_name 2-151
rpc_mgmt_ep_elt_inq_next_wlb 2-124
rpc_mgmt_ep_set_activated_wlb 2-126
rpc_set_ae_name 2-153
workspace creation (XDS) 4-10

XDS interface
header file definitions
Basic Directory Contents Package 4-35
Cell Directory Services (CDS) 4-39
Directory Services Package 4-27
Global Directory Services (GDS) 4-41
Message Handling System (MHS) 4-50
Message Handling System (MHS) Directory User Package 4-44
MHS Message Store General Attributes 4-59
Strong Authentication Package 4-47
header files
xds.h 4-27
xdsbdcp.h 4-35
xdscds.h 4-39
XDS interface (continued)

header files (continued)
- xdsdgs.h 4-41
- xdsmdup.h 4-44
- xdssap.h 4-47
- xmhp.h 4-50
- xmsga.h 4-59

overview of interfaces to 4-1

packages
- Basic Directory Contents Package 4-35
- Cell Directory Services (CDS) 4-39
- Directory Services Package 4-27
- Global Directory Services (GDS) 4-41
- Message Handling System (MHS) 4-50
- Message Handling System (MHS) Directory
  - User 4-44
- MHS Message Store General Attributes 4-59
- Strong Authentication 4-47

routines
- attribute values, comparing 4-8
- deleting directory workspace 4-21
- directory session, opening 4-6
- extracting attribute values from OM object 4-26
- initializing interface 4-10
- leaf entry, adding to directory 4-4
- listing directory subordinates 4-11
- modifying directory entry 4-13
- modifying directory interface 4-24
- querying directory entry by name 4-15
- removing directory entry 4-17
- unbinding from directory services 4-22

session, ending 4-22

terminology 4-2

xds interface routines
- directory, searching 4-19

xds.h header file 4-27

xds.h, description A-1

xdssbdcp.h header file 4-35

xdssbdcp.h, description A-2

xdsdcs.h header file 4-39

xdsdcs.h, description A-2

xdsdgs.h header file 4-41

xdsdgs.h, description A-2

xdsmdup.h header file 4-44

xdsmdup.h, description A-2

xdssap.h header file 4-47

xmhp.h header file 4-50

xmhp.h, description A-2

xmsga.h A-2

xmsga.h header file 4-59

XOM interface

header files
- xom.h 4-95
- xomext.h 4-101

interface routines
- converting OM object from descriptor to string
  format 4-91

XOM interface (continued)

interface routines (continued)
- converting OM object from string to descriptor
  format 4-93
- copying attribute value strings 4-66
- copying attribute values 4-77
- copying private directory object 4-64
- copying private object as public 4-72
- creating private object 4-68
- deleting attribute values of a private object 4-82
- deleting service-generated object 4-70
- determining class 4-75
- extracting OM type 4-86
- initializing OM descriptor 4-89
- initializing OM descriptor with OI value 4-90
- listing 4-62
- reading a string from a private object 4-80
- writing a string to a private object 4-84

overview of interfaces to 4-1

packages
- Basic Directory Contents Package 4-35
- Cell Directory Services (CDS) 4-39
- Directory Services Package 4-27
- Global Directory Services (GDS) 4-41
- Message Handling System (MHS) 4-50
- Message Handling System (MHS) Directory
  - User 4-44
- MHS Message Store General Attributes 4-59
- Strong Authentication 4-47

session, ending 4-22

terminology 4-2

xom.h A-2

xom.h header file 4-95

xomext.h header file 4-101

xomi.h A-2
Readers' Comments

z/OS
DCE
Application Development Reference
Volumes 1 and 2
Publication No. SC24-5908-00

You may use this form to report errors, to suggest improvements, or to express your opinion on the appearance, organization, or completeness of this book.

Date: ____________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

--- Note ---

Report system problems to your IBM representative or the IBM branch office serving you. U.S. customers can order publications by calling the IBM Software Manufacturing Solutions at 1-800-879-2755.

In addition to using this postage-paid form, you may send your comments by:

FAX 1-607-752-2327 Internet pubrcf@vnet.ibm.com
IBMLink GDLVME(PUBRCF)

Would you like a reply?  YES  NO  If yes, please tell us the type of response you prefer.

___ Electronic address: ____________________________

___ FAX number: _________________________________

___ Mail: (Please fill in your name and address below.)

Name ____________________________  Address ____________________________

Company or Organization ____________________________

Phone No. ____________________________
BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department G60
International Business Machines Corporation
Information Development
1701 North Street
ENDICOTT NY 13760-5553
IBM

z/OS DCE

Application Development Reference Volumes 1 and 2