NN$UMGR - N-NETWORKS EXTENSIBLE USER MANAGEMENT FRAMEWORK USER'S GUIDE AND REFERENCE

VERSION 1 .0.009

Copyright© 2002-2003 by N-Networks

All rights reserved.

Document ID: 001-200501-035
Last Revision: February 14, 2003

COPYRIGHT INFORMATION AND ACKNOWLEDGEMENTS

NN$UMGR and this document are Copyright© 2002-2003 by N-Networks

Oracle is a registered trademark of Oracle Corporation.

PL/SQL, Oracle8*i*, and Oracle9*i* are trademarks of Oracle Corporation.

Other company or product names are mentioned for identification purposes only and may be service marks, trademarks, or registered trademarks of their respective owners.

Although every effort was taken to make this document as accurate and complete as possible, no guarantees whatsoever are given in regard to document's accuracy and completeness. Also, no guarantees are given that this document fully covers the functionality of the product it describes.

Information in this document is subject to change without notice.

INTRODUCTION

N-Networks Extensible User Management Framework, or NN$UMGR for short, is a set of Oracle database objects and PL/SQL packages, which implement simple extensible framework for user management tasks in PL/SQL applications. The framework provides APIs for managing users, groups, roles, permissions and access control lists, and user sessions. The core set of programming interfaces is designed to suit basic requirements for a generic user management system, and hooks are provided to extend the system as needed (for example, to tie permissions managed by NN$UMGR to user-defined database structures, or to extend user profiles by creating additional database objects to store extra data for users and link them through foreign keys to the user data managed by NN$UMGR.)

This document describes the internal architecture of the framework, exposed database objects (tables and views) and packaged APIs provided with the framework.

ARCHITECTURE.

NN$UMGR is installed into separate Oracle schema with minimum set of privileges. Several views and tables are exposed to public, as well as API packages. NN$UMGR manages basic set of user management entities, like users, groups and roles, internally and provides only read and reference access to the underlying tables and views that store this information. All modifications to the data are performed through packaged APIs. Additionally, NN$UMGR maintains separate virtual database for each Oracle user, which may be using the framework. Each Oracle user sees and operates its own set of entities, including custom permissions, access control lists, user sessions, etc. Basic entities are:

- User. This entity reflects a physical application user. Basic attributes for this entity are identifier, login name, password, first, middle and last name, description and status. List of permissions granted to a user can also be considered user's attribute.
- Group. This entity reflects a group of Users and/or Groups. Attributes for this entity are identifier, name, description, status and member list. List of permissions granted to a group can also be considered group's attribute.
- Role. This entity reflects a special case of Group, which can be disabled and enabled at any point in time and can include only Users and Groups, but not other Roles. Attributes for this entity are identifier, name, description, status and member list. Roles should generally be recipients of permissions. Using roles, you can easily grant and revoke permissions to individual users and groups of users by assigning them particular roles – when you disable a role, all permissions granted to it appear to be revoked from role members, and when you re-enable it, all permissions are instantly back.
- Permission. This entity reflects a user-defined permission (a security token, which identifies the right to perform some action or access some entity). Attributes for this entity are identifier, name and description. Developers can create any number of custom permissions and use them as they see fit for their particular project needs.
- Access Control List. This entity reflects a set of Permissions associated with User, Group or Role entity. Each member of the set contains permission identifier and grantee (user, group or role, to which the permission is granted) identifier. ACLs may also be associated with custom entities, but developers should maintain them separately (see Extending the Framework section for an example of custom ACL and its use.)
- Session. This entity reflects User's system access activity. When user logs on, Session entity is automatically created for him and maintained for predetermined period of time or until User logs out. Attributes for this entity are globally unique identifier, owner identifier and last access timestamp.

The above entities comprise complete basic user management system. Framework provides APIs for manipulating Users, Groups, Roles, Permissions, Access Control Lists and Sessions. These APIs are described in detail later in this document.

SOFTWARE REQUIREMENTS

This section lists necessary software to be installed prior to NN$UMGR installation. NN$UMGR requires the following software versions to be present:

- Oracle8*i* Release 3 (Version 8.1.7) or later RDBMS with JServer (Java VM) option installed, Oracle9*i* Release 2 (Version 9.2.0) recommended.
- Dynamic PSP Version 2.1 or later (or NN$PSP_UTL package alone.)

DATABASE OBJECTS EXPOSED BY THE FRAMEWORK.

NN$UMGR exposes several views and tables that can be used to query and reference objects managed by the framework. NN$UMGR also actively uses one custom system context. This chapter describes these database objects and ways to use them to extend the framework to suit your particular project needs.

CONTEXTS.

The framework uses system contexts for maintaining some information during Oracle session and making it easily available through the use of `SYS_CONTEXT()` built-in function.

NN$UMGR_CTX CONTEXT.

One global system context - `NN$UMGR_CTX` - is created during framework installation. This context is initialized and is written by `NN$UMGR` package, and can be read by any Oracle user using `SYS_CONTEXT()` system function as follows:

```
SYS_CONTEXT('NN$UMGR_CTX', 'context variable name')
```

This context receives several variables when a user session is created or reconnected. Variables are:

`CURRENT_USER`	Contains identifier of currently logged in/connected user or `NULL`
`CURRENT_SESSION`	Contains identifier of current session or `NULL`

Here 'user' and 'session' refers to entities managed by the framework. They are not the same and generally have no relation to Oracle users and sessions.

TYPES.

NN$UMGR defines two SQL types which are used extensively throughout the framework. Note that since synonyms for types are not supported in Oracle versions prior to 9.2, developers need to fully qualify them by prefixing them with NN$UMGR schema name.

NN$TPERMLIST TYPE.

Definition:

```
TYPE NN$TPermList IS TABLE OF NUMBER(10,0);
```

This nested table type is used to provide an easy way to acquire and manipulate Access Control Lists. It is associated with a list of permissions assigned to particular entity, either granted or required. The framework maintains only lists of permissions granted to framework-managed entities (users, groups and roles). It is up to developer to maintain lists of permissions required to access certain custom entities.

NN$TPERMNAMESLIST TYPE.

Definition:

```
TYPE NN$TPermNamesList IS TABLE OF VARCHAR2(50);
```

This nested table type is used to provide an easy way to acquire and manipulate Access Control Lists using permission names rather than identifiers.

VIEWS AND TABLES.

NN$V$USERS VIEW AND NN$T$USERS TABLE.

Definition:

Column Name	Column Type	Column Description
ID	NUMBER(10,0)	identifier

Column Name	Column Type	Column Description
LOGIN	VARCHAR2(50)	login name
PASSWORD	VARCHAR2(50)	password hash (passwords are not stored in clear text)
FIRST_NAME	VARCHAR2(100)	First Name (not used for groups and roles)
MIDDLE_NAME	VARCHAR2(100)	Middle Name (not used for groups and roles)
LAST_NAME	VARCHAR2(100)	Last Name (not used for groups and roles)
UTYPE	CHAR(1)	Account type ('U' = User, 'G' = Group, 'R' = Role)
STATUS	NUMBER(1)	Status (1 = enabled, 0 = disabled, 2 = invalidated)
DESCRIPTION	VARCHAR2(250)	Verbose description of the account
CREATED	DATE	Date record created (default SYSDATE)
LAST_MODIFIED	DATE	Date record was last modified (default SYSDATE)

This view provides read access to all Users, Groups and Roles defined in context of current Oracle user. It is built on top of NN$T$USERS table with the same layout. PUBLIC is granted SELECT on this view and REFERENCES on the underlying table's primary key (ID). REFERENCES privilege allows creating related structures in other schemas and maintaining referential integrity with NN$UMGR-controlled data.

Passwords are case-insensitive and are stored as cryptographically strong hashes, not as plain text, to prevent peeking at other users' passwords. This means that if a user forgot his password, there is no way to restore it because used hash function is one-way; it can only be changed to a new one.

NN$V$PERMISSIONS VIEW AND NN$T$PERMISSIONS TABLE.

Definition:

Column Name	Column Type	Column Description
ID	NUMBER(10,0)	Permission identifier
NAME	VARCHAR2(50)	Permission name
DESCRIPTION	VARCHAR2(250)	Verbose permission description

This view provides read access to all permissions defined by particular Oracle user. It is built on top of NN$T$PERMISSIONS table with the same layout. PUBLIC is granted SELECT on this view and REFERENCES on the underlying table's primary key (ID). REFERENCES privilege allows using permissions defined in context of the framework to create access control lists associated with user objects in addition to those maintained internally for Users, Groups and Roles.

NN$V$GROUPS VIEW.

Definition:

Column Name	Column Type	Column Description
GRP_ID	NUMBER(10,0)	Group/role identifier
GRP_TYPE	CHAR(1)	Group type ('G' = Group, 'R' = Role)
GRP_NAME	VARCHAR2(50)	Group/Role name
MEMBER_ID	NUMBER(10,0)	Member identifier
MEMBER_TYPE	CHAR(1)	Member type ('G' = Group, 'U' = User, 'R' = Role)
MEMBER_NAME	VARCHAR2(50)	Member name (login)

This view provides read access to group/role membership information maintained by the framework. `PUBLIC` is granted `SELECT` on this view. Disabled roles do not appear in this view.

NN$V$UMGR_SESSION VIEW AND NN$T$UMGR_SESSION TABLE.

Definition:

Column Name	Column Type	Column Description
`ID`	`VARCHAR2(40)`	Globally unique (GUID) session identifier
`USER_ID`	`NUMBER(10,0)`	User identifier
`LAST_ACCESS`	`DATE`	Last access timestamp

This view provides read access to the user session information maintained by the framework. It is built on top of `NN$T$UMGR_SESSION` table with the same layout. `PUBLIC` is granted `SELECT` on this view and `REFERENCES` on the underlying table's primary key (`ID, USER_ID`). `REFERENCES` privilege allows creating related structures to hold additional information associated with NN$UMGR sessions in other Oracle schemas and maintaining referential integrity between framework-maintained session data and user-maintained session data.

NN$V$MYPERMISSIONS VIEW.

Definition:

Column Name	Column Type	Column Description
`ID`	`NUMBER(10,0)`	Permission identifier
`NAME`	`VARCHAR2(50)`	Permission name

This view provides relational view of currently logged in user's permissions. `PUBLIC` is granted `SELECT` on this view.

NN$V$USER_PERMISSIONS VIEW.

Definition:

Column Name	Column Type	Column Description
`USER_ID`	`NUMBER(10,0)`	User identifier
`PERM_ID`	`NUMBER(10,0)`	Permission identifier
`PERM_NAME`	`VARCHAR2(50)`	Permission name

This view provides relational view of users/groups/roles and their permissions. Users/groups/roles with no permissions do not appear in this view. `PUBLIC` is granted `SELECT` on this view.

NN$V$USER_PERMISSIONS_O VIEW.

Definition:

Column Name	Column Type	Column Description
`USER_ID`	`NUMBER(10,0)`	User identifier
`PERMISSIONS`	`NN$TPermList`	Permissions list (nested table)

This view provides object-relational view of all users/groups/roles and their permissions. All users/groups/roles appear in this view. Those with no assigned permissions will appear with empty `PERMISSIONS` nested table. `PUBLIC` is granted `SELECT` on this view.

NNVVALIDATIONS VIEW.

Definition:

Column Name	Column Type	Column Description
USER_ID	NUMBER(10,0)	User identifier
VAL_CODE_HASH	VARCHAR2(40)	Validation code hash
VAL_STARTED	DATE	Timestamp when validation process initiated (default SYSDATE)
VAL_RECEIVED	DATE	Timestamp when validation was completed (default NULL, non-NULL value means this validation was completed successfully)

This view tracks user validations. Validation is a process of confirming user account. It may be used for newly created accounts to verify certain account attributes, or periodically to confirm that user is still active. The validation process is started by inserting a new row into the internal validations table and sending special random validation code to the user whose account is validated. Once the user replies with proper validation code, VAL_RECEIVED column receives timestamp of successful code verification and the user account is re-enabled. PUBLIC is granted SELECT on this view. No relations may be established with underlying table as validation process is fully performed internally.

Note that the framework provides no way for communicating validation code to the user, it is up to developers to decide how to pass the code to the user and how to receive user's reply. For example, application may send the user an email with an URL pointing to the validation procedure and include validation code in that URL, and validation procedure will receive the code and complete validation when the user opens the URL. This approach may help to ensure that user's email address is correct.

APPLICATION PROGRAMMING INTERFACES (APIs).

This section describes APIs exposed by the framework and gives examples of their use. The framework exposes three packages, namely NN$UMGR_PERM, NN$UMGR_ACL and NN$UMGR, which comprise APIs to manipulate various objects maintained by the framework.

NN$UMGR_PERM PACKAGE.

This package deals with permission definitions. The package provides means to create new permissions, drop existing permissions, and map permission names to identifiers and vice-versa. Permission names are case-insensitive and should be unique for each Oracle user (different users may have permissions with the same names, but their meaning may be different in each case.)

SUMMARY OF SUBPROGRAMS.

```
FUNCTION createPerm(
                     sName        VARCHAR2
                    ,sDescription VARCHAR2
                   ) RETURN NUMBER;
```

This function defines new permission with name sName and description sDescription. It returns new permission identifier if successful, and NULL otherwise.

```
FUNCTION dropPerm(
                   sName VARCHAR2
                 ) RETURN NUMBER;
```

This function removes permission named sName and returns 1 if successful or –1 if failed.

```
FUNCTION dropPerm(
                   nID   NUMBER
                 ) RETURN NUMBER;
```

This function removes permission identified by nID and returns 1 if successful or –1 if failed.

```
FUNCTION getPermID(
                    sName  VARCHAR2
                  ) RETURN NUMBER;
```

This function returns identifier for permission named sName, or NULL if such permission does not exist.

```
FUNCTION getPermName(
                      nID   NUMBER
                    ) RETURN VARCHAR2;
```

This function returns name for permission identified by nID, or NULL if such permission does not exist.

```
FUNCTION mapPermNames(
                       PermNames NN$TPermNamesList
                     ) RETURN NN$TPermList;
```

This function takes an array of permission names as input and returns an array of permission identifiers for all found permissions. If no permissions in `PermNames` array could be resolved to identifiers, function will return empty `NN$TPermList` array, it will never return `NULL`.

```
FUNCTION editPerm(
                  sName        VARCHAR2
                 ,sDescription VARCHAR2
                 ) RETURN NUMBER;
```

This function updates description of `sName` permission to `sDescription`. Returns 1 if successful, -1 if failed.

```
FUNCTION editPerm(
                  nID          NUMBER
                 ,sDescription VARCHAR2
                 ) RETURN NUMBER;
```

This function updates description of `nID` permission to `sDescription`. Returns 1 if successful, -1 if failed.

```
FUNCTION getVersion RETURN VARCHAR2;
```

This function returns current package version as `VARCHAR2` string.

EXAMPLES.

Below are some examples of package calls.

```
declare
       nResult NUMBER;
       bResult Boolean;
       sResult VARCHAR2(100);
       aPerms  NN$TPermList;
begin
  -- create new permission
  nResult := NN$UMGR_PERM.createPerm('SamplePermission', 'Sample Permission');
  -- get permission name (should be uppercase SAMPLEPERMISSION)
  sResult := NN$UMGR_PERM.getPermName(nResult);
  -- example of permission names mapping
  aPerms  := NN$UMGR_PERM.mapPermNames(NN$TPermNamesList('samplepermission'));
end;
```

SECURITY CONSIDERATIONS.

This package does not control who creates, modifies or drops permissions by default. Each Oracle user who wants to turn on internal security checks must perform special security bootstrap procedure. When this procedure is performed, several predefined permissions and roles are created, and only those users that have these special permissions are allowed to manipulate permissions. Note that such privileged users may drop these special permissions effectively disabling internal security.

NN$UMGR_ACL PACKAGE.

This package provides API for manipulating Access Control Lists and permission lists: granting and revoking permissions to Users, Groups and Roles, and checking if effective permissions are adequate to requested ACL. Key difference between ACL and User permissions lists are that user permissions list specifies permissions the user has, while ACL specifies permissions the user should have to access the resource. It is up to developer to implement ACL storage and specification for custom entities. Examples in this section will show generic approach to implementing an ACL and using provided API to verify if a user has adequate privileges to access an object controlled by such ACL.

SUMMARY OF SUBPROGRAMS.

```
FUNCTION checkPerms(
                     Effective NN$TPermList,
                     Required  NN$TPermList,
                     Override  NN$TPermList DEFAULT NULL
                   ) RETURN Boolean;
FUNCTION checkPermsN(
                     Effective NN$TPermList,
                     Required  NN$TPermList,
                     Override  NN$TPermList DEFAULT NULL
                   ) RETURN INT;
FUNCTION checkPerms(
                     Effective NN$TPermList,
                     Required  NN$TPermNamesList,
                     Override  NN$TPermNamesList DEFAULT NULL
                   ) RETURN Boolean;
FUNCTION checkPermsN(
                     Effective NN$TPermList,
                     Required  NN$TPermNamesList,
                     Override  NN$TPermNamesList DEFAULT NULL
                   ) RETURN INT;
```

These functions are the centerpiece of the package. They provide a common way to verify if user's permissions are adequate for requested object's ACL or, optionally, to some list of permissions that override the ACL assigned to particular object. Functions operate with ACLs for objects rather than objects themselves. It is up to developer to maintain these ACLs and retrieve them in a fashion that allows using `checkPerms` function to verify them.

Functions accept three arrays as parameters: `Effective`, which is a list of user's effective permissions (those he has, including permissions he inherited from Group/Role membership), `Required`, which is an ACL for some object, and, optionally, `Override`, which is a list of permissions that override object's ACL and allow user to bypass the ACL if he has these permissions.

Functions come in two overloaded versions, one accepting arrays of permission identifiers, and one accepting arrays of permission names for `Required` and `Override` lists (note that `Effective` list should always be an array of permission identifiers). Version with names lists allows for easy passing of fixed lists of permissions without the need to call `mapPerms` to map them to identifiers. `checkPermsN` functions return `INT` instead of `Boolean`, which allows to use them in SQL.

Functions take the most restrictive approach: everything that is not explicitly granted is denied. Function returns `TRUE` only if either `Required` or `Override` is neither `NULL` nor empty, and `Effective` permissions contain either all of `Required` permissions, or all of `Override` permissions, otherwise it returns `FALSE`. Function also returns `FALSE` if `Effective` list is empty or `NULL`. The following table illustrates the function outcome for various combinations of arguments (`MINUS` means set minus operation, which, when applied to two sets A and B as A `MINUS` B, removes all elements from set A that are also in set B):

`Effective`	`Required`	`Override`	Result
`NULL` or empty	Any	Any	`FALSE or 0`
Has some elements	`NULL` or empty	`NULL` or empty	`FALSE or 0`
Has some elements	Has some elements	`NULL` or empty	`TRUE or 1` if `Required MINUS Effective` is empty
Has some elements	`NULL` or empty	Has some elements	`TRUE or 1` if `Override MINUS Effective` is empty
Has some elements	Has some elements	Has some elements	`TRUE or 1` if (`Required MINUS Effective` is empty) or (`Override MINUS Effective` is empty)

This means, that to allow access to an object, one should assign some permissions to this object's ACL and then those users that have all these permissions or all `Override` permissions (if any) will be able to access the object. For example, to grant only certain group or role members access to an object, you can assign the group or role as object's owner and verify that requesting user has the same permissions as the owner of the object. Unauthenticated users (or those that don't have any permissions) are not allowed to access anything (you may override this behavior by adding your own rules to permission checks, like allow object owners to access their own objects without the need for any specific permissions assigned to them).

```
FUNCTION grantPerms(
                     nID        NUMBER,
                     Perms      NN$TPermList
                   ) RETURN NUMBER;
FUNCTION grantPerms(
                     nID        NUMBER,
                     Perms      NN$TPermNamesList
                   ) RETURN NUMBER;
```

This function grants specified `Perms` permissions to a User, Group or Role identified by `nID` identifier. It comes in two overloaded versions, one accepting array of permission identifiers, and another accepting array of permission names for `Perms` parameter. Function returns 0 if no permissions were granted, and > 0 if some or all permissions were granted (function does not re-grant permissions that were already granted with previous calls to this function). Function may also raise `NN$UMGR.ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges to grant permissions to other users, including self.

```
FUNCTION revokePerms(
                     nID        NUMBER,
                     Perms      NN$TPermList
                    ) RETURN NUMBER;
FUNCTION revokePerms(
                     nID        NUMBER,
                     Perms      NN$TPermNamesList
                    ) RETURN NUMBER;
```

This function is the opposite of `grantPerms` and revokes specified permissions from User, Role or Group identified by `nID`. Function returns 0 if no permissions were revoked, > 0 if some or all permissions were revoked, and may raise `NN$UMGR.ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges to revoke permissions from other users. Note that user cannot revoke system permissions from himself even if he has adequate privileges to revoke them in general – system privileges will be silently ignored in revoke request from self.

```
FUNCTION getPerms( nID    NUMBER
                 ) RETURN NN$TPermList;
```

This function retrieves and returns as nested table list of effective permissions for User, Group or Role identified by `nID`. This list includes permissions inherited from other Groups and Roles the requested entity is member of. Resulting list contains all distinct permissions the entity has at the time of the call (this list may change, for example, if a Role assigned to the entity is disabled in a concurrent transaction). If no permissions were found, function will return an empty array.

```
FUNCTION getVersion RETURN VARCHAR2;
```

This function returns current package version as `VARCHAR2` string.

EXAMPLES.

Below are some examples of package use.

```
declare
       nResult    NUMBER;
begin
  -- grant some permissions to a user
     nResult := NN$UMGR_ACL.grantPerms(NN$UMGR.getUserID('User'),
                                  NN$TPermNamesList('Perm1', 'Perm2') );
  -- verify user's effective permissions against required ACL
  if NN$UMGR_ACL.checkPerms(
      NN$UMGR_ACL.getPerms(NN$UMGR.getUserID('User1')),
      NN$TPermNamesList('Perm1', 'Perm2')
    ) then
    dbms_output.put_line('User has needed permissions.');
  else
    dbms_output.put_line('User DOES NOT have needed permissions.');
  end if;
  -- now verify current user's permissions using shortcut function
  -- myPermissions in NN$UMGR package
  if NN$UMGR_ACL.checkPerms(
      NN$UMGR.myPermissions,
      NN$TPermNamesList('Perm1', 'Perm2')
    ) then
    dbms_output.put_line('Current user has needed permissions.');
  else
    dbms_output.put_line('Current user DOES NOT have needed permissions.');
  end if;
end;
```

SECURITY CONSIDERATIONS.

This package does not control who grants and revokes permissions by default. Each Oracle user who wants to turn on internal security checks must perform special security bootstrap procedure. When this procedure is performed, several predefined permissions and roles are created, and only those users that have these special permissions are allowed to grant and revoke permissions to and from other users.

NN$UMGR PACKAGE.

This package is the main framework package. It provides functions and procedures for manipulating users, groups, roles and sessions.

VARIABLES.

```
G_TIMEOUT NUMBER := 1;
```

This variable defines session timeout in days. Default session timeout is 1 day. Developers may reinitialize this variable to another value before calling session-specific subprograms in the package to modify default timeout to a shorter or longer period of time.

```
G_VAL_GRACE_PERIOD NUMBER := 3;
```

This variable defines grace period in days for users in validation process while they still can login into system (to revalidate themselves for example). If a user is in validation process and validation was not performed within this period, the user will not be able to login until a user with administrative privileges will forcibly revalidate him. Default grace period is 3 days.

CONSTANTS.

```
G_US_DISABLED   CONSTANT NUMBER := 0; -- disabled
G_US_ACTIVE     CONSTANT NUMBER := 1; -- active
G_US_INVALID    CONSTANT NUMBER := 2; -- invalidated, pending validation
```

These constants define standard values for `STATUS` column of `NN$T$USERS` table.

EXCEPTIONS.

ACCESS_VIOLATION

This is generic Access Violation exception, which is raised if internal security is active and user attempts to perform an operation for which he has insufficient privileges. Internal security is inactive by default and is initialized by special security bootstrap procedure, which should be performed by each Oracle user using the framework separately. The bootstrap procedure creates some predefined permissions and groups, which are then used by the framework to perform internal security checks.

SUMMARY OF SUBPROGRAMS.

```
FUNCTION loginUser( p_login       VARCHAR2
                  ,p_password    VARCHAR2
                  ,b_reconnect   Boolean DEFAULT FALSE
                 ) RETURN VARCHAR2;
```

This function attempts to login given user and returns session ID if successful, or `NULL` otherwise. If `b_reconnect` is `TRUE`, the function also attempts to reconnect to an existing session associated with the user if one exists and is not timed out (equivalent to `initSession()` call). In addition, `NN$UMGR_CTX` context receives two values:

`CURRENT_USER` = logged in user ID

`CURRENT_SESSION` = current session ID (same as function return)

for future reference within Oracle session if login operation is successful.

```
FUNCTION initSession( p_session_id VARCHAR2
                     ,bSecure      Boolean  DEFAULT FALSE
```

```
                      ,p_password   VARCHAR2 DEFAULT NULL
                     ) RETURN Boolean;
```

This function attempts to re-establish the user session identified by given session ID. Returns `TRUE` if successful, `FALSE` if the session does not exist or is timed out. In addition, `NN$UMGR_CTX` context receives two values:

`CURRENT_USER` = logged in user ID

`CURRENT_SESSION` = current session ID (same as function return)

for future reference within Oracle session if operation is successful. Since session information is freely available through `NN$V$UMGR_SESSION` view and includes session user ID, this function may be used to impersonate a privileged user without having to authenticate. To prevent this, two optional parameters may be specified: `bSecure`, which turns on authentication on session reconnect, and `p_password`, which is checked against session user's password if `bSecure` is `TRUE`.

```
PROCEDURE logMeOff;
```

Logs off currently logged on user. `NN$UMGR_CTX` context is cleared and user's session is terminated (removed from the list of sessions). This procedure call is equivalent to `logoutSession( SYS_CONTEXT('NN$UMGR_CTX', 'CURRENT_SESSION'))` call.

```
FUNCTION logoutSession( p_session_id VARCHAR2
                      ) RETURN Boolean;
```

Terminates designated session. If `p_session_id` happens to be current session, then this function call is equivalent to `logMeOff` procedure. Returns `TRUE` if session was terminated, `FALSE` otherwise.

```
FUNCTION myPermissions RETURN NN$TPermList;
```

This function returns a nested table filled with effective permissions of currently logged in user or empty table if no user session is currently established. Effective permissions are a combination of permissions granted directly to user and those inherited through user's group and/or role membership. This function is equivalent to `NN$UMGR_ACL.getPerms( SYS_CONTEXT('NN$UMGR_CTX', 'CURRENT_USER') )` call.

```
FUNCTION changePassword( nID NUMBER
                        ,p_password VARCHAR2
                       ) RETURN Boolean;
```

This function attempts to assign new password to the user, returns `TRUE` if successful, `FALSE` otherwise (no such user). In addition, this function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user is not attempting to change his own password and does not have privileges to manage other users.

```
FUNCTION createUser( p_login         VARCHAR2
                    ,p_password      VARCHAR2
                    ,p_fname         VARCHAR2 DEFAULT NULL
                    ,p_mname         VARCHAR2 DEFAULT NULL
                    ,p_lname         VARCHAR2 DEFAULT NULL
                    ,p_description   VARCHAR2 DEFAULT 'User Account'
                   ) RETURN NUMBER;
```

This function creates new user account and returns newly created account ID or `NULL` if failed. `p_login` is unique user login name, `p_password` is user's password (case-insensitive), `p_fname` is user's first name, `p_mname` is user's middle name, `p_lname` is user's last name and `p_description` is verbose user account description. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION editUser( n_userID          NUMBER
                  ,p_fname           VARCHAR2 DEFAULT NULL
                  ,p_mname           VARCHAR2 DEFAULT NULL
                  ,p_lname           VARCHAR2 DEFAULT NULL
                  ,p_description     VARCHAR2 DEFAULT NULL
                 ) RETURN BOOLEAN;
```

This function attempts to update user information for given user ID and returns `TRUE` if successful, `FALSE` otherwise. In addition, the function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user is not modifying his own account and does not have adequate privileges to manage other users.

```
FUNCTION setUserStatus( n_UserID    NN$T$USERS.ID%TYPE
                       ,n_Status    NN$T$USERS.STATUS%TYPE
                      ) RETURN BOOLEAN;
```

This function changes user status. `n_UserID` is user identifier and `n_Status` should be one of `G_US` constants. If internal access control is active, only administrators may change user status.

```
FUNCTION createGroup( p_groupname    VARCHAR2
                     ,p_description  VARCHAR2 DEFAULT 'Group'
                    ) RETURN NUMBER;
```

This function attempts to create a new group and returns newly created group ID if successful, or `NULL` if failed. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges for managing users and groups. `p_groupname` is unique group name, `p_description` is verbose group description.

```
FUNCTION editGroup( n_GroupID        NUMBER
                   ,p_description    VARCHAR2 DEFAULT NULL
                  ) RETURN BOOLEAN;
```

This function attempts to modify description for the given group, and returns `TRUE` if successful, or `FALSE` otherwise. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges for managing users and groups. `n_GroupID` is group identifier and `p_description` is new description.

```
FUNCTION createRole( p_rolename      VARCHAR2
                    ,p_description   VARCHAR2 DEFAULT 'Role'
                   ) RETURN NUMBER;
```

This function creates a new role and returns newly create role ID, or `NULL` if failed. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges. `p_rolename` is new unique role name, and `p_description` is verbose role description.

```
FUNCTION editRole( n_RoleID          NUMBER
                  ,p_description     VARCHAR2 DEFAULT NULL
                  ) RETURN BOOLEAN;
```

This function attempts to change the role description and returns TRUE if successful, FALSE otherwise. The function may raise ACCESS_VIOLATION exception if internal security is active and calling user does not have adequate privileges. n_RoleID is role identifier and p_description is new description

```
FUNCTION getGroupID( p_groupname     VARCHAR2) RETURN NUMBER;
```

This function returns group identifier for the given group name.

```
FUNCTION getRoleID( p_rolename       VARCHAR2) RETURN NUMBER;
```

This function returns role identifier for the given role name.

```
FUNCTION getUserID( p_login       VARCHAR2) RETURN NUMBER;
```

This function returns user ID for the given user name (login).

```
FUNCTION getUserRecord( n_UserID NUMBER ) RETURN NN$V$USERS%ROWTYPE;
```

This function attempts to return user, group or role record for given user/group/role identifier, for example, urec := getUserRecord(TO_NUMBER(SYS_CONTEXT('NN$UMGR','CURRENT_USER'))). The function returns NULL if user record with specified identifier was not found. Throws standard exceptions if something goes wrong, except NO_DATA_FOUND, which is handled internally. This function is provided as a shortcut to selecting the user/group/role record from NN$V$USERS view.

```
FUNCTION addGroupMember( n_user_id   NUMBER
                        ,n_group_id  NUMBER
                        ) RETURN Boolean;
```

This function attempts to add a user or a group n_user_id to the group n_group_id and returns TRUE if successful, or FALSE if not. The function may raise ACCESS_VIOLATION exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION addRoleMember( n_user_id    NUMBER
                       ,n_role_id    NUMBER
                       ) RETURN Boolean;
```

This function attempts to add a user or a group n_user_id to the role n_role_id and returns TRUE if successful, or FALSE if not. The function may raise ACCESS_VIOLATION exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION dropGroupMember( n_user_id   NUMBER
                         ,n_group_id  NUMBER
                         ) RETURN Boolean;
```

This function attempts to remove a user or a group n_user_id from the group n_group_id and returns TRUE if successful, or FALSE if not. The function may raise ACCESS_VIOLATION exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION dropRoleMember(  n_user_id   NUMBER
                         ,n_role_id   NUMBER
                        ) RETURN Boolean;
```

This function attempts to remove a user or a group `n_user_id` from the role `n_role_id` and returns `TRUE` if successful, or `FALSE` if not. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION dropUser( n_UserID           NUMBER
                 ) RETURN Boolean;
```

This function attempts to drop specified user, returns `TRUE` if successful, `FALSE` otherwise. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges or when calling user attempts to drop himself.

```
FUNCTION disableRole( n_RoleID        NUMBER
                    ) RETURN BOOLEAN;
```

This function attempts to disable specified role, returns `TRUE` if successful, `FALSE` otherwise. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION enableRole( n_RoleID         NUMBER
                   ) RETURN BOOLEAN;
```

This function attempts to enable given role, returns `TRUE` if successful, `FALSE` otherwise. The function may raise `ACCESS_VIOLATION` exception if internal security is active and calling user does not have adequate privileges.

```
FUNCTION getValidationCode( n_UserID       NN$T$USERS.ID%TYPE
                           ,b_Invalidate   BOOLEAN DEFAULT TRUE
                   ) RETURN VARCHAR2;
```

This function initiates validation process for the user identified by `n_UserID`. `b_Invalidate` flag signals if user account status should be automatically set to `G_US_INVALID`. If internal access control is active, only administrators may start the validation process. The function returns random validation code the user should reply with to be re-validated. This code, when supplied to `validateUser()` function, will reactivate the user account. Validation codes are case-sensitive and should be presented to `validateUser()` as is with no case conversions. Validations initiated with this function call are tracked in `NN$V$VALIDATIONS` view. Only one incomplete validation can be active at one time. Function will return `NULL` if specified user does not exist, or current validation code for the user if the user is already in the validation process. The function may also throw `ACCESS_VIOLATION` exception if calling user does not have sufficient privileges to start the validation.

```
FUNCTION validateUser( n_UserID             NN$T$USERS.ID%TYPE
                      ,s_valCode            VARCHAR2
                      ,b_ForceValidation    BOOLEAN DEFAULT FALSE
                     ) RETURN BOOLEAN;
```

This function re-validates the user identified by `n_UserID` if `s_valCode` is correct. The user must be in grace period or `b_ForceValidation` should be `TRUE` for validation to take place. If the user is in grace period and the validation code is correct, user account is automatically enabled. If `b_ForceValidation` is specified, `s_valCode` is not checked, but calling user must have administrative privileges, otherwise `ACCESS_VIOLATION` exception will be thrown. Forced validation may be used by administrators to manually revalidate users who were unable to complete validation in time while administrator is certain that the user is still active.

```
FUNCTION inGracePeriod( n_UserID   NN$T$USERS.ID%TYPE
                      ) RETURN NUMBER;
```

This function returns 1 if user is invalidated and is in grace period, 0 otherwise. This function can be used in SQL.

```
FUNCTION getVersion RETURN VARCHAR2;
```

This function returns current package version as `VARCHAR2` string.

EXAMPLES.

The following example is an SQL*Plus script that can be used for testing NN$UMGR framework installation and functionality. It demonstrates all basic functions of the framework and their use.

```
set serveroutput on
set timing on

-- define NN$UMGR types schema – synonyms for types are not supported
-- in Oracle, thus we need to reference types with full qualification. Packages,
-- on the other hand, have public synonyms.
define typesch='nnumgr.'

set echo off

declare
  bdummy Boolean;
  ndummy number;
  sdummy varchar2(40);
  perms  &typesch.NN$TPermList;
begin
  dbms_output.enable(1000000);
  -- create two sample permissions
  ndummy := nn$umgr_perm.createPerm('sysadmin','Administer System');
  ndummy := nn$umgr_perm.createPerm('useradmin','Administer Users');
  -- create admin user
  ndummy := nn$umgr.createUser(
              'admin',
              'password',
              'System',
              '',
              'Administrator',
              'System Administrator Account');
  if ndummy is not null then
   dbms_output.put_line('User account #'||ndummy||' created');
  end if;
  -- create Administrators group
  ndummy := nn$umgr.createGroup('Administrators','System Administrators');
  if ndummy is not null then
   dbms_output.put_line('Group #'||ndummy||' created');
  end if;
  -- create UserAdmins group
  ndummy := nn$umgr.createGroup('UserAdmins','User Administrators');
  if ndummy is not null then
   dbms_output.put_line('Group #'||ndummy||' created');
  end if;
  -- grant some permissions to administrators group
  ndummy := nn$umgr_acl.grantPerms(nn$umgr.getGroupID('administrators'),

&typesch.NN$TPermNamesList('sysadmin','useradmin'));
  -- add admin to administrators group
  bdummy := nn$umgr.addGroupMember(nn$umgr.getUserID('admin'),
                                   nn$umgr.getGroupID('administrators'));
  if bdummy then
   dbms_output.put_line('addGroupMember succeeded.');
  end if;
  -- attempt to login as admin
  sdummy := nn$umgr.loginUser('admin','password');
  if sdummy is not null then
   dbms_output.put_line('User logged in, session id = '||sdummy||', effective
permissions:');
  -- print effective user permissions (note that we did not grant any
  -- permissions directly to admin user, only to administrators group
   perms := nn$umgr.myPermissions;
   if perms is not null and perms.count > 0 then
    for i in perms.First..perms.Last loop
      dbms_output.put_line(nn$umgr_perm.getPermName( perms(i) ) );
    end loop;
   else
     dbms_output.put_line('None.');
   end if;
  end if;
  -- verify effective permissions of current user against fixed set of
  -- permissions
  if nn$umgr_acl.checkPerms(nn$umgr.myPermissions,
                            &typesch.NN$TPermNamesList('useradmin'),
                            &typesch.NN$TPermNamesList('sysadmin')
                           ) then
   dbms_output.put_line('Current user can administer other users.');
  else
   dbms_output.put_line('Current user CAN NOT administer other users.');
  end if;
  -- log off
  nn$umgr.logMeOff;
  dbms_output.put_line('Effective permissions after logout:');
  -- check permissions after logout
  perms := nn$umgr.myPermissions;
  if perms is not null and perms.count > 0 then
   for i in perms.First..perms.Last loop
     dbms_output.put_line(nn$umgr_perm.getPermName( perms(i) ) );
   end loop;
  else
    dbms_output.put_line('None.');
  end if;
  if nn$umgr_acl.checkPerms(nn$umgr.myPermissions,
                            &typesch.NN$TPermNamesList('useradmin'),
                            &typesch.NN$TPermNamesList('sysadmin')
                                  ) then
   dbms_output.put_line('Current user can administer other users.');
  else
   dbms_output.put_line('Current user CAN NOT administer other users.');
  end if;
end;
/
```

The above example should produce the output similar to the following:

```
User account #31805710 created
Group #180344 created
Group #330135 created
addGroupMember succeeded.
User logged in, session id = 0B74B194AA504D53853DD3A99336C7EE, effective
permissions:
SYSADMIN
USERADMIN
Current user can administer other users.
Effective permissions after logout:
None.
```

```
Current user CAN NOT administer other users.

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.30
```

SECURITY CONSIDERATIONS.

This package does not control who creates, modifies and drops users, groups and roles by default. Each Oracle user who wants to turn on internal security checks must perform special security bootstrap procedure. When this procedure is performed, several predefined permissions and roles are created, and only those users that have these special permissions are allowed to manage other users, groups and roles.

EXTENDING THE FRAMEWORK.

This section will provide examples of extending the basic user management capabilities provided by the framework to support additional entity attributes, and using ACL API to control access to custom entities.

ADDING ATTRIBUTES TO NN$UMGR ENTITIES.

Entities managed by the framework (users, groups, roles and sessions) have only a limited basic set of attributes for each entity type. Developers may want to support additional attributes for these entities (for example, E-mail Address, Date of Birth, Sex and Marital Status attributes for user). The framework provides referential access to entity storage that provides means for linking additional attributes to framework-managed entities. The following example demonstrates adding E-mail Address, Date of Birth, Sex and Marital Status attributes to User entity.

```
CREATE TABLE USER_ATTRIBUTES(
 USER_ID    NUMBER(10,0) NOT NULL
,EMAIL      VARCHAR2(100)
,DOB        DATE
,SEX        CHAR(1) CHECK ( SEX IN (‘M’,’F’) )
,MARITAL    VARCHAR2(8) CHECK ( MARITAL IN (‘SINGLE’, ‘MARRIED’, ‘DIVORCED’) )
,CONSTRAINT UA$PK$USER_ID PRIMARY KEY (USER_ID)
,CONSTRAINT UA$FK$NN$T$USERS_ID FOREIGN KEY (USER_ID)
  REFERENCES NN$T$USERS (ID) ON DELETE CASCADE
)
/
```

The table definition above creates master-detail relationship between this table and `NN$T$USERS` table, for which public is granted `REFERENCES` privilege (that is, anyone is allowed to create foreign keys on this table and maintain relationship with it). Now, you can create new user and populate additional attributes this way:

```
DECLARE
  usr_id   NUMBER;
BEGIN
 usr_id := NN$UMGR.createUser(‘jdoe’,
                              ‘password’,
                              ‘John’,
                              ‘’,
                              ‘Doe’,
                              ‘John Doe account’);
 IF usr_id IS NOT NULL THEN
  INSERT INTO USER_ATTRIBUTES
   VALUES(usr_id,
          ‘jdoe@mycompany.com’,
          to_date(’14 Jun 1970’, ‘DD Mon YYYY’),
          ‘M’,
          ‘MARRIED’);
  COMMIT;
 END IF;
END;
```

If you subsequently drop the `jdoe` account using `dropUser` call, corresponding data in `USER_ATTRIBUTES` will be automatically deleted through `ON DELETE CASCADE` foreign key constraint on this table.

Using similar techniques you can extend NN$UMGR user sessions by creating foreign keys on `NN$T$UMGR_SESSION(ID, USER_ID)`, and link your own ACLs to framework-managed permissions by creating foreign keys on `NN$T$PERMISSIONS(ID)`.

Copyright© 2002-2003 by N-Networks. All rights reserved.

MAINTAINING AND CHECKING YOUR OWN ACCESS CONTROL LISTS.

ACLs managed by `NN$UMGR_ACL` package are only related to NN$UMGR entities and represent permissions granted to users, groups and roles. However, you can easily attach your own framework-compatible ACLs to your objects and use `NN$UMGR_ACL.checkPerms()` function to validate user privileges against these ACLs. This section provides an example of such custom ACL and ways to validate user privileges against it.

First, we create a table, which will hold objects (documents) we want to control access to:

```
CREATE TABLE DOCS ( DOC_ID     NUMBER(10,0) NOT NULL
                   ,DOC_NAME   VARCHAR2(200)
                   ,DOC_BODY   CLOB
                   ,DOC_OWNER  NUMBER(10,0) NOT NULL
                   ,CONSTRAINT DOCS$PK PRIMARY KEY (DOC_ID)
                   ,CONSTRAINT DOCS$FK$NN$T$USERS FOREIGN KEY (DOC_OWNER)
                     REFERENCES NN$T$USERS ON DELETE SET NULL
)
/
```

We defined relationship with `NN$T$USERS` table to know who's the document owner, and set referential integrity constraint to `ON DELETE SET NULL`, which means that if the row owner is dropped, the document will be left intact, but its owner information will be nullified (the document will be left stranded in the database, but it will not be removed). Next, we create an ACL table that will hold ACLs for our documents:

```
CREATE TABLE DOCS_ACL( DOC_ID   NUMBER(10,0) NOT NULL
                      ,PERM_ID  NUMBER(10,0) NOT NULL
                      ,CONSTRAINT DOCS_ACL$PK PRIMARY KEY (DOC_ID, PERM_ID)
                      ,CONSTRAINT DOCS_ACL$FK$DOCS FOREIGN KEY (DOC_ID)
                        REFERENCES DOCS(DOC_ID) ON DELETE CASCADE
                      ,CONSTRAINT DOCS_ACL$FK$PERMS FOREIGN KEY (PERM_ID)
                        REFERENCES NN$T$PERMISSIONS(ID) ON DELETE CASCADE
)
/
```

Here we defined relationships with `DOCS` table and `NN$T$PERMISSIONS` table, and entries in `DOCS_ACL` will be automatically deleted if parent document is deleted, or permission is dropped. Now it's time to create some utility subprograms that will help us add permissions to the ACL, remove them, and receive them into a `NN$TPermList` array to pass it to `NN$UMGR.checkPerms()` function for verification. We will create a package for this purpose. Assume that we have defined `DOCADMIN` permission, which allows user to modify ACLs for documents he doesn't own, and that NN$UMGR is installed into `NNUMGR` schema.

Package definition:

```
CREATE OR REPLACE PACKAGE DOCS$ACL
AS
PROCEDURE addPermsToACL(n_doc_id NUMBER, Perms NNUMGR.NN$TPermList);
PROCEDURE addPermsToACL(n_doc_id NUMBER, Perms NNUMGR.NN$TPermNamesList);
PROCEDURE removePermsFromACL( n_doc_id NUMBER, Perms NNUMGR.NN$TPermList);
PROCEDURE removePermsFromACL( n_doc_id NUMBER, Perms NNUMGR.NN$TPermNamesList);
FUNCTION getDocACL(n_doc_id NUMBER) RETURN NNUMGR.NN$TPermList;
FUNCTION canAccess( n_user_id NUMBER, n_doc_id NUMBER) RETURN Boolean;
END DOCS$ACL;
/
CREATE OR REPLACE PACKAGE BODY DOCS$ACL
AS
PROCEDURE addPermsToACL(n_doc_id NUMBER, Perms NNUMGR.NN$TPermList)
IS
-- perform addition in an autonomous transaction to avoid interference with
-- current transaction.
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  IF (Perms IS NULL OR Perms.Count = 0) THEN
```

```
    RETURN;
  END IF;
  DECLARE
    l_doc_owner NUMBER(10,0);
  BEGIN
   SELECT OWNER_ID INTO l_doc_owner FROM DOCS
    WHERE DOC_ID = n_doc_id;
   IF l_doc_owner != TO_NUMBER(SYS_CONTEXT('NN$UMGR_CTX', 'CURRENT_USER')) THEN
     -- user is not owner of the document
     IF NOT NN$UMGR_ACL.checkPerms(NN$UMGR.myPermissions,
                               NNUMGR.NN$TPermNamesList('DOCADMIN') ) THEN
        -- user doesn't have needed permissions
        -- to modify ACLs for other user's documents
        RETURN;
     END IF;
   END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      -- no such document
      RETURN;
  END;
  -- insert new permissions into the ACL table. FORALL bulk binding INSERT
  -- would be more effective here, but we also want to skip duplicates on
  -- insert – FORALL does not allow this.
  FOR i IN Perms.First..Perms.Last LOOP
   BEGIN
     INSERT INTO DOCS_ACL
       VALUES(n_doc_id, Perms(i));
   EXCEPTION
     -- ignore duplicates, we already have them granted
     WHEN DUP_VAL_ON_INDEX THEN NULL;
   END;
  END LOOP;
  -- commit our inserts
  COMMIT;
END addPermsToACL;

PROCEDURE addPermsToACL(n_doc_id NUMBER, Perms NNUMGR.NN$TPermNamesList)
IS
-- overloaded version which maps names to identifiers and calls original
BEGIN
 addPermsToACL(n_doc_id, NN$UMGR_PERM.mapPermNames(Perms));
END addPermsToACL;

PROCEDURE removePermsFromACL( n_doc_id NUMBER, Perms NNUMGR.NN$TPermList)
IS
-- perform deletion in an autonomous transaction to avoid interference with
-- current transaction.
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  IF (Perms IS NULL OR Perms.Count = 0) THEN
    RETURN;
  END IF;
  DECLARE
    l_doc_owner NUMBER(10,0);
  BEGIN
   SELECT OWNER_ID INTO l_doc_owner FROM DOCS
    WHERE DOC_ID = n_doc_id;
   IF l_doc_owner != TO_NUMBER(SYS_CONTEXT('NN$UMGR_CTX', 'CURRENT_USER')) THEN
```

```
     -- user is not owner of the document
     IF NOT NN$UMGR_ACL.checkPerms(NN$UMGR.myPermissions,
                               NNUMGR.NN$TPermNamesList('DOCADMIN') ) THEN
        -- user doesn't have needed permissions
        -- to modify ACLs for other user's documents
        RETURN;
     END IF;
   END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      -- no such document
      RETURN;
  END;
  -- do a bulk binding DELETE of all specified permissions
  FORALL i IN Perms.First..Perms.Last
   DELETE DOCS_ACL
    WHERE DOC_ID = n_doc_id
      AND PERM_ID = Perms(i);
  COMMIT;
END removePermsFromACL;

PROCEDURE removePermsFromACL( n_doc_id NUMBER, Perms NNUMGR.NN$TPermNamesList)
IS
-- overloaded version which maps names to identifiers and calls original
BEGIN
 removePermsFromACL(n_doc_id, NN$UMGR_PERM.mapPermNames(Perms));
END removePermsFromACL;

FUNCTION getDocACL(n_doc_id NUMBER) RETURN NNUMGR.NN$TPermList
IS
   rv NNUMGR.NN$TPermList := NNUMGR.NN$TPermList();
BEGIN
   SELECT PERM_ID
     BULK COLLECT INTO rv
     FROM DOCS_ACL
    WHERE DOC_ID = n_doc_id;
   -- note that if no document will match n_doc_id, this SELECT statement
   -- will not throw an exception, rather it will not fill the receiving
   -- nested table, so it will effectively be empty, which will be interpreted
   -- as 'no access'.
   RETURN rv;
END getDocACL;

FUNCTION canAccess(n_user_id NUMBER, n_doc_id NUMBER) RETURN Boolean
IS
   cnt NUMBER;
BEGIN
   IF (n_user_id IS NULL) THEN
     RETURN FALSE;
   END IF;
   -- check if n_user_id is owner of n_doc_id, which means he has access
   SELECT COUNT(1) INTO cnt FROM DOCS
    WHERE DOC_ID = n_doc_id
      AND OWNER_ID = n_user_id;
   IF (cnt = 0) THEN
      -- use checkPerms() to verify access, empty getDocACL result will be
      -- interpreted as 'no access'
      RETURN NN$UMGR_ACL.checkPerms(
                NN$UMGR_ACL.getPerms(n_user_id), -- n_user_id's effective perms
                getDocACL(n_doc_id) -- n_doc_id’s ACL
                );
   ELSE
      -- n_user_id is owner of n_doc_id, so it’s ok to access it
      RETURN TRUE;
   END IF;

END canAccess;

END DOCS$ACL;
/
```

Here we have our utility package. It has everything necessary to manage ACLs for documents in DOCS plus utility functions to verify access rights to them. Now, if we have a user logged in, we can verify if he can access certain document by calling our verification function this way:

```
IF DOCS$ACL.canAccess(TO_NUMBER(SYS_CONTEXT(‘NN$UMGR_CTX’,‘CURRENT_USER’)),
some_doc_id) THEN
 -- proceed with document access
ELSE
  RAISE_APPLICATION_ERROR(-20001,
    ‘Access denied: sorry, you do not have privileges to access document #’||
    to_char(some_doc_id) );
END IF;
```

If, for example, a document belongs to user JDOE, and user JDOE required, say, READ permission to access it by adding this permission to the document’s ACL, the document will be seen only to this user JDOE (even if he doesn’t have READ himself) and those users having READ permission in their permission list (either granted directly or inherited through group/role membership).