MQSeries® Integrator

Programming Reference for NEONRules™

Version 1.0

SC34-5506-00

Note: Before using this information and the product it supports, be sure to read the general information under Appendix B entitled "Notices".

First edition (January 1999)

This edition applies to IBM® MQSeries Integrator, Version 1.0 and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

At the back of this publication is a page titled "Sending your comments to IBM". If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories
Information Development,
Mail Point 095,
Hursley Park,
Winchester,
Hampshire,
England,
SO21 2JN.

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

Contents

Chapter 1

Introduction

The ***MQSeries Integrator Programming Reference for NEONRules*** provides descriptions and examples for each function in the Rules and Rules Management APIs.

This document is divided into two main sections: Rules APIs and Rules Management APIs.

Product Documentation Set

The MQSeries Integrator documentation set includes:

- ***MQSeries Integrator Installation and Configuration Guide*** helps end users and engineers install and configure MQSeries Integrator.
- ***MQSeries Integrator User's Guide*** helps MQSeries Integrator users understand and apply the program through its graphical user interfaces (GUIs).
- ***MQSeries Integrator System Management Guide*** is for system administrators and database administrators who work with MQSeries Integrator on a day-to-day basis.
- ***MQSeries Integrator Application Development Guide*** assists programmers in writing applications that use MQSeries Integrator APIs.
- ***Programming References*** are intended for users who build and maintain the links between MQSeries Integrator and other applications. The documents include:
  - ***MQSeries Integrator Programming Reference for NEONFormatter*** is a reference to Formatter APIs for those who write applications to translate messages from one format to another.
  - ***MQSeries Integrator Programming Reference for NEONRules*** is a reference to Rules APIs for those who write applications to perform actions based on message contents.

Note

For information on message queuing, refer to the ***IBM MQSeries*** documentation.

Documentation Conventions

Tip

Tips point out shortcuts or procedures that can help you use MQSeries Integrator more effectively.

Note

Notes point out useful extra information.

WARNING!

Do not ignore anything associated with a warning—it alerts you to something that can cause loss of, or damage to your work.

Supported Platforms and Compilers

Operating System	DBMS	Compiler
Windows NT 4.0	DB2 5.0 Oracle 7.3 Oracle 8 SQL Server 6.5 Sybase Client 11.1.1 Sybase Server 11.03, 11.5	Microsoft Visual C++ version 4.2
Solaris 2.5.1, 2.6	DB2 5.0 Oracle 7.3 Sybase Client 11.1.1 Sybase Server 11.03, 11.5	Sparcworks C++ compiler version 4.0
HP-UX 10.20	DB2 5.0 Oracle 7.3 Oracle 8 Sybase Client 11.1.1 Sybase Server 11.03, 11.5	HP C++ version 10.34
AIX 4.2	DB2 5.0 Oracle 7.3 Sybase Client 11.1.1 Sybase Server 11.03, 11.5	IBM C Set ++ version 3.1.4

Chapter 2 Rules Overview

NEONRules enables you to evaluate a string of data (message) and react to the evaluation results. The following overview describes Rules components and what types of APIs are available for rule processing.

Application Groups

Application groups are logical divisions of rule sets for different business needs. You can define as many application groups as you need. For instance, you might want rules for the accounting department and the application development department separated into two groups. You could define "Accounting" as one application group, "Application Development" as another, and then associate rules with each group as appropriate.

Message Types

Message types define the layout of a string of data. Each application group can contain several message types, and a message type can be used with more than one application group. Message types are defined by the user. When using MQSeries Integrator Formatter, a message type is the same as an input format name. This format name is used by the Formatter to parse input messages for Rules evaluation.

Rules

When users create rules, they give each rule a rule name and associate the rule name with an application group and message type. Each rule is uniquely identified by its application group / message type / rule name triplet.

Each rule must have the following three items defined: evaluation criteria (an expression containing arguments and operators), subscription information (subscriptions, actions, and options), and permission information. Each of these is described below.

Expressions, Arguments, Boolean Operators, and Rules Operators

An expression (evaluation criteria) consists of a list of fields, associated operators, and associated comparison data (either static values or other fields) connected with Boolean operators. An argument consists of the combination of a field name, Rules comparison operator, and static value or other field name. Field names depend on the message type (input format name) and they are defined using Formatter. Rules comparison operators are already defined within Rules. Field comparisons can be made against static data or other field values. Arguments are linked together with Boolean operators '&' (AND) and '|' (OR) and parentheses can be used to set the evaluation priority.

Subscriptions, Actions, and Options

When a rule evaluates to true, it is considered a "hit." If the rule does not evaluate to true, it is considered a "no-hit." When a rule hits, Rules lets you retrieve associated subscriptions to be taken by the application. These subscriptions are the actions or commands, and the associated parameters or options to execute them.

Subscriptions are lists of actions to take when a message evaluates to true. Each rule must have at least one associated subscription. Subscriptions are uniquely identified within an application group / message type pair by a user-defined subscription name. Permissions must be defined for subscriptions as for rules. You can define as many subscriptions as you need. Each action within a subscription is defined by action name and need not be unique since all actions are intended to be executed in sequence. A single subscription can be “shared” by multiple rules where the same subscription is associated with each of the rules. In this case, the shared subscription would be retrieved only once no matter how many of its rules hit.

An action has a list of one or more associated options. An option consists of an option name-value pair. The user defines all action names and option name-value pairs.

Rules/Subscription Permissions

Rule and Subscription permissions restrict user access to individual complete rules or subscriptions or their components in the Rules database. The rule is uniquely identified by its application group name, message type, and rule name. A complete rule includes everything associated with it, including an expression (arguments) and associated subscriptions. The subscription is uniquely defined by its application group name, message type, and subscription name. A complete subscription includes everything associated with it including its actions and options. Permissions only apply to managing rule and subscription contents, not rule evaluation.

The Rules component (rule or subscription) or subscription owner is the user who created the component. When the rule or subscription is created, owner information is determined by the software. Owners can update their own permissions, create and update the PUBLIC user's permissions, and change ownership to another user.

Only read and update permissions are implemented. The owner is given both read and update permission by default. All other users are grouped into a public user group named PUBLIC and given read permissions by default.

Note

Owners can change their own permissions at any time from read to update and back again, but they must have update permissions to change a rule or subscription contents. Read permission cannot be denied.

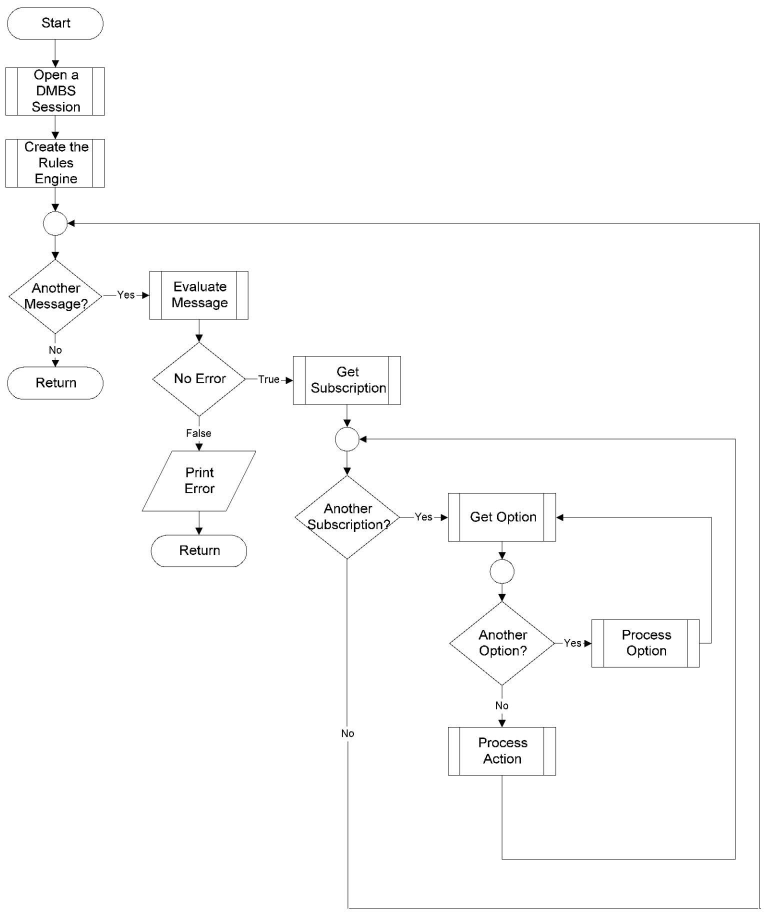
APIs

Two types of APIs exist for Rules: Rules APIs and Rules Management APIs.

Use Rules APIs to evaluate rules and retrieve subscription, hit, and no-hit information. Before you evaluate a rule, the rule must exist and you must use CreateRulesEngine() to create a VRule object. After that, you can do as many evaluations and subscription retrievals as needed. When you finish, destroy the MQSeries Integrator Rules daemon object using DeleteRuleEngine().

Use Rules Management APIs to maintain rule information. Add, Read, and Update APIs are implemented and available as well as APIs to delete an entire rule or subscription and all their associated information.

Flow of Calls



Suggested Flow of Calls for Rules Processing

Using eval(), Rules processing evaluates rules by taking in a text message and the definitions of the rule set (application group / message type). The user then retrieves the list of user actions with their parameters (options) that should be performed based on the rules that evaluated true for the message. These actions and options are retrieved by calling getsubscription() and getopt() in nested loops.

Open the DBMS Session:

```
DbmsSession *RulesSession = OpenDbmsSession
(RulesSessionName, DB-Identifier);
```

Create the Rules engine:

```
VRule *rules = CreateRulesEngine(RulesSession);
```

```
For each Message
    Evaluate Message against the Rule Set::
     if (!rules->eval(appname, msgname, msg, msglen) )
            Get the error number and print it:
               Print (GetRerror(rules->GetErrorNo())) or
               Print (rules->GetErrorMessage())
            else
               For each Subscription
                  while ( (pAct = rules->getsubscription()))
```

Note

This gets the next action associated with this subscription and removes it from the list of subscriptions to execute. You must differentiate between subscription boundaries by performing any initialization associated with a new subscription prior to getting the next subscription, including saving the SubId field from the SUBSCRIPTION structure. This SubId field should be compared to the saved SubId field each time an action is retrieved to determine when a new subscription has been reached.

Now, the SUBSCRIPTION structure is populated.

```
            For each Option
               while ( (popt = rules->getopt()) )
```

Note

This gets all of the options associated with this subscription. Looping terminates when the next option is NULL.

The OPTIONPAIR structure is populated each time the getopt function is called and will be overwritten the next time getopt is called. The user must save and/or process the options associated with a given action prior to retrieving the next option.

Thread-Safe Rule Evaluation

When a function is "thread-safe," that function may be called by one or more threads without adversely affecting the data in each thread. Functions executing in multiple threads synchronize themselves as appropriate behind the scenes.

Global resources for a process, such as globally allocated memory and files, get shared by all threads for that process. Access to those resources must be regulated to keep them in a consistent state when routines in the library are entered by different threads at the same time.

Non-Threaded Environment

The Suggested Flow of Calls for Rules Processing discussion in the previous section describes how to evaluate and retrieve results for messages run against a set of rules in a non-threaded environment.

The general algorithm resembles the following pseudocode:

```
Instantiate an instance of the DbmsSession class to open a database session.
Instantiate an instance of the Rules Engine, passing it the DbmsSession instance.
When you want to do evaluations:
    Retrieve the message, application group, and message type for evaluation.
    Evaluate the message against the rules described by the application group/message type pair
    If the evaluation succeeds,
        [You can call gethitrule() and getnohitrule() in separate lists to retrieve lists of rules here.]
        While there are subscriptions to retrieve...
            [Do something based on the actions for this subscription.]
            While there are options for an action...
                [Do something with the options for this action.]
            end While
        end While
    Else
        Get the error that occurred.
    [Continue doing evaluations.]
end While
Destroy the instance of the Rules Engine.
Close the database session.
```

Multi-Threaded Environment

To evaluate messages concurrently, thread-safe Rules APIs can be called in a multi-threaded environment. Only the Rules Engine and Formatter APIs are thread-safe.

However, note that one thread cannot call any of the following APIs to retrieve the results of an evaluation done by another thread, since each thread only has access to its own evaluation results:

- gethitrule()
- getnohitrule()
- getsubscription()
- getaction()
- getoption()
- GetErrorNo()
- GetErrorMessage()

The general algorithm in a multi-threaded environment resembles the following pseudocode:

Thread A:

```
Instantiate an instance of the DbmsSession class to open a database session.
Instantiate an instance of the Rules Engine, passing it the DbmsSession
instance.
```

Thread A, B, C - all do the same:

```
While you want to do evaluations...
    Retrieve the message, application group, and message type for
    evaluation.
    Evaluate the message against the rules described by the application
    group / message type pair
    If the evaluation succeeds,
        [You can call gethitrule() and getnohitrule() in separate lists to
        retrieve lists of rules here.]
        While there are subscriptions to retrieve...
            [Do something based on the actions for this subscription.]
            While there are options for an action...
                [Do something with the options for this action.]
            end While
        end While
    Else
        Get the error that occurred.
    [Continue doing evaluations.]
end While
```

Thread B, C:

[When done evaluating messages in threads B and C, destroy the threads.]

Call VRule::ThreadCleanup()

[Exit thread]

Thread A:

Destroy the instance of the Rules Engine.

Close the database session.

APIs and Header Files

The Rules API is made up of classes of objects that have member functions:

Header Files

Object Class	Header File	Description
VRule	vrule.h	Rules Processing APIs
NNRMgr	nnrmgr.h	Rules Management APIs
—	ruleuser.h	Subscription Structures
—	nnrmerr.h	Rules Management Error
—	rerror.h	Rules Error Handling

VRule Member Functions

Return Type	Function	Arguments
VRule *	CreateRulesEngine	(DbmsSession *Session)
VRule *	CreateRulesEngine	(DbmsSession* Session, int alert=1, char *logfile=NULL)
void	DeleteRuleEngine	(VRule * pEngine)

Return Type	Function	Arguments
int	eval	(char *AppName, char*Msg Name, char *msg, int msglen, int log=0)
RULE*	gethitrule	None
RULE*	getnohitrule	None
FORMATTER*	getformatterobject	None
char*	getaction †	None
OPTIONPAIR*	getopt	None
void	ThreadCleanup	None
char *	getlog	None
int	GetErrorNo	None
int	ReloadRuleSet	(char *AppGrp, char*MsgType, int LoadNow=0)
char *	GetErrorMessage	None

Rules Error Handling Function

Return Type	Function	Arguments	Notes
char*	GetRerror †	(int ErrorNo)	Use VRules:GetError Message instead.

† These functions are used for backward compatibility. We recommend using other functions to perform the desired action.

Rules Management Functions and Macros

Return Type	Function	Arguments
NNRMgr *	NNRMgrInit	(DbmsSession *session)
void	NNRMgrClose	(NNRMgr *pMgr)
N/A	NNR_CLEAR	(_p)
N/A	NN_CLEAR	(_p)
const long	NNRMgrAddOwner Permission	(NNRMgr *pMgr, const NNRRule *pRRule, const NNPermissionData *pPermissionData)
const long	NNRMgrAddOther UserPermission †	(NNRMgr *pMgr, const NNRRule *pRRule, const NNPermissionData *pPermissionData)

Return Type	Function	Arguments
const long	NNRMgrGetFirst RulePermission †	(NNRMgr *pMgr, const NNRRule *pRRule, NNUserPermissionData* const pPermissionData)
const long	NNRMgrGetNext RulePermission †	(NNRMgr *pMgr, const NNUserPermissionData *pPermissionData)
const long	NNRMgrUpdate Owner †	(NNRMgr *pMgr, const NNRRule *pRRule, char *pNewOwner)
const long	NNRMgrUpdate OwnerPermission †	(NNRMgr *pMgr, const NNRRule *pRRule, const NNPermissionData *pPermissionData)
const long	NNRMgrUpdate OtherUserPermission †	(NNRMgr *pMgr, const NNRRule *pRRule, const NNPermissionData *pPermissionData)
const long	NNRMgrAddApp	(NNRMgr *pMgr, const NNRApp *pRApp, const NNRAppData *pRAppData)
const long	NNRMgrReadApp	(NNRMgr *pMgr, NNRApp *pRApp, NNRAppData *pRAppData)
const long	NNRMgrUpdateApp	(NNRMgr *pMgr, const NNRApp *pRApp, const NNRAppUpdate *pRAppUpdate)
const long	NNRMgrAddMsg	(NNRMgr *pMgr, NNRMsg *pRMsg, NNRMsgData *pRMsgData)
const long	NNRMgrReadMsg	(NNRMgr *pMgr, NNRMsg *pRMsg, NNRMsgData *pRMsgData)
const long	NNRMgrAddRule	(NNRMgr *pMgr, NNRRule *pRRule, NNRRuleData *pRRuleData)
const long	NNRMgrReadRule	(NNRMgr *pMgr, NNRRule *pRRule, NNRRuleData *pRRuleData)
const long	NNRMgrUpdateRule	(NNRMgr *pMgr, const NNRRule *pRule, const NNRRuleUpdate *pRRuleUpdate)
const long	NNRMgrDelete EntireRule	(NNRMgr *pMgr, const NNRRule *pRRule)

Return Type	Function	Arguments
const long	NNRMgrGetFirst Rule	(NNRMgr *pMgr, const NNRRule *pRRule, NNRRuleReadData * const pRRuleData)
const long	NNRMgrGetNext Rule	(NNRMgr *pMgr, NNRRuleReadData * const pRRuleData)
const long	NNRMgrGetFirst Operator	(NNRMgr *pMgr, NNROperator * const pOperator)
const long	NNRMgrGetNext Operator	(NNRMgr *pMgr, NNROperator * const pOperator)
const long	NNRMgrAdd Argument	(NNRMgr *pMgr, NNRArg *pRArg, NRArgData *pRArgData)
const long	NNRMgrUpdate Argument	(NNRMgr *pMgr, NNRArg * const pRArg, NNRArgUpdate *pRArgUpdate, int position)
const long	NNRMgrGetFirst Argument	(NNRMgr *pMgr, NNRArg * const pRArg, NNRArgData * const pRArgData)
const long	NNRMgrGetNext Argument	(NNRMgr *pMgr, NNRArg * const pRArg, NNRArgData * const pRArgData)
const long	NNRMgrAdd Subscription	(NNRMgr *pMgr, NNRSubs *pRSubs, NNRSubsData *pRSubsData)
const long	NNRMgrRead Subscription	(NNRMgr *pMgr, NNRSubs *pRSubs, NNRSubsData *pRSubsData)
const long	NNRMgrUpdat eSubscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, const NNRSubsUpdate *pRSubsUpdate)
const long	NNRMgrGetFirst Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, NNRSubsReadData * const pRSubsReadData)
const long	NNRMgrGetNext Subscription	(NNRMgr *pMgr, NNRSubsReadData * const pRSubsReadData)
const long	NNRMgrAddAction	(NNRMgr *pMgr, NNRAction *pRAction, NNRActionData *pRActionData, int *pActionId)

Return Type	Function	Arguments
const long	NNRMgrUpdate Action	(NNRMgr *pMgr, const NNRAction *pRAction, const NNRActionUpdate *pRActionUpdate, int position)
const long	NNRMgrResequence Action	(NNRMgr *pMgr, const NNRAction *pRAction, int oldPosition, int newPosition)
const long	NNRMgrGetFirst Action	(NNRMgr *pMgr, NNRAction * const pRAction, NNRActionReadData * const pRActionData)
const long	NNRMgrGetNext Action	(NNRMgr *pMgr, NNRAction * const pRAction, NNRActionReadData * const pRActionData)
const long	NNRMgrAddOption	(NNRMgr *session, NNROption *pROption, NNROptionData *pROptionData)
const long	NNRMgrUpdate Option	(NNRMgr *pMgr, const NNROption *pROption, const NNROptionUpdate *pROptionUpdate, int position)
const long	NNRMgrResequence Option	(NNRMgr *pMgr, const NNROptionUpdate *pROptionUpdate, int oldPosition, int newPosition)
const long	NNRMgrGetFirst Option	(NNRMgr *pMgr, NNROption * const pROption, NNROptionReadData * const pROptionData)
const long	NNRMgrGetNext Option	(NNRMgr *pMgr, NNROption * const pROption, NNROptionReadData * const pROptionData)
const long	NNRMgrAddOwner Perm	(NNRMgr *pRMgr const NNRComponent, *pRComponent, const NNPermissionData *pPermissionData)
const long	NNRMgrAddPublic Perm	(NNRMgr *pRMgr const NNRComponent, *pRComponent, const NNPermission Data *pPermissionData)

Return Type	Function	Arguments
const long	NNRMgrGetFirst Perm	(NNRMgr *pRMgr, const, NNRComponent * pRComponent, NNUserPermissionData const * pPermissionData)
const long	NNRMgrGetNext Perm	(NNRMgr *pRMgr, NNUserPermissionData const * pPermissionData)
const long	NNRMgrChange Owner	(NNRMgr *pRMgr, const NNRComponent * pRComponent, char *pNewOwner)
const long	NNRMgrUpdate OwnerPerm	(NNRMgr *pRMgr, const NNRComponent * pRComponent, const NNPermission Data * pPermission Data)
const long	NNRMgrUpdate PublicPerm	(NNRMgr *pRMgr const NNRComponent * pRComponent, const NNPermission Data * pPermission Update)
const int	NNRMgrGetErrorNo	(NNRMgr *pRMgr)
const char *	NNRMgrGetError Message	(NNRMgr *pRMgr)

Libraries

Rules APIs must be linked with the following libraries:

Link Libraries for Rules APIs

Library	Description
libformat	Dynamic Formatter Library
libntools	Generic Tool Set
librmgr	Rules Manager Library
librules	Rules Library
libNoQsqlobj	SQL Object Interface Library
—	System / Compiler-Specific Libraries
—	Database Dependent Libraries

Note

For MQSeries installations, link with libmqs, libinterop, and libMQSruleng.

Note

Library file extensions are ".a" for UNIX and ".lib" for NT.

Note

THREAD SAFETY NOTE: For multithreading, you must also link with the appropriate thread library matching the MQSeries Integrator release. For example, link with the thread library for UI threads, pthread for POSIX threads, and so on.

Chapter 3

Rules APIs

This chapter details both Supporting and Member Functions for Rules.

Class/Type Definitions

VRule

Overview

A VRule object is a MQSeries Integrator Virtual Rules Engine instance. This class provides a standard interface for handling MQSeries Integrator Rules API calls and allows the user to perform all MQSeries Integrator rule evaluation and subscription retrieval. A VRule object is created using CreateRulesEngine() and deleted by DeleteRuleEngine().

Syntax

```
class VRule {
  public:
    VRule(){}
    virtual ~VRule();
    virtual int GetErrorNo() = 0;
    virtual int eval(char * AppName, char * MsgName,
        char * msg, int msglen, int log=0) = 0;
    virtual int eval (char * MsgName,
       Formatter * formatter, int log=0) = 0;
    virtual char * getaction() = 0;
    virtual SUBSCRIPTION * getsubscription() = 0;
    virtual OPTIONPAIR * getopt() = 0;
    virtual RULE * gethitrule() = 0;
    virtual RULE * getnohitrule() = 0;
    virtual char * getlog() = 0;
    virtual char * GetErrorMessage() = 0;
    virtual void ThreadCleanup() = 0;
    virtual int LoadRuleSet(char* AppGrp, char* MsgType,
                          int LoadNow = 0) = 0;
    Virtual Formatter *getformatterobject() = 0;

    };
```

SUBSCRIPTION

Overview

Each rule has an associated list of subscriptions, and each subscription has an associated list of one or more actions. The list of actions for a subscription is a list of SUBSCRIPTION structures.

When stepping through the list of actions for a specific subscription, the presence of a new subscription identifier (long SubId) signifies that a new subscription has been reached and that the action is the first associated with the new subscription.

Syntax

```
struct SUBSCRIPTION{
    long SubId;
    char * action;
}; char *SubName;
```

Parameters

Name	Type	Description
SubId	long	Subscription sequence identifier
SubName	char*	Subscription name
action	char*	Action name

Example

The following code fragment illustrates stepping through a list of actions:

```
while ((p=rules->getsubscription()){
    if ( strcmp(p->action,"my_fun1" ) == 0){
        my_fun1();
    }else if ( strcmp(p->action,"my_fun2") == 0 ){
        my_fun2();
    }else{
        //perform logging or exception handling
 }
}
```

OPTIONPAIR

Overview

Each rule has an associated list of subscriptions and each subscription has a list of one or more actions. Actions are intended to be executed in sequence and each action may have one or more associated option name-value pairs.

Option name-value pairs are OPTIONPAIR structures. An option pair may be unique to an action. A NULL OPTIONPAIR in a subscription's option list signifies the end of the options for that subscription action.

Syntax

```
struct OPTIONPAIR{
    int Sequence;
    char * Name;
    char * Value;
};
```

Parameters

Name	Type	Description
Sequence	int	Sequence Identifier.
Name	char*	Option name.
Value	char*	Option value.

Example

The following code segment illustrates walking through a list of options. Note that the presence of a NULL popt signifies the end of the list of options.

```
while ((popt=rules->getopt()){
    if ( strcmp(popt->Name,"Command_Argument1") == 0 ){
        pCommand_Argument1 = strdup(popt->Value);
    }
    if ( strcmp(popt->Name,"Command_Argument2") == 0 ){
        pCommand_Argument2 = strdup(popt->Value);
    }
}

if ( pCommand_Argument1 && pCommand_Argument2 ){
    my_fun1(pCommand_Argument1,pCommand_Argument2 );
}else {
    //error handling for missing options to my call
}
```

RULE

Overview

gethitrule() and getnohitrule() return records of rule information contained in a RULE structure.

Syntax

```
struct RULE{
    int RuleId;
    char *RuleName;
};
```

Parameters

Name	Type	Description
RuleId	int	Rule identifier.
RuleName	char*	Rule name.

Example

The following code fragment describes how to walk through lists of “hit” and “no hit” rules.

```
RULE *r;
cout << "NO HIT RULES" << endl;
while ( (r=rules->getnohitrule())){
    cout << "    " << r->RuleName << endl;
}
cout << "HIT RULES" << endl;
while ( (r = rules->gethitrule())){
    cout << "    " << r->RuleName << endl;
}
```

VRule Supporting Functions

To use Rules APIs, you must include the following header files located in the MQSeries Integrator include directory:

- dbtypes.h
- ses.h
- sqlapi.h
- rerror.h
- ruleuser.h
- vrule.h

Also, you must link with the following libraries in the MQSeries Integrator library directory:

- libformat.a
- librules.a
- libNoQsqlobj.a

Note

THREAD SAFETY NOTE: For multithreading, you must also link with the appropriate thread library matching the MQSeries Integrator release. For example, link with the thread library for UI threads, pthread for POSIX threads, and so on.

CreateRulesEngine

Overview

Note

THREAD SAFETY NOTE: For multi-threaded applications, OpenDbmsSession() and CreateRulesEngine() should only be called by the main thread. The VRule pointer returned by CreateRulesEngine() should then be passed to separate threads so that each thread can perform evaluations in parallel.

DeleteRuleEngine() should be called by the main thread only after ALL threads are done with evaluations.

Syntax#1

```
VRule* CreateRulesEngine(DbmsSession* Session);
```

Description

CreateRulesEngine() creates a VRule object for the MQSeries Integrator session provided in the session parameter. By default, errors are sent through the NNAlert mechanism (see Failure Processing in the ***MQSeries Integrator System Management Guide)***.

Parameters

Name	Type	Input/ Output	Description
Session	DbmsSession *	Input	Name of the currently open MQSeries Integrator session.

Syntax#2

```
VRule* CreateRulesEngine(DbmsSession* Session,
                      int alert=1,
                      char *logfile=NULL);
```

Description

CreateRulesEngine() creates a VRule object for the MQSeries Integrator session provided in the session parameter and enables the user to specify whether alerts should be sent to the NNAlert mechanism or to a log file.

Parameters

Name	Type	Input/ Output	Description
Session	DbmsSession *	Input	Name of the currently open MQSeries Integrator session.
alert	int	Input	True(1)/False (zero(0) option determining whether or not to send errors through the alert mechanism. Defaults to True (1).
logfile	char *	Input	File to log errors to instead of sending them through the NNAlert mechanism. Only valid if alert is True (1). Defaults to no file (NULL).

Remarks

CreateRulesEngine() must be called prior to doing any rules processing and prior to calling DeleteRuleEngine().

Return Value

Returns a VRule object if successful; NULL on failure. All error handling of a failed call to CreateRulesEngine() must be done by the code that calls this API.

Example#1

```
DbmsSession *session = OpenDbmsSession("fred", DbType);
    if (!session || !session->Ok()){
        cout << "Failed to open rules database session" <<
            endl;
        exit(1);
    }
    VRule *rule = CreateRulesEngine(session);
    if (!rule)
        cout << "Error no rules engine created" << end1;
```

Example#2

```
DbmsSession *session = OpenDbmsSession("fred", DbType);
    if (!session || !session->Ok()){
        cout << "Failed to open rules database session" <<
            endl;
        exit(1);
    }
VRule *rule =
    CreateRulesEngine(session,1,"rerrlog.log");
    if (!rule)
        cout << "Error no rules engine created" < end1;
```

See Also

DeleteRuleEngine()

DeleteRuleEngine

Overview

DeleteRuleEngine() cleans up a VRule object created using the CreateRulesEngine() function.

Note

THREAD SAFETY NOTE: For multi-threaded applications, DeleteRuleEngine() should only be called by the main thread after ALL threads are done with evaluations.

Syntax

```
void DeleteRuleEngine(VRule * pEngine);
```

Parameters

Name	Type	Input/ Output	Description
pEngine	VRule*	Input	Name of the currently open VRule object. See CreateRulesEngine().

Remarks

DeleteRuleEngine() must be called after CreateRulesEngine() and after all MQSeries Integrator rules processing is complete.

Return Value

None

There are no error handling functions for DeleteRuleEngine().

Example

```
DbmsSession *session = OpenDbmsSession("fred", DbType);
if (!session || !session->Ok()) {
    cout << "Failed to open session" << endl;
    exit(1);
}
Vrule *rule = CreateRulesEngine(session);
if (!rule) {
    cout << "Unable to create rules object" << endl;
    exit(2);
}
char MessageString[65];
memset(MyMessageString, 0, 65);
strcpy(MyMessageString, "Field1|Field2,Field3");
if ( !rule->eval("MyAppGroup", "MyMessageType",
MyMessageString,
strlen(MyMessageString)) ){
```

```
cout << "Failure" << endl;
exit(3);
}
if (rule){
    DeleteRuleEngine(rule);
}
if (session){
    CloseDbmsSession(session);
}
```

See Also

CreateRulesEngine()

VRule Member Functions

eval

Overview

Using the application group and message type, eval() retrieves all associated active rules, then parses the message into fields and evaluates those fields based on the evaluation criteria.

Syntax

```
int VRule::eval(char* AppName,
               char* MsgName,
               char* msg,
               int msglen,
               int log=0);
```

Parameters

Name	Type	Input/ Output	Description
AppName	char*	Input	Application Group Name. Should be the Application Group in which the user defined rules for evaluating this message. This string should not be empty.

Name	Type	Input/ Output	Description
MsgName	char*	Input	Type of message to be evaluated. If Formatter is used, message type is the input format name. This name should be the message type in which the user defined rules for evaluating this message. This string should not be empty.
msg	char*	Input	String containing the message to be evaluated. This message should be in the format expected by the message type. The string should not be empty.
msglen	int	Input	Message length, in bytes, of the message to be evaluated. msglen should be greater than zero (0).
log	int	Input	For increased logging capability in a future release, log defaults to zero (0) for now.

Remarks

eval() should be called after CreateRulesEngine() and before DeleteRuleEngine(). In addition, eval() should be called prior to returning subscriptions or hit/no-hit rules.

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to retrieve subscriptions, actions, and options from the same thread as the one that called eval().

Return Value

Returns 1 if the rules evaluated completely, regardless of the outcome; zero (0) if the evaluation failed.

Note that a successful evaluation does not imply that a rule "fired," only that all rules associated with the application group and message type were evaluated against the message completely.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

```
if (!rules->eval(appname, msgname, msg, msglen)){
    cout << "Failure" << endl;
} else {
    cout << "Success" << endl;
}
```

See Also

CreateRulesEngine(), DeleteRuleEngine(), getaction(), getsubscription(), getoption(), gethitrule(), getnohitrule(), GetErrorNo(), GetRerror(), GetErrorMessage()

gethitrule

Overview

gethitrule() retrieves one hit rule from the hit rules list created by eval(), placing it in a RULE structure. When stepping through the hit rules list using gethitrule(), a NULL indicates the end of the list.

Syntax

```
RULE *VRule::gethitrule();
```

Parameters

None

Remarks

gethitrule() should be called after the eval() function, which should follow a call to CreateRulesEngine() but precede a call to DeleteRuleEngine(). gethitrule() must be called before getsubscription() or getopt() because these functions change the hit rules list.gethitrule() will not work after getsubscription() is called.

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to call gethitrule() from the same thread as the one that called eval().

Return Value

Returns a pointer to a single RULE structure with a number and name indicating which rule was hit. When the return value is NULL, the list of hit rules has been exhausted. The rules are not returned in any specific order.

Note

Each time this API is called, the returned rule is removed from the list.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

The following code fragment describes how to walk through a list of rules that did not hit and a list of rules that hit. It should be noted that these APIs would be called after the Rules eval() API.

```
RULE *r;
cout << "NO HIT RULES" << endl;
while ( (r=rules->getnohitrule())){
   cout << "    " << r->RuleName << endl;
}
cout << "HIT RULES" << endl;
while ( (r = rules->gethitrule())){
    cout << "    " << r->RuleName << endl;
}
```

See Also

getnohitrule(), eval()

getnohitrule

Overview

getnohitrule() retrieves one no-hit rule from the no-hit rules list created by eval(), placing it in a RULE structure. Only active rules are retrieved. When stepping through the no-hit rules list using getnohitrule(), a NULL indicates the end of the list.

Syntax

```
RULE *VRule::getnohitrule();
```

Parameters

None

Remarks

getnohitrule() should be called after the eval() function, which should follow a call to CreateRulesEngine() but precede a call to DeleteRuleEngine(). getnohitrule() must be called before getsubscription() or getopt() because these functions change the hit rules list. getnohitrule() will not work after getsubscription() is called.

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to call getnohitrule() from the same thread as the one that called eval().

Return Value

Returns a pointer to a single RULE structure with a number and name indicating which rule was not hit. When the return value is NULL, the list of “no hit” rules has been exhausted. The rules are not returned in any specific order.

Note

Each time this API is called, the returned rule is removed from the list.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

The following code fragment describes how to walk through a list of rules that did not hit and a list of rules that hit. These APIs would be called after the Rules eval() API.

```
RULE *r;
cout << "NO HIT RULES" << endl;
while ( (r=rules->getnohitrule())){
   cout << "    " << r->RuleName << endl;
}
```

```
cout << "HIT RULES" << endl;
while ( (r = rules->gethitrule())){
    cout << "    " << r->RuleName << endl;
}
```

See Also

gethitrule(), eval()

getsubscription

Overview

getsubscription() gets an action within a subscription associated with a rule that evaluated to true, retrieving the subscription identifier, subscription name, and action name. When using this API within a loop, a change in the SubId (subscription sequence) of the SUBSCRIPTION structure signifies the end of one subscription and the beginning of the next.

Subscriptions are retrieved in the order in which they were created.

Syntax

```
SUBSCRIPTION* VRule::getsubscription();
```

Parameters

None

Remarks

getsubscription() should be called after the eval() function, which should follow a call to CreateRulesEngine() but before a call to DeleteRuleEngine(). getaction() should not be called after getsubscription() because it has the same functionality. getopt() should be called to retrieve the action options.

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to call getsubscription() from the same thread as the one that called eval().

Return Value

Returns a pointer to a single subscription action with a number indicating which subscription it belongs to, strictly for the purposes of checking the current subscription. If previous subscriptions have been retrieved, a different Subscription Identifier indicates that the action is for a new subscription. The subscription name and action name are also retrieved for the user. When the return value is NULL, the list of subscriptions has been exhausted. The subscription are not returned in any specific order.

Note

Each time this API is called, the returned subscription is removed from the subscription list for the hit rules.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

The following code fragment illustrates walking through a list of actions:

```
OldSubId = NULL;
int ActionCount = 0;
```

```
char * Actionlist[MY_ACTIONS_MAX];
while ((p=rules->getsubscription())){
    if ( (p->SubId != OldSubId) || (!OldSubId) ){
        //this is the first action of the new subscription
        OldSubId = p->SubId;
        myfun(ActionList,ActionCount);
        cleanup(ActionList,ActionCount);
        ActionCount = 0;
    }
    Actionlist[ActionCount] = strdup (p->action);
    ActionCount++;
    // the options should be checked here if options
    // are relevant to the action. Options only have
    // meaning if the applications programmer has
    // written code to handle options within the program
}
```

See Also

getaction(), getopt()

getaction

Overview

getaction() returns action names for rules that evaluated to true.

Syntax

```
char * VRule::getaction();
```

Parameters

None

Remarks

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to call getaction() from the same thread as the one that called eval().

Return Value

Returns a pointer to a string containing the action name. When the return value is NULL, the list of actions has been exhausted.

Note

Each time this API is called, the returned action is removed from the list.

Note

getsubscription() serves the same function as getaction().Both functions return the Subscription Identification and name, so subscription boundaries can be determined. We recommend using getsubscription() instead of getaction().

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

```
DbmsSession *session = OpenDbmsSession("fred", DbType);
if (!session || !session->Ok()) {
    cout << "Failed to open session" << endl;
    exit(1);
}
Vrule *rule = CreateRulesEngine(session);
if (!rule) {
    cout << "Unable to create rules object" << endl;
    exit(2);
}
char MessageString[65];
memset(MyMessageString, 0, 65);
```

```
strcpy(MyMessageString, "Field1|Field2,Field3");
if ( !rule->eval("MyAppGroup", "MyMessageType",
MyMessageString,
strlen(MyMessageString)) ){
cout << "Failure" << endl;
exit(3);
}
char *actionname = rule->getaction();
cout << "Action: " << actionname << endl;
DeleteRuleEngine(rule)
CloseDbmsSession(session);
```

See Also

getopt(), getsubscription()

getopt

Overview

Each subscription may contain several actions, each of which can contain several options. getopt() gets an option within an action, retrieving the option sequence number, option name, and option value. When this API is used within a loop to retrieve all options for an action, a NULL option signifies the end of the options for that subscription.

Syntax

```
OPTIONPAIR *VRule::getopt();
```

Parameters

None

Remarks

getopt() should be called after the CreateRulesEngine(), eval() and getsubscription() functions have been called and before DeleteRuleEngine().

Note

THREAD SAFETY NOTE: For multi-threaded applications, be sure to call getopt() from the same thread as the one that called eval().

Return Value

Returns a pointer to a single name-value option pair composed of an option name and option value. When the return value is NULL, the list of options for the subscription action has been exhausted.

Note

Each time this function is called, the option is removed from the list.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

The following code fragment illustrates walking through a list of options for a subscription action. This particular action finds the occurrences of a word in a file using the UNIX grep command as the action:

```
SUBSCRIPTION *psubscription;
OPTIONPAIR *poptionpair;
char string_to_find[MAX_LENGTH_STRING_TO_FIND];
VRule * rules = CreateRulesEngine(session);
    if ( !rules ){
        cout << "ERROR" << endl;
        exit(2);
    }
    if (psubscription=rules->getsubscription()) {
        if (!strcmp(psubscription->action,
                "UNIX_GREP_COMMAND")) {
            strcpy(action_string, psubscription->action);
            strcat(action_string, " ");
            while ((poptionpair=rules->getopt()){
                if (!strcmp(poptionpair->Name,
                     "WORD_TO_FIND")) {
                    strcat(string_to_find, poptionpair
                        ->Value);
                    strcat(action_string, " ");
            } else if (!strcmp(poptionpair->Name,
 "FILENAME")) {
            strcat(filename, poptionpair->Value)
            }
        }
     }
    }
    // Now execute ‘grep word filename’
system(action_string);
DeleteRuleEngine(rule);
```

See Also

getaction(), getsubscription()

getlog

Overview

getlog() retrieves a list of Rules error messages and returns it in a string format. This string will usually contain more information than GetErrorMessage() since it saves more than just the last API error.

Syntax

```
char * VRule::getlog();
```

Parameters

None

Return Value

Returns a pointer to a character string containing error messages; NULL if there are no errors.

Use GetErrorNo() to retrieve the number for the last error that occurred.

Example

```
Vrule *rule = CreateRulesEngine(session);
if (!rule) {
    cout << "Unable to create rules object" << endl;
    exit(2);
}
if (rule->GetErrorNo() ){
    cerr << "Unable to create rules engine" << endl;
    cerr << rule->getlog() << endl;
    exit(3);
}
```

ThreadCleanup

Overview

ThreadCleanup() removes data from a specific thread prior to exiting the thread. For example, if you are using UI threads, ThreadCleanup() would be used prior to a thread_exit() call. This function would typically be called for a specific thread immediately before it is destroyed.

Syntax

```
void VRule::ThreadCleanup();
```

Parameters

None

Return Value

Returns a 1 if an error occurred; zero(0) if there are no errors.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

```
struct ThreadRuleArgs
{
public:
    VRule * rules;
};

main ()// called by the main thread
DbmsSession * session = OpenDbmsSes"sion("rules",DBTYPE);

if ( !session || !session->Ok() ){
cout << "Failed to open rules database session"" << endl;
exit_thread(1);
}

VRule * rules = CreateRulesEngine(session, 0);

    thread_handle*  threads = new thread_handle[thread_number];

    ThreadRuleArgs RuleArgs;

    RuleArgs.rules = rules;

int i;

    for (i = 0; i < thread_number; ++i)
    {
#if defined (THREAD_POSIX)
        pthread_create(&threads[i], 0, run_test, &RuleArgs);
#elif defined (THREAD_NT)
        threads[i] = CreateThread(0, 0, run_test, &RuleArgs, 0, 0);
#elif defined (THREAD_UI)       // UI
        thr_create(0, 0, run_test, &RuleArgs, 0, &threads[i]);
#endif
    }
```

```
// wait for threads to complete

void * result = NULL;

#if defined (THREAD_NT)
        WaitForMultipleObjects(thread_number, threads, TRUE,
INFINITE);

#else
    for (i = 0; i < thread_number; ++i)
    {
#if defined (THREAD_POSIX)
        pthread_join(threads[i], &result);
#elif defined (THREAD_UI)       // UI
        thr_join(threads[i], NULL, &result);
#endif
    }
#endif

DeleteRuleEngine(rules);
    CloseDbmsSession(session);

exit_thread(0);
}

thread_result run_test(thread_arg arg)
{
ThreadRuleArgs * RuleArgs = (ThreadRuleArgs *) arg;
VRule * rules = RuleArgs->rules;

// get msg, msglen, AppGrp, MsgType, open outfile
// - if take from input mutex_lock as needed

if ( !rules->eval(AppGrp, MsgType, msg, pos) ){
outfile << "Fail, errno = " << rules->GetErrorNo();
outfile << " - " << rules->GetErrorMessage() << endl;
} else{

outfile << "\n\nNO HIT RULES - Rule Name (Id) " << endl;
while ( (r=rules->getnohitrule()) ) {
outfile << "    " << setw(32) << setiosflags(ios::left) << r->RuleName
<<
"(" << r->RuleId << ")" << endl;
}

outfile << "\n\nHIT RULES - Rule Name (Id)" << endl;
while ( (r = rules->gethitrule()) ){
outfile << "    " << setw(32) << setiosflags(ios::left) << r->RuleName
<<
"(" << r->RuleId << ")" << endl;
}

outfile << "\n\nACTIONS" << endl;
while ( (p=rules->getsubscription()) ){
outfile <<  "    Action(Id): " << p->action <<
"(" << p->SubId << ")" << endl;
while ( (popt=rules->getopt()) ){
outfile << "        " << popt->Sequence << " : ";
outfile << popt->Name << " - ";
outfile << popt->Value << endl;
}
}
outfile << endl;
}
}
```

```
rules->ThreadCleanup();

#ifndef WIN32
return 0;
#endif

}
```

LoadRuleSet

Overview

Using the application group and message type, LoadRuleSet() sets a flag indicating that the system should clear any current Rule Set information (identified by an Application Group / Message Type pair) and load the Rule Set indicated by the AppName and MsgName parameters.

WARNING!

LoadRuleSet() must be called after OpenDbmsSession() and CreateRulesEngine(), but before DeleteRuleEngine(). As needed, it should be called before VRule::eval(). However, it should NEVER be called after an eval() and before getsubscription(), getopt(), gethitrule(), etc.

Syntax

```
int VRule::LoadRuleSet(char* AppName,
                       char* MsgName,
                       int LoadNow=0);
```

Parameters

Name	Type	Input/ Output	Description
AppName	char*	Input	Application Group Name. Should be the Application Group for the Rule Set to load. This string should not be empty.
MsgName	char*	Input	Type of message to be evaluated. If Formatter is being used, message type is the input format name. Should be the Message Type for the Rule Set to load. This string should not be empty.
LoadNow	int	Input	Indicates when to reload the Rule Set information.

Remarks

If LoadNow is zero (0) (the default), the system will reload Rule Set information when the next eval() is called. If LoadNow is 1, the reload is done immediately, effectively ending the evaluation cycle, though eval() will complete retrieving subscription, action, and option information if doing so when receiving the signal to reload. If the rule set has not been loaded previously, LoadRuleSet() will load it only if LoadNow is set.

Return Value

Returns 1 if the load was performed or if the reload indicator was set for the Rule Set indicated; 2 if the Rule Set has not been loaded, though the reload indicator was set correctly; zero (0) if the load could not be performed.

Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

```
int result;
int LoadImmediately = 0; // Do not load immediately
char *appname;
char *msgname;
DbmsSession * session = OpenDbmsSession("fred", DbType);
if ( !session || !session->Ok() ){
    cout << "Failed to open rules database session" <<
endl;
    exit(1);
}
VRule * rule = CreateRulesEngine(session);
if ( !rule ) {
    cout << "Unable to create rules object" << endl;
    CloseDbmsSession(session);
    exit(2);
}
result = rule->LoadRuleSet(appname, msgname,
LoadImmediately);
switch (result) {
    case 0:
        cout << "Failure Loading rule set" << endl;
        cout << "Errno: " << rules->GetErrorNo();
        cout << " Error msg: " << rules->GetErrorMessage()
<< endl;
        break;
    case 1:
        cout << "Successfully loaded rule set" << endl;
        break;
    default:
        // LoadRuleSet returned 2
        cout << "Success, load will occur on first eval"
<< endl;
        break;
}
```

Note

The LoadRuleSet API will return a value of two (2) if the Rules Engine instance has never evaluated a message using the specified application group / message name pair. In this case, the LoadRuleSet API will not load the rule set, instead, the load will occur when the eval() API is invoked.

See Also

CreateRulesEngine(), DeleteRuleEngine(), eval(), GetErrorNo(), GetRerror(), GetErrorMessage()

getformatterobject

Overview

getformatterobject is a new formatter object retrieval function that will take no parameter and return the instance of the formatter that the VRule::eval() used to parse the input message. A user may want to use this function to retrieve the parsed fields and thus not have to parse before a reformat done after the eval().

Syntax

```
Formatter* VRule::getformatterobject();
```

Parameters

None

Remarks

If getFormatterobject is called prior to eval(), then the return value will be Null.

Return Value

Returns a pointer to a formatter object.

Note

In a multithreaded environment, the returned Formatter instance will be thread-safe. It will contain the data / parse results for the thread in which the eval() and Parse() was performed. Do not access other threads with this Formatter instance because the data in those threads will be different.

Example

```
Formatter * MyFormatter = NULL;
if (!rules->eval (appname, msgname,msg,msglen)) {
    cout << "Failure" <<endl;
}else{
    cout <<< "success" << endl;
    MyFormatter = rules->getformatterobject();
{
```

Rules Error Handling

GetErrorNo

Overview

GetErrorNo() returns the error number associated with the last error that occurred.

Syntax

```
int *VRule::GetErrorNo();
```

Parameters

None

Return Value

Returns the error number associated with the last error that occurred (for that thread in a multi-threaded application). 0 or -1000 is returned if no error occurred.

Example

```
VRule *rules=CreateRulesEngine(session);
    if (!rules->eval("Bravo", msgname, msg, msglen)){
        cout << "Fail, errno = ";
        cout << GetRerror(rules->GetErrorNo()) << endl;
    }else{
        // process Subscription Actions by Subscription
        // and process options by Subscription Action
    }
```

See Also

GetRerror(), GetErrorMessage()

GetErrorMessage

Overview

GetErrorMessage() returns the last error message (including any specific data such as an Application Group Name) for the current thread. This function should be used in place of GetRerror().

Syntax

```
char* VRule::GetErrorMessage();
```

Parameters

None

Return Value

Returns a pointer to a NULL-terminated string containing the description for the last error that occurred (for that thread in a multi-threaded application).

Example

```
VRule *rule=CreateRulesEngine(session);
    if (!rules->eval("Bravo", msgname, msg, msglen)){
        cout << "Fail, errno = ";
        cout << rules->GetErrorMessage() << endl;
    }else{
        // process Subscription Actions by Subscription
        // and process options by Subscription Action
    }
```

See Also

GetErrorNo(), GetRerror()

GetRerror

Overview

GetRerror() returns the description for the error number relating to a SQL or Rules processing error. Rules SQL and Rules processing errors are shown in the next section. The static error message is returned with "%s" representing where the additional data would be placed.

For example, if GetRerror(-1001) is called, it would return the following message:

Rules configuration missing Application Group -- AppGrp - %s, MsgType - %s

Note

GetErrorMessage() will return the last error message including additional information replacing the "%s".

Syntax

```
char* GetRerror(int ErrorNo);
```

Parameters

Name	Type	Input/ Output	Description
ErrorNo	int	Input	Used to determine the string value containing the meaning of the error.

Return Value

Returns a pointer to a NULL-terminated string containing the description for the error number passed into the function.

Example

```
if (!rules->eval("Bravo", msgname, msg, msglen)){
    cout << "Fail, errno = ";
    cout << GetRerror(rules->GetErrorNo()) << endl;
}else{
    // process Subscription Actions by Subscription
    // and process options by Subscription Action
}
```

See Also

GetErrorNo(), GetErrorMessage()

Chapter 4

Rules Management APIs

Rules Management APIs enable users to add, update, delete, and read rules. To use Rules Management APIs, you must include the following header files located in the MQSeries Integrator include directory:

- nnrmgr.h
- nnperm.h
- rdefs.h

You must also link with the following libraries located in the MQSeries Integrator library directory:

- librmgr.a
- libsqlobj.a
- libntools.a

Rules components must be added in the following order:

1. Application Group
2. Message Type
3. Rule
4. Rule Permission
5. Rule Expression
6. Argument
7. Subscription
8. Subscription Permission
9. Action
10. Option

WARNING!

The names of formats and fields should not be changed if they are used by a rule. The following occurs if either or both format and field names are changed:

- If you change a format name or the field names in a format, rules associated with that format become invalid.
- After a format name is changed, Rules permissions will not retrieve the correct format name, causing permission error messages.
- Subscription actions using format names fail if the format name is changed.
- If a field name is changed, the arguments using the field name become invalid and the rule will fail.

See the ***MQSeries Integrator Programming Reference for NEON Formatter*** for information on changing formats and field names.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to be case-sensitive.

Rules Management API Structures

NNDate

Overview

NNDate is passed as part of an argument in several Rules Management functions and should be cleared (see NNR_CLEAR) prior to use in a function call.

Currently, dates are defaulted, and this structure is provided for forward compatibility.

Syntax

```
typedef struct NNDate{
       unsigned char century;
       unsigned char year;
       unsigned char month;
       unsigned char day;
       unsigned char hours;
       unsigned char minutes;
       unsigned char seconds;
       unsigned char _filler;
       unsigned short mseconds;
       long InitFlag;
}
```

Members

Name	Type	Description
century	unsigned char	Century for the year. Currently, 19 (as in 1997) and 20 (as in 2001) are acceptable values.
year	unsigned char	Number for the year exclusive of the century. For example, 1996 is saved as 96 and 2001 is saved as 01.
month	unsigned char	Numeric month within the year (range 1 to 12).
day	unsigned char	Numeric day of the month (range 1 to 31).
hours	unsigned char	Number of hours past midnight in a 24 hour notation (range 0 to 23).
minutes	unsigned char	Number of minutes past the hour (range 0 to 59).
seconds	unsigned char	Number of seconds past the minute (range 0 to 59).
filler	unsigned char	This field exists to insure proper alignment of the mseconds field below and is set to zero (0).
mseconds	unsigned char	Number of milliseconds past the second (range 0 to 999).
InitFlag	long	This field is present so the software can detect if this structure was preset to zero (0) before use.

Overall Rules Management APIs and Macros

NNRMgrInit

Overview

When using Rules Management APIs, users are expected to initialize rules management by calling NNRMgrInit(). NNRMgrInit() initializes the rules management data access capability and error handling.

Syntax

```
NNRMgr * NNRMgrInit (DbmsSession *session);
```

Parameters

Name	Type	Input/ Output	Description
session	DbmsSession *	Input	Name of the MQSeries Integrator session currently open.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns a pointer to an instance of a NNRMgr object.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrClose()

NNRMgrClose

Overview

When using Rules Management APIs, users are expected to close rules management by calling the NNRMgrClose() function. NNRMgrClose() removes the user's ability to perform rules management.

Syntax

```
void NNRMgrClose (NNRMgr *pMgr);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr*	Input	Name of a current Rules Management object. See NNRMgrInit().

Remarks

A call to NNRMgrClose() should be the last call made when managing rules. Once a call to NNRMgrClose() is made, the user will not be able to manage rules without calling NNRMgrInit() again.

Note

NNRMgrClose() only cleans up resources claimed by NNRMgrInit() and does not close the DbmsSession.

Return Value

None

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit()

NNR_CLEAR

Overview

When using Rules Management APIs, user are expected to clear structures prior to invoking each function. Clearing structures should be done with a call to the NNR_CLEAR macro. NNR_CLEAR clears a structure in such a way that the Rules Management APIs can alert the user to a non-initialized structure.

Syntax

```
NNR_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any rules management structure	Input	Any structure used in Rules Management API calls except permission structures (see structure descriptions for details).

Return Value

None

Example

```
struct NNRApp app;

NNR_CLEAR(&app);
```

See Also

NN_CLEAR

Application Group Management APIs

An application group is a logical division of rules.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Application Group Management API Structures

NNRApp

Overview

NNRApp is passed as a pointer as the second parameter of the Application Group Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Application Group Management API calls.

Syntax

```
typedef struct NNRApp{
       char AppName[APP_NAME_LEN];
       long InitFlag;
}
```

Members

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRAppData

Overview

NNRAppData is passed as a pointer as the third parameter of some of the Application Group Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or any Application Group Management API calls). Use of this structure is described in each Application Group Management API section.

Syntax

```
typedef struct NNRAppData{
       NNDate DateChange;
       int ChangeAction;
       long InitFlag;
}
```

Members

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRAppUpdate

Overview

NNRAppUpdate is a structure designed to pass update information within the Rules Management APIs. It must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Rules Management API update calls.

Syntax

```
typedef struct NNRAppUpdate {
       char AppName[APP_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       long InitFlag;
}
```

Members

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group, defined by the API using this structure.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Application Group Management API Functions

NNRMgrAddApp

Overview

NNRMgrAddApp() enables the user to define a name for one application group in Rules. The user creates a name and provides it to NNRMgrAddApp(), which then saves it in Rules. Only after an application group has been defined can the application name be used in other Rules Management functions.

Syntax

```
const long NNRMgrAddApp(NNRMgr *pMgr,
                        const NNRApp *pRApp,
                        const NNRAppData *pRAppData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRApp	const NNRApp *	Input	Should be populated prior to this function call. See the Application Group Management API structures description.
pRAppData	const NNRAppData *	Input	Should be populated prior to this function call. DateChange and ChangeAction should be populated with NULL values since they are provided only for future enhancements. See the Application Group Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddApp().

A call to NNR_CLEAR for both pRApp and pRAppData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the application was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrReadApp(), NNRMgrUpdateApp()

NNRMgrReadApp

Overview

NNRMgrReadApp() attempts to read all rules defined for a specific application group name.

Syntax

```
const long NNRMgrReadApp(NNRMgr *pMgr,
                         const NNRApp *pRApp,
                         NNRAppData* const pRAppData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRApp	const NNRApp *	Input	Should be populated prior to this function call. See the Application Group Management API structures description.
pRAppData	NNRAppData* const	Output	NNRMgrReadApp populates this structure. If DateChange is non-NULL, it is assumed that the application group exists. See the Application Group Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrReadApp().

A call to NNR_CLEAR for both pRApp and pRAppData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the application was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddApp(), NNRMgrUpdateApp()

NNRMgrUpdateApp

Overview

NNRMgrUpdateApp() enables the user to update an application group name by providing the name of the application group to change (in the pRApp structure) and the new application group name to change it to (in the pRAppUpdate structure).

Syntax

```
const long NNRMgrUpdateApp (NNRMgr *pMgr,
                      const NNRApp *pRApp,
                      const NNRAppUpdate *pRAppUpdate);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRApp	const NNRApp *	Input	Should be populated prior to this function call. See the Application Group Management API structures description.
pRAppUpdate	const NNRAppUpdate *	Input	Should be populated prior to this function call. See the Application Group Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the application group was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRApp           key;
struct NNRAppData       data;
struct NNRAppUpdate update;
```

```
NNR_CLEAR(&key);
NNR_CLEAR(&data);
NNR_CLEAR(&update);

cout << "Enter old app group name \n>";
cin >> key.AppName;
cout << "Enter new app group name \n>";
cin >> update.AppName;

if (NNRMgrUpdateApp(pmgr, &key, &update)){
       cout    << endl
                      << "\tApp Group Name: " <<
key.AppName << "changed to "
                      << update.AppName << endl << endl;
       CommitXact(session);
} else {
       DisplayError(pmgr);
       RollbackXact(session);
}

CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddApp(), NNRMgrReadApp()

Message Type Management APIs

A message type identifies the type of data to which the rules will apply. As long as the user is using Formatter, message type is the same as the input format name.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Determination APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to be case-sensitive.

Message Type Management API Structures

NNRMsg

Overview

NNRMsg is passed as a pointer as the second parameter of the Message Type Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Message Type Management API calls.

Syntax

```
typedef struct NNRMsg{
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       long InitFlag;
}
```

Members

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. The message type is the input format name if the user is using Formatter.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRMsgData

Overview

NNRMsgData is passed as a pointer as the third parameter of the Message Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or by Message Type Management API calls). Use of this structure is described in each Message Type Management API section.

Syntax

```
typedef struct NNRMsgData{
       NNDate DateChange;
       int ChangeAction;
       long InitFlag;
}
```

Members

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Message Type Management API Functions

NNRMgrAddMsg

Overview

A message is a string of data to be processed. NNRMgrAddMsg() associates a message type with a specific application group. The application group and message type (if using Formatter, an input format of this name must exist) must exist prior to associating the message type to an application group using NNRMgrAddMsg(). Messages must be associated with an application group prior to adding a rule using NNRMgrAddRule().

Syntax

```
const long NNRMgrAddMsg(NNRMgr *pMgr,
                       const NNRMsg *pRMsg,
                       const NNRMsgData *pRMsgData)
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRMsg	const NNRMsg *	Input	Should be populated prior to this function call. See the Message Management API structures description.
pRMsgData	const NNRMsgData *	Input	Default the DateChange and ChangeAction parameters to NULL This is provided only for future enhancements. See the Message Type Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddMsg().

A call to NNR_CLEAR for both pRMsg and pRMsgData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the message was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR,NNRMgrReadMsg()

NNRMgrReadMsg

Overview

NNRMgrReadMsg() enables the user to read a message type.

Syntax

```
const long NNRMgrReadMsg(NNRMgr *pMgr,
                      const NNRMsg *pRMsg,
                      NNRMsgData* const pRMsgData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRMsg	const NNRMsg *	Input	Should be populated prior to this function call. See the Message Management API structures description.
pRMsgData	NNRMsgData* const	Output	NNRMgrReadMsg() populates this structure. If DateChange is non-NULL, the user should assume a message exists. See the Message Type Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrReadMsg().

A call to NNR_CLEAR for both pRMsg and pRMsgData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the message was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddMsg()

Rule Management APIs

Rule Management APIs are rules that contain expressions and are associated with subscriptions and user permissions.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Rule Management API Structures

NNRRule

Overview

NNRRule is passed as a pointer as the second parameter for some of the Rule Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Rule Management API calls. NNRRule is also part of the permission API Structures.

Syntax

```
typedef struct NNRRule{
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       char RuleName[RULE_NAME_LEN];
       long InitFlag;
}
```

Members

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. As long as the user is using Formatter, the message type is the input format name.
RuleName [RULE_NAME_LEN]	char	Name of the rule to be defined within an application group and message name pair. This rule name is defined by the user.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRRuleData

Overview

NNRRuleData is passed as a pointer as the third parameter of the Rule Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (by the user or by Rules Management API calls). Use of this structure is described in each Rule Management API section.

Syntax

```
typedef struct NNRRuleData{
       NNDate DateChange;
       int ChangeAction;
       int ArgumentCount;
       int OrCondition;
       int SubscriberIndex;
       int RuleActive;
       NNDate RuleEnableDate;
       NNDate RuleDisableDate;
       long InitFlag;
}
```

Members

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
ArgumentCount	int	Number of arguments associated with this rule.
OrCondition	int	Defaulted for now, provided for future capability.
SubscriberIndex	int	Defaulted for now, provided for future capability.
RuleActive	int	Value of 1 indicates that the rule is active, a value of zero (0) indicates that the rule is inactive.
RuleEnableDate	NNDate	Defaulted for now, provided for future capability.
RuleDisableDate	NNDate	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRRuleReadData

Overview

NNRRuleReadData is passed as a pointer to select functions in the Rule Management API. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to any Rule Management API read calls.

Syntax

```
typedef struct NNRRuleReadData {
       char RuleName[RULE_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       int OrCondition;
       int SubscriberIndex;
       int RuleActive;
       NNDate RuleEnableDate;
       NNDate RuleDisableDate;
       long InitFlag;
}
```

Members

Name	Type	Description
RuleName [RULE_NAME_LEN]	char	Name of the rule, previously defined by the user.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
OrCondition	int	Defaulted for now, provided for future capability.
SubscriberIndex	int	Defaulted for now, provided for future capability.
RuleActive	int	Value of 1 indicates that the rule is active, a value of zero (0) indicates that the rule is inactive.
RuleEnableDate	NNDate	Defaulted for now, provided for future capability.
RuleDisableDate	NNDate	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRRuleUpdate

Overview

NNRRuleUpdate is a structure containing rule update information. It must be cleared (using (NNR_CLEAR) prior to being populated, and must be populated prior to any Rule Management API update calls.

Syntax

```
typedef struct NNRRuleUpdate{
       char RuleName[RULE_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       int OrCondition;
       int SubscriberIndex;
       int RuleActive;
       NNDate RuleEnableDate;
       NNDate RuleDisableDate;
       long InitFlag;
}
```

Members

Name	Type	Description
RuleName [RULE_NAME_LEN]	char	Name of the rule to be evaluated within an application group and message type (defined by the user).
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
OrCondition	int	Defaulted for now, provided for future capability.
SubscriberIndex	int	Defaulted for now, provided for future capability.
RuleActive	int	Value of 1 indicates that the rule is active, a value of zero (0) indicates that the rule is inactive.
RuleEnableDate	NNDate	Defaulted for now, provided for future capability.
RuleDisableDate	NNDate	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Rule Management API Functions

NNRMgrAddRule

Overview

NNRMgrAddRule() enables the user to add a rule associated with a specific application group and message type pair by providing the unique application group, message type, and rule name for the rule to be added (in the pRule structure) and the new information for the rule (in the pRRuleData structure).

Prior to adding a rule, the application group and message type must be defined and exist in Rules using NNRMgrAddApp() and NNRMgrAddMsg().

When adding a rule, the current user is set as the rule owner for permissions. The owner is automatically granted read and update permission for the rule. PUBLIC is given read permission.

Syntax

```
const long NNRMgrAddRule(NNRMgr *pMgr,
                 const NNRRule *pRRule,
                 const NNRRuleData *pRRuleData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pPRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management API structures description.
pRRuleData	const NNRRuleData *	Input	DateChange, ChangeAction, RuleEnableDate and RuleDisableDates should be populated with NULL. These are provided only for future enhancements. ArgumentCount defaults to zero (0). See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddRule().

A call to NNR_CLEAR for both pRRule and pRRuleData should be made prior to populating the structures and calling this API.

Return Value

Returns 1 if the rule was added successfully; zero (0) if an error occurred. An error will occur if the component cannot be stored, if either the owner or PUBLIC cannot be stored, or if the Read / Update permissions for both the owner and PUBLIC cannot be stored.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error r message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrReadRule(), NNRMgrUpdateOwnerPerm(), NNRMgrUpdatePublicPerm()

NNRMgrReadRule

Overview

NNRMgrReadRule() enables the user to retrieve rule management information. Note that this API just reads rule maintenance information, not rule evaluation or subscription information. (To read rule evaluation or subscription information, use NNRMgrReadExpression() or NNRMgrReadSubscription()). Prior to reading a rule, the application group, message, and rule maintenance information must be defined and exist in Rules using NNRMgrAddApp(), NNRMgrAddMsg() and NNRMgrAddRule().

When retrieving rule management information, user permission to read the rule will be checked. If the user is the owner or another user and he has read permissions for the rule, he will be able to see the rule information. If the user attempting to access rule information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrReadRule(NNRMgr *pMgr,
                  const NNRRule *pRRule,
                  NNRRuleData* const pRRuleData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management API structures description.
pRRuleData	`NNRRuleData* const`	Output	NNRMgrReadRule() populates this structure. If DateChange is non-NULL, this rule exists. See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrReadRule().

A call to NNR_CLEAR for both pRRule and pRRuleData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the rule was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR,NNRMgrAddRule()

NNRMgrUpdateRule

Overview

NNRMgrUpdateRule() enables the user to update a rule associated with a specific application group / message type pair by providing the unique application group, message type, and rule name for the rule to be updated (in the pRule structure) and the new information for the rule (in the pRRuleUpdate structure).

When updating rule management information, user permission to update the rule will be checked. If the user is the owner or another user and has update permission for the rule, the user will be able to update the rule information. If the user attempting to update rule information does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrUpdateRule (NNRMgr *pMgr,
                  const NNRRule *pRule,
                  const NNRRuleUpdate *pRRuleUpdate);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management API structures description.
pRRuleUpdate	const NNRRuleUpdate *	Input	Should be populated prior to this function call. See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the rule was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
```

```
InitNNRMgrSession(pmgr, session);

struct NNRRule  key;
struct NNRRuleData      data;
struct NNRRuleUpdate update;
NNR_CLEAR(&key);
NNR_CLEAR(&data);
NNR_CLEAR(&update);

cout << "Enter app group name" << endl << ">";
cin >> key.AppName;
cout << "Enter message type name" << endl << ">";
cin >> key.MsgName;
cout << "Enter old rule name" << endl << ">";
cin >> key.RuleName;
cout << "Enter new rule name" << endl << ">";
cin >> update.RuleName;
cout << "Enter rule active (1 -> Active, 0 -> Inactive)" << endl << ">";
cin >> update.RuleActive;

if ( NNRMgrUpdateRule(pmgr,&key,&update) ) {
        cout << endl
                 << "\tOld Rule Name: " << key.RuleName << endl
                 << "\tNew rule name: " << update.RuleName << endl
                 << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}
CloseNNRMgr(pmgr,session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddRule(), NNRMgrReadRule(), NNRMgrDeleteEntireRule(), NNRMgrGetFirstRule(), NNRMgrGetNextRule()

NNRMgrDeleteEntireRule

Overview

NNRMgrDeleteEntireRule() deletes a rule by deleting each component for the rule (rule, expression, and associations with subscriptions). Subscriptions can be deleted from the rule set using NNRMgrDeleteEntireSubscription. The user provides the application name, message type, and rule name.

Note

NNRMgrDeleteEntireRule() DELETES ALL COMPONENTS ASSOCIATED WITH A RULE, SO UNLESS THE USER WANTS A RULE TO BE DELETED, THIS API SHOULD NOT BE CALLED.

When deleting rule management information, user permission to update the rule will be checked. If the user is the owner and has update permissions for the rule, the rule.will be deleted. If the user attempting to update rule information is not the owner but does have update access, the rule will be set to inactive but not deleted. If the user does not have update access, an error will be returned indicating that the user does not have update permission and no change will occur.

Syntax

```
const long NNRMgrDeleteEntireRule (NNRMgr *pMgr,
                                   const NNRRule *pRRule);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the rule was deleted successfully; 2 if the rule was deactivated; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRRule          key;
struct NNRRuleData      data;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter rule name \n>";
cin >> key.RuleName;

if (NNRMgrDeleteEntireRule(pmgr, &key)){
        cout    << endl
                << "\tRule Name: " << key.RuleName << "
Deleted."
                << endl << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrUpdateRule(), NNRMgrAddRule(), NNRMgrReadRule(), NNRMgrGetFirstRule(), NNRMgrGetNextRule()

NNRMgrGetFirstRule

Overview

NNRMgrGetFirstRule() and NNRMgrGetNextRule() enable the user to iterate through a list of rules associated with a message type and application group pair.

When retrieving rule management information, user permission to read the rule will be checked. If the user is the owner or another user and has read or update permissions for the rule, the user will be able to see the rule information. If the user attempting to access rule information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetFirstRule (NNRMgr *pMgr,
                   const NNRRule *pRRule,
                   NNRRuleReadData * const pRRuleData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be completely populated except for the RuleName field prior to this function call. See the Rule Management API structures description.
pRRuleReadData	NNRRuleRead Data const *	Output	NNRMgrGetFirstRule populates this structure. See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the rule was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_RULES, no rules were found for the application group and message type specified in the pRRule structure.

Example

See *Sample Program 2: Rules Management API.*

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrUpdateRule(), NNRMgrAddRule(), NNRMgrReadRule(), NNRMgrDeleteEntireRule(), NNRMgrGetNextRule()

NNRMgrGetNextRule

Overview

NNRMgrGetFirstRule() and NNRMgrGetNextRule() enable the user to iterate through a list of rules associated with a message type and rule name pair.

When retrieving rule management information, user permission to read the rule will be checked. If the user is the owner or another user and they have read or update permissions for the rule, they will be able to see the rule information. If the user attempting to access rule information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetNextRule (NNRMgr *pMgr,
                   NNRRuleReadData * const pRRuleData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRuleRead Data	NNRRuleRead Data const *	Output	NNRMgrGetFirstRule populates this structure. See the Rule Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls. NNRMgrGetFirstRule() must be called before NNRMgrGetNextRule().

Return Value

Returns 1 if the rule was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_RULES, the end of the rules list has been reached.

Example

See *Sample Program 2: Rules Management API.*

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrUpdateRule(), NNRMgrAddRule(), NNRMgrReadRule(), NNRMgrDeleteEntireRule(), NNRMgrGetFirstRule()

Permissions APIs

When a rule is added using NNRMAddRule(), the user is given ownership of the rule, as well as read and update permissions. PUBLIC is given read permission.

The same occurs when a subscription is added using NNRMgrAddSubscription(). These default permissions can be changed by using NNRMgrUpdateOwnerPerm() and NNRMgrUpdatePublicPerm().

The rule expression or subscription actions can be added by the owner without changing the default permissions. Once permissions are defined for a rule or subscription, an owner can give ownership to another user and change permissions for themselves or other PUBLIC using other Permissions APIs.

Replace calls to the permission APIs that take a NNRRule structure with calls to the following new APIs that take a NNRComponent structure:

Instead of using:	Use:
NNRMgrAddOwnerPermission	NNRMgrUpdateOwnerPerm
NNRMgrAddOtherUserPermission	NNRMgrUpdatePublicPerm
NNRMgrGetFirstRulePermission	NNRMgrGetFirstPerm
NNRMgrGetNextRulePermission	NNRMgrGetNextPerm
NNRMgrUpdateOwner	NNRMgrChangeOwner
NNRMgrUpdateOwnerPermission	NNRMgrUpdateOwnerPerm
NNRMgrUpdateOtherUserPermission	NNRMgrUpdatePublicPerm

Permission Management API Structures

NNUserPermissionData

Overview

NNUserPermissionData is passed as an argument in several Rules Management functions affecting permissions and should be cleared (see NN_CLEAR) prior to use in a function call.

Syntax

```
typedef struct NNUserPermissionData{
       NNPermissionData Permission;
       char ParticipantName[NN_PARTICIPANT_NAME_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
Permission	NNPermission Data	Specifies the permission for this specific participant.
ParticipantName [NN_PARTICIPANT_ NAME_LEN]	char	MQSeries Integrator login name of the user to whom the permission is being assigned. All capital letters for Oracle. Case sensitive for Sybase. PUBLIC for all users other than the owner.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NN_CLEAR).

NNPermissionData

Overview

NNPermissionData is passed as an argument in several Rules Management functions affecting permissions and should be cleared prior to use in a function call (see NN_CLEAR).

Syntax

```
typedef struct NNPermissionData{
       int Sequence;
       char PermissionName[NN_PERMISSION_NAME_LEN];
       char PermissionValue[NN_PERMISSION_VALUE_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
Sequence	int	Ordering value for this specific permission name-value pair.
PermissionName [NN_PERMISSION_NAME_LEN]	char	Type of permission being defined for the rule and user permission. Currently, only Update is valid.
PermissionValue [NN_PERMISSION_VALUE_LEN]	char	Value for the permission being defined for the rule and user permission. Currently, only values associated with Update are valid. These values are either Granted or DenyAll.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NN_CLEAR).

NNRComponent

Overview

After a NNRRule structure (for example, myRule) is created for a rule, the user must then create a NNRComponent with ComponentType = NNRCOMP_RULE and ComponentUnion.pRRule = &myRule.

After an NNRSubs structure (for example, mySubs) is created for a rule, the user must create a NNRComponent with ComponentType = NNRCOMP_SUBS and ComponentUnion.pRSubs = &mySubs.

The NNRComponent is then sent into the desired Permission API. NNRComponent can be initialized by calling NNR_CLEAR before populating.

Syntax

```
typedef enum NNRComponentTypes{
   NNRCOMP_RULE      =1,
   NNRCOMP_SUBS      =2
  }NNRComponentTypes;

typedef union NNRComponentUnion {
   const struct NNRRule *pRRule;
    const struct NNRSubs *pRSubs;
  }NNRComponentUnion;

typedef struct {
   Long InitFlag:
   NNRComponentTypes ComponentType;
   NNRComponentUnion ComponentUnion:
  }NNRComponent;
```

Parameters

Name	Type	Description
InitFlag	Long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (See NNR_CLEAR).
ComponentType	NNRComponentType	Either NNRCOMP_RULE or NNROCOMP_SUBS to label the type of component used in ComponentUnion.
ComponentUnion	NNRComponentUnion	A union where either pRRule is set to point to a previously defined NNRRule structure or pRSubs is set to point to a previously defined NNRSubs structure.

Overall Permission Macro

NN_CLEAR

Overview

When using Rules Management APIs affecting permissions, users are expected to clear structures prior to invoking each function. Clearing structures should be done with a call to the NN_CLEAR macro. NN_CLEAR clears a structure in such a way that the Rules Management APIs can alert the user to a non-initialized structure.

Syntax

```
NN_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any rules management permissions structure	Input	Any structure used in Rules Management API calls affecting permissions (see structure descriptions for details).

Return Value

None

Example

```
struct NNPermission permit;

NN_CLEAR(&permit);
```

Permission API Functions

NNRMgrAddOwnerPermission

Overview

Note

This routine is not needed in Release 1.0 because default permissions are added for the owner when the component is added. See NNRMgrUpdateOwnerPerm().

This API is retained for backward compatibility and will update the permission.

NNRMgrAddOwnerPermission() enables the user to add/update one permission to an owner-component pair. NNRMgrAddOwnerPermission() must be called after NNRMgrAddRule().

Syntax

```
const long NNRMgrAddOwnerPermission(NNRMgr *pMgr,
              const NNRRule *pRRule,
              const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Permission structures description.

Remarks

NNRMgrAddRule() must be called prior to calling NNRMgrAddOwnerPermission().

A call to NNR_CLEAR for pRRule and NN_CLEAR for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the owner permission was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession          *session:
NNRRMgr              *pmgr;

struct NNPermissionData PermissionData:
struct NNRArg           aarkey

NNR_CLEAR (aarkey);
NN_CLEAR (&PermisionData);
NNRMGrInit (pmgr, session);

strcpy (PermissionData.PermissionName, "Update");
strcpy (PermissionData.PermissionValue, "Granted");

BeginXact (session);
// Grant owner update permission
iret = NNRMgrAddOwnerPermission (pmgr, &aarkey,
&PermissionData);
if(iret){
    cout << endl
         << "Permission: " <<
         << "\tName: " << PermissionData.PermissionName
         <<endl
         << "\tValue: " <<PermissionData.PermissionValue
         << "Added." << endl << endl;
    CommitXact (session);
} else {
    cout <<"Error number is " << NNRGetErrorNo (pmgr)
        <<  end;
    cout << "Message is: " << NNRGetErrorMessage (pmgr)
        << endl;
        RollbackXact (session);
}
CloseNNRMgr (pmgr, session);
return;
```

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrAddRule(), NNRMgrAddOtherUserPermission(), NNRMgrUpdateOwnerPerm(), NNRMgrUpdatePublicPerm()

NNRMgrAddOtherUserPermission

Overview

Note

This routine is not needed in Release 1.0 because default permissions are added for PUBLIC when the component is added. See NNRMgrUpdatePublicPerm().

This API is retained for backward compatibility and will update the permission.

NNRMgrAddOtherUserPermission() enables the user to add one permission to another user-component pair. The other user is assumed to be the public user, PUBLIC.

Syntax

```
const long NNRMgrAddOtherUserPermission(NNRMgr *pMgr,
               const NNRRule *pRRule,
               const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Permission structures description.

Remarks

NNRMgrAddRule() should be called prior to calling NNRMgrAddOtherUserPermission().

The other user is assumed to be the public user, PUBLIC.

A call to NN_CLEAR for pRRule and NNR_Clear for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the other user’s permission was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession              *session;
NNRMgr                   *pmgr;

struct NNPermissionDataPublicPermissionData;
struct NNRArg           aarkey;

NNR_CLEAR(aarkey);
NN_CLEAR (&PublicPermisionData);
NNRMGrInit (pmgr, session);

strcpy (PublicPermissionData.PermissionName, "Update");
strcpy (PublicPermissionData.PermissionValue, "Granted");

BeginXact (session);
// Grant owner update permission
iret = NNRMgrAddOtherUserPermission (pmgr, &aarkey,
&PublicPermissionData);
if( iret){
    cout << endl
         << "Permission: " <<
         << "\tName: "
         << PublicPermissionData.PermissionName
         << endl
         <<"\tValue:
                   "<<PublicPermissionData.PermissionValue
         << "Added." << endl << endl;
    CommitXact (session);
} else {
    cout <<"Error number is " << NNRGetErrorNo (pmgr)
        <<  end;
    cout << "Message is: " << NNRGetErrorMessage (pmgr)
        << endl;
        RollbackXact (session);
}
CloseNNRMgr (pmgr, session) :
return;
```

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrAddRule(), NNRMgrAddOwnerPermission(), NNRMgrUpdatePublicPerm(), NNRMgrUpdateOwnerPerm()

NNRMgrGetFirstRulePermission

Overview

Note

Using NNRMgrGetFirstPerm() instead of this API is recommended.

NNRMgrGetFirstRulePermission() enables the user to prepare the list of user-permissions pairs for retrieval by the NNRMgrGetNextRulePermission() API.

Syntax

```
const long NNRMgrGetFirstRulePermission(
          NNRMgr *pMgr,
          const NNRRule *pRRule,
          NNUserPermissionData* const pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pPermissionData	NNUserPermission Data* const	Output	Will be populated by the call to NNRMgrGetFirst Rule Permission().

Remarks

A call to NNR_CLEAR for pRRule and NN_CLEAR for pPermissionData should be made prior to populating the structure or calling this API.

Return Value

Returns 1 if the list of user-permission pairs is prepared successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_PERMISSIONS, no permissions were found for the rule or subscription specified in the pRComponent structure.

Example

See Sample Program 2: NNRMgrGetFirstPerm API

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrGetNextRulePermission()

NNRMgrGetNextRulePermission

Overview

Note

Using NNRMgrGetNextPerm() instead of this API is recommended.

NNRMgrGetNextRulePermission() enables the user to retrieve an owner-permission pair from the owner-permissions list for a rule. When iterating through the list, a NULL pPermissionData indicates the end of the list. NNRMgrGetFirstRulePermission() MUST be called prior to using this routine.

Syntax

```
const long NNRMgrGetNextRulePermission(
          NNRMgr *pMgr,
          const NNUserPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pPermissionData	const NNUserPermission Data *	Output	Will be populated by the call to NNRMgrGetNextRule Permission().

Remarks

A call to NN_CLEAR for pPermissionData should be made prior to calling this API.

NNRMgrGetFirstRulePermission() MUST be called prior to using this routine.

Return Value

Returns 1 if an user-permission pair was read from the list successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_PERMISSIONS, the end of the permissions list has been reached.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrGetFirstPerm(), NNRMgrGetNextPerm()

NNRMgrGetFirstPerm

Overview

NNRMgrGetFirstPerm() enables the user to prepare the list of user-permissions pairs for rules or subscriptions for retrieval by the NNRMgrGetNextPerm() API.

Syntax

```
const long NNRMgrGetFirstPerm(
         NNRMgr *pMgr,
         const NNRComponent *pRComponent
         NNUserPermissionData* const pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRComponent	const NNRComponent *	Input	Should be populated prior to this function call. See the Permission Management structures description.
pPermissionData	NNUserPermission Data* const	Output	Will be populated by the call to NNRMgrGetFirst Rule Perm().

Remarks

A call to NNR_CLEAR for pRComponent and NN_CLEAR for pPermissionData should be made prior to populating the structure or calling this API.

Call NNRMgrGetNextPerm() to retrieve all remaining rule or subscription permissions before calling NNRMgrGetFirstPerm() to retrieve permissions for another rule or subscription.

Return Value

Returns 1 if the list of user-permission pairs is prepared successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_PERMISSIONS, no permissions were found for the application group, message type, and rule or subscription specified in the pRComponent structure.

Example

See Sample Program 2: NNRMgrGetNextPerm API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrGetNextPerm()

NNRMgrGetNextPerm

Overview

NNRMgrGetNextPerm() enables the user to retrieve an user-permission pair from the user-permissions list for a rule. When iterating through the list, a NULL pPermissionData indicates the end of the list. NNRMgrGetFirstPerm() MUST be called prior to using this routine.

Syntax

```
const long NNRMgrGetNextPerm(
          NNRMgr *pMgr,
          const NNUserPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pPermissionData	const NNUserPermission Data *	Output	Will be populated by the call to NNRMgrGetNext Perm().

Remarks

A call to NN_CLEAR for pPermissionData should be made prior to calling this API.

NNRMgrGetFirstPerm() MUST be called prior to using this routine.

Return Value

Returns 1 if an user-permission pair was read from the list successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_PERMISSIONS, the end of the permissions list has been reached.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrGetFirstPerm()

NNRMgrUpdateOwner

Overview

Note

Using NNRMgrChangeOwner() instead of this API is recommended.

NNRMgrUpdateOwner() enables the rule's owner to change ownership to a new user. Only the current rule owner can change ownership. The new owner's name must exist in the database and must be in the same group / role as the current user. The original owner's permissions are transferred to the new owner, overwriting any previous permissions of the new owner.

Syntax

```
const long NNRMgrUpdateOwner(NNRMgr *pMgr,
                             const NNRRule *pRRule,
                             char *pNewOwner);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pNewOwner	char *	Input	Should be populated with the new owner's login name prior to this function call.

Remarks

A call to NNR_CLEAR for pRRule should be made prior to populating the structures or calling this API.

Note that for Oracle, all owner names must be in uppercase. For example, owner should be OWNER. Sybase uses the same case as the login name.

Return Value

Returns 1 if the owner was changed successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrUpdateOwnerPerm(), NNRMgrUpdatePublicPerm()

NNRMgrChangeOwner

Overview

NNRMgrChangeOwner() enables the owner of the rule or subscription to change ownership to a new user. Only the current owner can change ownership. The new owner's name must exist in the database and must be in the same group / role as the current owner. The original owner's permissions are transferred to the new owner, overwriting any previous permissions of the new owner.

Syntax

```
const long NNRMgrChangeOwner(NNRMgr *pMgr,
                             const NNRComponent *pRComponent,
                             char *pNewOwner);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRComponent	const NNRComponent *	Input	Should be populated prior to this function call. See the Permission Management structures.
pNewOwner	char *	Input	Should be populated with the new owner's login name prior to this function call.

Remarks

A call to NNR_CLEAR for pRComponent should be made prior to populating the structures or calling this API.

Note that for Oracle, all owner names must be in upper-case. For example, owner should be OWNER. Sybase uses the same case as the login name.

Return Value

Returns 1 if the owner was changed successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrUpdateOwnerPerm(), NNRMgrUpdatePublicPerm()

NNRMgrUpdateOwnerPermission

Overview

Note

Using NNRMgrUpdateOwnerPerm instead of this API is recommended.

NNRMgrUpdateOwnerPermissions() enables the user to change rule permissions for the owner. Only the owner can affect owner permissions. The owner’s permissions will be added if they do not already exist.

Syntax

```
const long NNRMgrUpdateOwnerPermission(
             NNRMgr *pMgr,
             const NNRRule *pRRule,
             const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Rule Permission structures description.

Remarks

A call to NNR_CLEAR for pRRule and NN_CLEAR for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the owner’s permissions were updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: NNRMgrUpdateOwnerPerm.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrUpdateOtherUserPermission(), NNRMgrUpdateOwnerPerm()

NNRMgrUpdateOwnerPerm

Overview

NNRMgrUpdateOwnerPerm() enables the user to add / change permissions for the owner. Only the owner can affect owner permissions. By default, update and read permissions for all rules and subscriptions are given to their owner.

Syntax

```
const long NNRMgrUpdateOwnerPerm(
             NNRMgr *pMgr,
             const NNRComponent *pRComponent,
             const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRComponent	const NNRComponent *	Input	Should be populated prior to this function call. See the Permission Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Rule Permission structures description.

Remarks

A call to NNR_CLEAR for pRComponent and NN_CLEAR for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the owner's permissions were updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNRMgrUpdatePublicPerm()

NNRMgrUpdateOtherUserPermission

Overview

Note

Using NNRMgrUpdatePublicPerm() instead of this API is recommended.

NNRMgrUpdateOtherUserPermissions() enables the owner to change rule permissions for another user. Only the owner may change permissions for other users. The other user's permissions (PUBLIC) will be added if they do not already exist.

Syntax

```
const long NNRMgrUpdateOtherUserPermission(
            NNRMgr *pMgr,
            const NNRRule *pRRule,
            const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call. See the Rule Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Rule Permission structures description.

Remarks

NNRMgrAddRule() should be called prior to calling NNRMgrAddOtherUserPermission().

A call to NNR_CLEAR for pRRule and NN_CLEAR for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the other user's permission was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: NNRMgrUpdatePublicPerm.

See Also

NNRMgrInit(), NN_CLEAR, NNR_CLEAR,
NNRMgrUpdateOwnerPermission(), NNRMgrUpdatePublicPerm()

NNRMgrUpdatePublicPerm

Overview

NNRMgrUpdatePublicPerm() enables the owner to change permissions for another user. Only the owner can change permissions for other users. By default, other users (PUBLIC) will be granted read permission and denied update privilege. NNRMgrUpdatePublicPerm() can add any permissions that do not currently exist.

Syntax

```
const long NNRMgrUpdatePublicPerm(
            NNRMgr *pMgr,
            const NNRComponent *pRComponent,
            const NNPermissionData *pPermissionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRComponent	const NNRComponent *	Input	Should be populated prior to this function call. See the Permission Management structures description.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call. See the Permission structures description.

Remarks

NNRMgrAddOtherUserPermission() should be called prior to calling NNRMgrUpdatePublicPerm().

A call to NNR_CLEAR for pRComponent and NN_CLEAR for pPermissionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the other user’s permission was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NN_CLEAR, NNR_CLEAR, NNRMgrUpdateOwnerPerm()

Operator Determination APIs

Operators List

An operator is defined by type (existence, integer, string, float, date, time, datetime, field-to-field) and associated symbol.

Existence operators enable a user to determine if a field exists and is not empty in a message. Integer, string, and float date, time and datetime operators evaluate a message field against a static value using the operator symbol. Field-to-field operators compare two groups of data (fields) within the message.

For a more detailed explanation of a field, see the ***MQSeries Integrator Programming Reference for NEON Formatter***.

Operator Symbols

Operators are defined by type: existence, integer, string, date, time, datetime, float, integer field-to-field, string field-to-field, date field-to-field, time field-to-field, and datetime field-to-field. All operators except NOT_Exist will not hit if the field does not exist or is empty--even for field-to-field comparisons.

Existence Operators

Existence operators determine if a field exists or is empty in a message.

Integer Operators

Integer operators compare numeric values. For static value comparisons, the comparison value must be a whole number (preceded by ‘+’ or ‘-’ if desired). If the message field is not numeric, its value will be assumed to be 0 (zero), so the rule may hit in this case.

String Operators

String Operators compare strings of characters. Case-sensitive operators will notice that the characters ‘a’ and ‘A’ are different. On an EBCDIC machine, the order of characters is as follows: ‘a’ - ‘z’ < ‘A’- ‘Z’ < ‘0’ - ‘9’. In ASCII, the order of characters is as follows: ‘0’ - ‘9’ < ‘A’ - ‘Z’ < ‘0’ - ‘9’. Therefore, rules may work differently on different platforms. In the current release of MQSeries Integrator, trailing blanks are kept, and a field comtaining just blanks is not empty. Also, a field containing "Integrator" does not equal "Integrator".

WARNING!

Case-sensitive operators will not work correctly on case-insensitive databases.

Float Operators

Float operators compare decimal (real) numeric values. For static value comparison, the comparison value must be a numeric value (preceded by ‘+’ or ‘-’ if desired) and contain a decimal point (‘.’). Note that when comparing float values, ‘1.5’ does not always equal ‘1.5’ because of real number precision.

Rules Date, Time, DateTime Operators

The Rules Date, Time, and DateTime operators enable users to create and evaluate the rule arguments that perform Date, Time, and DateTime comparisons. Rules performs comparisons between unmatched Date, Time, and DateTime types based on the operator used in the argument. The Date operators compare the date portion (i.e., YYYYMMDD), the Time operators, the time portion (i.e., hhmmss) and DateTime operators, the entire value (i.e., YYYYMMDDhhmmss). For example, if an argument using a Date operator compares a Date against a DateTime (e.g., F1 DATE=F2, where F1 is a Date and F2 is a DateTime), then the value of the first field (F1) will be compared against only the date portion of the second field (F2).

The International ISO-8601:1988 standard date notation has been selected as the standard format. The International Standard ISO 8601:1988 specifies numeric representations of date and time. The standard date notation is YYYYMMDD, where YYYY is the year in the usual Gregorian calendar, MM is the month of the year between 01 (January) and 12 (December), and DD is the day of the month between 01 and 31. The standard time notation is hhmmss where hh is the number of complete hours that have passed since midnight between 00 and 23, mm is the number of complete minutes that have passed since the start of the hour between 00 and 59, and ss is the number of seconds since the start of the minute between 00 and 59.

Note

The visual representation of dates in the GUI does not adhere to the standard DateTime format (i.e., YYYYMMDD and hhmmss). However, the Management APIs must receive Date, Time, and DateTime values in the standard DateTime format. For both the GUI and the APIs, all years must be given in four digits for Year 2000 (Y2K) compliance.

Specifying a Year Cutoff Value

The internal application functions of MQSeries Integrator use date-time information for archiving, time stamping, logging, etc. These functions use four-digit year notation, or use Universal Time Coordinated (UTC) for time stamps and, therefore, are Y2K compliant.

Within the message handling and processing functionality, date information can be embedded, reformatted, etc. MQSeries Integrator provides date and date-time comparison and reformatting functions for this. Date/date-time formats and supported date/date-time rules facilities are Y2K compliant for accepting input and providing output date information. Default date and date-time formats use four-character years and are, thus, Y2K compliant. MQSeries Integrator also supports two-character years as custom formats.

For an input control that specifies a data type of custom date/time with a two-digit year format string, specify a “year cutoff” value, which tells Formatter how to convert the two-digit year date value to a four-digit year date value.

To perform this conversion, Formatter compares the year value of the input data to the specified Year Cutoff value and assigns the century designation as required. That is, based on the comparison, Formatter converts the year value “XX” to “20XX” (21st century year) or “19XX” (20th century year) as appropriate.

The year cutoff algorithm is as follows:

- year value >= cutoff value -> 19XX
- year value < cutoff value -> 20XX

Valid year cutoff values: 0 to 99 inclusive. With this method, any year 00 to 99 may be converted into either 19XX or 20XX.

The following are some examples of how the New Era of Networks, Inc., Formatter interprets the Year Cutoff number:

- If you specify the Year Cutoff number as 50, then all two-digit input dates from 50 to 99 will be designated as 1950 to 1999 output dates. All two-digit input dates from 00 to 49 will be designated as 2000 to 2049 output dates.
- If you specify the cutoff date as 75, then all two-digit input dates from 75 to 99 will be designated as 1975 to 1999 output dates. All two-digit input dates from 00 to 74 will be designated as 2000 to 2074 output dates.

If the output data type is a custom date and time, you must specify a format (date/time string). The formats that you can specify are predefined by the Formatter installation.

Operator Symbols

Operator Handle	Operator Symbol	Operator Type	Description
Existence Operators			
0	NOT_EXIST	1	Required Field Is Not Present
1	EXIST	1	Required Field Is Present
Integer Operators			
2	INT=	2	Integer Equals
3	INT>	2	Integer Greater Than
4	INT<	2	Integer Less Than
5	INT>=	2	Integer Greater Than Or Equal To
6	INT<=	2	Integer Less Than Or Equal To
7	INT<>	2	Integer Not Equal To
String Operators			
8	STRING=	3	String Equal To
9	STRING>	3	String Greater Than
10	STRING<	3	String Less Than
11	STRING>=	3	String Greater Than Or Equal To
12	STRING<=	3	String Less Than Or Equal To
13	STRING<>	3	String Not Equal To
Field To Field Integer Operators			
18	F2FINT=	4	Field To Field Integer Equal To

Operator Handle	Operator Symbol	Operator Type	Description
19	F2FINT>	4	Field to Field Integer Greater Than
20	F2FINT<	4	Field to Field Integer Less Than
21	F2FINT>=	4	Field to Field Integer Greater Than Or Equal To
22	F2FINT<=	4	Field to Field Integer Less Than Or Equal To
23	F2FINT<>	4	Field To Field Integer Not Equal To
Field To Field String Operators			
24	F2FSTRING=	5	Field To Field String Equal To
25	F2FSTRING>.	5	Field To Field String Greater Than
26	F2FSTRING<	5	Field To Field String Less Than
27	F2FSTRING>=	5	Field To Field String Greater Than Or Equal To
28	F2FSTRING<=	5	Field To Field String Less Than Or Equal To
29	F2FSTRING<>	5	Field To Field String Not Equal To
Float Integer Operators			
34	FLOAT=	6	Float Equals
35	FLOAT>	6	Float Greater Than
36	FLOAT<	6	Float Less Than
37	FLOAT>=	6	Float Greater Than Or Equal To
38	FLOAT<=	6	Float Less Than Or Equal To
39	FLOAT<>	6	Float Not Equal To
Case-sensitive String Operators			
56	CSSTRING =	9	Case Sensitive String Equal To
57	CSSTRING>	9	Case Sensitive String Greater Than
58	CSSTRING<	9	Case Sensitive String Less Than

Operator Handle	Operator Symbol	Operator Type	Description
59	CSSTRING>=	9	Case Sensitive String Greater Than Or Equal To
60	CSSTRING<=	9	Case Sensitive String Less Than Or Equal To
61	CSSTRING<>	9	Case Sensitive String Not Equal To
Field To Field Case-sensitive Operators			
62	F2FCSSTRING=	10	Field To Field Case Sensitive String Equal To
63	F2FCSSTRING>	10	Field To Field Case Sensitive String Greater Than
64	F2FCSSTRING<	10	Field To Field Case Sensitive String Less Than
65	F2FCSSTRING>=	10	Field To Field Case Sensitive String Greater Than Or Equal To
66	F2FCSSTRING<=	10	Field To Field Case Sensitive String Less Than Or Equal To
67	F2FCSSTRING<>	10	Field To Field Case Sensitive String Not Equal To
Date Operators			
68	DATE=	11	Date Equal To
69	DATE>	11	Date Greater Than
70	DATE<	11	Date Less Than
71	DATE>=	11	Date Greater Than Or Equal To
72	DATE<=	11	Date Less Than Or Equal To
73	DATE<>	11	Date Not Equal To
Field to Field Date Operators			
74	F2FDATE=	12	Field To Field Date Equal To
75	F2FDATE>	12	Field To Field Date Greater Than
76	F2FDATE<	12	Field To Field Date Less Than
77	F2FDATE>=	12	Field To Field Date Greater Than Or Equal To
78	F2FDATE<=	12	Field To Field Date Less Than Or Equal To

Operator Handle	Operator Symbol	Operator Type	Description
79	F2FDATE<>	12	Field To Field Date Not Equal To
Time Operators			
80	TIME=	13	Time Equal To
81	TIME>	13	Time Greater Than
82	TIME<	13	Time Less Than
83	TIME>=	13	Time Greater Than Or Equal To
84	TIME<=	13	Time Less Than Or Equal To
85	TIME<>	13	Time Not Equal To
86	F2FTIME=	14	Field To Field Time Equal To
Field to Field Time Operators			
87	F2FTIME>	14	Field To Field Time Greater Than
88	F2FTIME	14	Field To Field Time Less Than
89	F2FTIME>=	14	Field To Field Time Greater Than Or Equal To
90	F2FTIME<=	14	Field To Field Time Less Than Or Equal To
91	F2FTIME<>	14	Field To Field Time Not Equal To
DateTime Operators			
92	DATETIME=	15	DateTime Equal To
93	DATETIME>	15	DateTime Greater Than
94	DATETIME<	15	DateTime Less Than
95	DATETIME>=	15	DateTime Greater Than Or Equal To
96	DATETIME<=	15	DateTime Less Than Or Equal To
97	DATETIME<>	15	DateTime Not Equal To
Field to Field DateTime Operators			
98	F2FDATETIME=	16	Field To Field DateTime Equal To
99	F2FDATETIME>	16	Field To Field DateTime Greater Than

Operator Handle	Operator Symbol	Operator Type	Description
100	F2FDATETIME<	16	Field To Field DateTime Less Than
101	F2FDATETIME>=	16	Field To Field DateTime Greater Than Or Equal To
102	F2FDATETIME<=	16	Field To Field DateTime Less Than Or Equal To
103	F2FDATETIME<>	16	Field To Field DateTime Not Equal To

Operator Management API Structures

NNROperator

Overview

NNROperator is passed as a pointer to the second parameter of the Operator Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to Operator Management API calls. Use of this structure is described in each Operator Management API section.

Syntax

```
typedef struct NNROperator {
       int OperatorHandle;
       char OperatorSymbol[OPERATOR_SYMBOL_LEN];
       int OperatorType;
}
```

Parameters

Name	Type	Description
OperatorHandle	int	Unique operator handle.
OperatorSymbol [OPERATOR_SYMBOL_LEN]	char	String definition of operator.
OperatorType	int	Type of data.

Operator Management API Functions

NNRMgrGetFirstOperator

Overview

Prior to adding arguments, users must know what operators are available and supported within the current Rules installation.

NNRMgrGetFirstOperator() provides a way of starting to retrieve this information. NNRMgrGetFirstOperator() returns the first operator (in the pOperator parameter), after which the user should call NNRMgrGetNextOperator().

The pOperator structure contains a unique operator, specified by an operator symbol, operator type, and operator handle. The operator type and operator symbol provide a means for the user to choose the operator handle to provide to the argument addition function, NNRMgrAddArgument(), or the operator symbol to provide the expression addition/update functions, which are NNRMgrAddExression() and NNRMgrUpdateExpression().

Syntax

```
const long NNRMgrGetFirstOperator(NNRMgr *pRMgr,
                  NNROperator * const pOperator);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pOperator	NNROperator * const	Output	Will be populated by NNRMgrGetFirstOperator(). See the Operator Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetFirstOperator().

A call to NNR_CLEAR for pOperator should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the first operator was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_OPERATORS, no operators were found.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetNextOperator(), NNRMgrGetErrorNo(), NNRMgrGetError().

NNRMgrGetNextOperator

Overview

Prior to adding arguments, users must know what operators are available and are supported within the current Rules installation. After retrieving the first operator using NNRMgrGetFirstOperator()), NNRMgrGetNextOperator() provides a way to iterate through the operators. NNRMgrGetNextOperator() returns an operator (in the pOperator parameter). The pOperator structure contains a unique operator specified by an operator symbol, operator type, and operator handle. The operator type and operator symbol provide a means for the user to choose the operator handle to provide to the argument addition function, NNRMgrAddArgument(), or the operator symbol to provide the expression addition / update functions, which are NNRMgrAddExression() and NNRMgrUpdateExpression().

Syntax

```
const long NNRMgrGetNextOperator(NNRMgr *pRMgr,
             NNROperator * const pOperator );
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pOperator	NNROperator * const	Output	Will be populated by NNRMgrGetFirstOperator() See the Operator Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetNextOperator().

A call to NNR_CLEAR for pOperator should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the next operator was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_OPERATORS, the end of the operators list has been reached.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstOperator(), NNRMgrGetErrorNo(), NNRMgrGetError().

Expression Management APIs

Rules can include Boolean expressions containing the operators '&' (AND) and '|' (OR) arguments, and parentheses to control the order of evaluation. The user can evaluate the messages against the rule F1 INT = 1 | F2 INT = 2.

Note

For MQSeries Integrator Rules release 1.0, all arguments must be active. Therefore, all inactive arguments must be activated or deleted during the database upgrade. (NNRie will automatically delete inactive rules.)

Definitions

Boolean Expression

A single argument or more than one argument connected by Boolean operators with optional parentheses to change the order of evaluation from the standard Boolean operator precedence.

Examples:

Arg1

Arg1 BooleanOp1 Arg2

Arg1 BooleanOp1 Arg2 BooleanOp2 Arg3

- standard precedence used for evaluation

Arg1 BooleanOp1 (Arg2 BooleanOp2 Arg3)

- arguments inside the parentheses will be evaluated first.

The maximum length of a Boolean expression for a rule is 1024 characters (bytes) plus a terminating NULL to end the string.

Argument

The smallest component of a rule that can be evaluated. This consists of a field name, a Rules comparison operator, and another field name (field to field comparisons), a static value (static comparisons), or nothing (existence operators).

The predefined MQSeries Integrator Rules operators contain a type in uppercase (e.g, STRING) and an operator (e.g., =) concatenated with no spaces.

There must be at least one space between the field name and the Rules operator and between the Rules operator and the comparison value. The EXIST and NOT_EXIST operators must be followed by at least one space before a parenthesis or a Boolean operator.

If the field name or static comparison value contains spaces, quotes, or parentheses, the item must be enclosed in quotes (either single or double--whatever the value does NOT contain). A value may not have both single and double quotes. If the Rules operator is a DATE, TIME, or DATETIME operator, the static comparison value MUST have a four-digit year.

For the Management APIs, the value must be in ISO-8601:1988 standard format (YYMMDDhhmmss), with the TIME or DATE portions padded with zeros (0) if the operator is DATE or TIME, respectively.

Boolean Operator

A Boolean operator is either '&' (AND) or '|' (OR)

Field Name

A field name is defined by the user when an input format is defined. A rule's Message Type is the input format that must contain the field or contain a nested format that contains that field. If the field name contains spaces, quotes, or parentheses, the name must be enclosed in quotes (either single or double--whatever the name does NOT contain). A field name may not contain both single and double quotes.

Comparison Operator

Rules Comparison Operators are defined to be field existence, field non-existence, and the following operators: <,<=,>,>=,<>,= for INT (whole number), FLOAT (decimal number), DATE, TIME, DATETIME and STRING fields. Field-to-field comparisons (e.g., field1 compares against field2) and case-sensitive string (e.g., where "a" does not equal "A") comparisons are also possible.

Parentheses

Arguments can be grouped in parentheses based on Boolean algebraic definitions:

1. Parentheses may surround a single complete argument.

   Example: (F1 INT= 1).

2. Parentheses may surround two or more arguments, separated by a Boolean "&" (AND) or "|"(OR).

   Example: (F1 INT= 1 & F2 INT= 2)

3. Parentheses must be balanced and in accordance with definition 1 and 2.

4. Parentheses may be nested within other parentheses in accordance with definitions 1, 2, and 3.

   Example: ((F1 INT= 1 | F2 INT= 2) & F3 INT= 3)

Boolean algebra dictates the appropriate placing of left and right parentheses in the Boolean expression.

Boolean Operator Precedence

Boolean algebra defines the AND operator as having higher precedence than the OR operator if no parentheses are present. This requirement was adhered to in the implementation of Rules Boolean Expressions.

If the following rule is defined:

F1 INT= 1 | F2 INT= 2 & F3 INT= 3

the rules evaluation API evaluates the expression as if parentheses were added as follows:

F1 INT= 1 | (F2 INT= 2 & F3 INT= 3).

Arguments in the innermost set of parentheses are evaluated first regardless of the Boolean operator for the arguments. The evaluation then progresses outward until the whole expression is evaluated.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See ***System Management for MQSeries Integrator*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. For more information, see section Operator Management APIs of this document.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Expression Management API Structures

NNRExp

Overview

NNRExp is passed as an argument in several Rules Management APIs to identify what rule owns the Expression. It should be cleared (see NNR_CLEAR) prior to use in a function call.

Syntax

```
    typedef struct NNRExp {
        char AppName[APP_NAME_LEN];
        char MsgName[MSG_NAME_LEN];
        char RuleName[RULE_NAME_LEN];
        long InitFlag;
}       NNRExp;
```

Parameters

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. As long as the user is using Formatter, the message type is the input format name.
RuleName [RULE_NAME_LEN]	char	Name of the rule to be evaluated within an application group and message name pair. This rule name is defined by the user.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRExpData

Overview

NNRExpData is passed as an argument in several Rules Management APIs affecting Rule expressions. It should be cleared (see NNR_CLEAR) prior to use in a function call.

Syntax

```
typedef struct NNRExpData {
       NNDate DateChange;
       int ChangeAction;
       long InitFlag
       NNDate EnableDate;
       NNDate DisableDate;
       char Expression[EXPRESSION_LEN];
           // This will always be the last data
       }
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
EnableDate	NNDate	Defaulted for now, provided for future capability.
DisableDate	NNDate	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).
Expression [EXPRESSION_LEN]	char	Boolean expression containing arguments together with the Boolean operators & (AND) and | (OR) with parentheses to determine order of evaluation.

Expression Management API Functions

NNRMgrAddExpression

Overview

NNRMgrAddExpression() adds an expression to a rule. A rule can have only one expression containing any number of arguments (see above definitions). NNRMgrAddExpression() can be called only once per rule. Prior to adding an expression, the user must define the application group, associated message type and rule using NNRMgrAddApp(), NNRMgrAddMsg(), and NNRMgrAddRule(). Before adding an expression, the user must also know the operator information, obtained using NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().

When adding expression information, user permission to update the rule will be checked. If the user is the owner or has update permission for the rule, he will be able to add the expression information. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Note

All users should use the NNRMgrAddExpression() API instead of NNRMgrAddArgument() when writing a new code.

Syntax

```
const long NNRMgrAddExpression (NNRMgr *pMgr,
                     const NNRExp* pRExp,
                     NNRExpData* pRExpData)
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRExp	const NNRExp *	Input	Should be populated prior to this function call. See the Rule Expression Management API structures description.

Name	Type	Input/ Output	Description
pRExpData	const NNRExpData *	Input	DateChange, ChangeAction, EnableDate and DisableDate should be set to NULL, these are provided only for future enhancements.

Remarks

To store data related to expressions, the application group, message type, and rule information must exist.

NNRMgrInit() should be called before NNRMgrAddExpression. A call to NNR_CLEAR for both pRExp and pRExpData should be made prior to populating the structures and calling this API.

Note

Field names are not checked for validity, and the validity of static comparison values are only checked for Date, Time, and DateTime operators. Static Date, Time, or DateTime comparisons values are valid if they comply with the ISO-8601:1988 standard notation. Date, Time, and DateTime static values in expressions must be specified in the YYYYMMDDhhmmss format. Consequently, Date values must have the Time component (hhmmss) padded with zeros, and Time values must have the Date component (YYYYMMDD) padded with zeros.

Return Value

Returns 1 if the expression was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrAddArgument(), NNRMgrUpdateArgument(), NNRMgrGetFirstArgument(), NNRMgrGetNext Argument(), NNRMgrDeleteEntireRule(), NNRMgrReadExpression(), NNRMgrUpdateExpression()

NNRMgrReadExpression

Overview

NNRMgrReadExpression() retrieves the rule expression associated with the application group, message type, and rule triplet. Prior to retrieving an expression, it must be defined. See NNRMgrAddApp(), NNRMgrAddMsg(), NNRMgrAddRule(), and NNRMgrAddExpression().

When retrieving the rule expression, user permission to read the rule is checked. If the user has read permission for the rule, he will be able to see the rule information. If the user attempting to access rule information does not have read access, an error will be returned, indicating that the user does not have read permission.

Syntax

```
const long NNRMgrReadExpression (NNRMgr *pMgr,
                                 const NNRExp *pRExp,
                                 NNRExpData* pRExpData)
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRExp	const NNRExp *	Input	Should be populated prior to this function call. See the Rule Expression Management API structures description.
pRExpData	const NNRExp Data *	Output	NNRMgrReadExpression () populates this structure. See the Rule Expression Management API structures description.

Remarks

To read expression data, the application group, message type and rule information (including the expression) must exist.

NNRMgrInit() should be called before NNRMgrReadExpression. A call to NNR_CLEAR for both pRExp and pRExpData should be made prior to populating the structures and calling this API.

Return Value

Returns 1 if the expression was added successfully; zero (0) if an error occured.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrAddArgument(), NNRMgrUpdateArgument(), NNRMgrGetFirstArgument(), NNRMgrGetNext Argument(), NNRMgrDeleteEntireRule(), NNRMgrAddExpression(), NNRMgrUpdateExpression()

NNRMgrUpdateExpression

Overview

NNRMgrUpdateExpression() updates an expression in a rule. Prior to adding an expression, the user must define the application group, associated message type, and rule using NNRMgrAddApp(), NNRMgrAddMsg(), and NNRMgrAddRule(). Before adding or updating an expression, the user must also know the operator information, obtained using NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().

When updating expression information, user permission to update the rule will be checked. If the user has update permission for the rule, the user can update the expression information. If the user attempting to update an expression does not have update access, an error will be returned indicating that the user does not have update permission and no change will occur.

Note

All users should use the NNRMgrUpdateExpression() API instead of NNRMgrUpdateArgument() when writing a new code.

Syntax

```
const long NNRMgrUpdateExpression(NNRMgr *pMgr,
                     const NNRExp *pRExp,
                     const NNRExpData *pRExpData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRExp	const NNRExp *	Input	Should be populated prior to this function call. See the Expression Management API structures description.
pRExpData	const NNRExp Data *	Input	DateChange, ChangeAction, EnableDate and DisableDate should be set to NULL, these are provided only for future enhancements.

Remarks

To update data related to expressions, the application group, message type and rule information (including the expression) must exist.

NNRMgrInit() should be called before NNRMgrUpdateExpression. A call to NNR_CLEAR for both pRExp and pRExpData should be made prior to populating the structures and calling this API.

Note

Field names are not checked for validity and the data type of comparison value is checked only for Date, Time, DateTime operators.

Return Value

Returns 1 if the expression was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrAddArgument(), NNRMgrUpdateArgument(), NNRMgrGetFirstArgument(), NNRMgrGetNext Argument(), NNRMgrDeleteEntireRule(), NNRMgrAddExpression(), NNRMgrReadExpression()

Argument Management APIs

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Note

For Rules release 1.0, all arguments must be active. Therefore, all inactive arguments must be activated or deleted during the database upgrade.

Note

Expression Management APIs should be used instead of Argument Management APIs because of the added capability of the Boolean OR operator.

Argument Management API Structures

NNRArg

Overview

NNRArg is passed as a pointer as the second parameter of select Argument Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Argument Management API calls.

Syntax

```
typedef struct NNRArg {
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       char RuleName[RULE_NAME_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. Using Formatter, the message type is the input format name.
RuleName [RULE_NAME_LEN]	char	Name of the rule to be evaluated within an application group and message name pair. This rule name is defined by the user.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRArgData

Overview

NNRArgData is passed as a pointer as the third parameter of select Argument Management APIs. The pointer cannot be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or by Argument Management API calls). Use of this structure is described in each Argument Management API section.

Syntax

```
typedef struct NNRArgData{
       NNDate DateChange;
       int ChangeAction;
       char FieldName[FIELD_NAME_LEN];
       int OperatorId;
       char SecondFieldName[SECOND_FIELD_NAME_LEN];
       char ArgValue[ARG_VALUE_LEN];
       int ArgActive;
       NNDate ArgEnableDate;
       NNDate ArgDisableDate;
       int ArgSequence;
       long InitFlag;
}
```

Members

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
FieldName [FIELD_NAME_ LEN]	char	Name of the field to which the operator will be applied.
OperatorId	int	ID retrieved by NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().
SecondFieldName [SECOND_FIELD_ NAME_LEN]	char	Value to which the field will be compared for a field to field operator.
ArgValue [ARG_VALUE_ LEN]	char	Value of the comparison (static value).
ArgActive	int	Specifies whether the argument is active (value of 1). For release 1.0, all arguments MUST be active.
ArgEnableDate	NNDate	For future enhancements, ignore for now.

Name	Type	Description
ArgDisableDate	NNDate	For future enhancements, ignore for now.
ArgSequence	int	Sequence of this argument within the rule.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRArgUpdate

Overview

NNArgUpdate is a structure containing update information for arguments contained within an application group / message type / rule. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Argument Management API calls.

Syntax

```
typedef struct NNRArgUpdate {
       NNDate DateChange;
       int ChangeAction;
       char FieldName[FIELD_NAME_LEN];
       int OperatorId;
       char SecondFieldName[SECOND_FIELD_NAME_LEN];
       char ArgValue[ARG_VALUE_LEN];
       int ArgActive;
       NNDate ArgEnableDate;
       NNDate ArgDisableDate;
       long InitFlag;
}
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
FieldName [FIELD_NAME_ LEN]	char	Name of the field to which the operator will be applied.
OperatorId	int	ID retrieved by NNRMgrReadFirstOperator() or NNRMgrReadNextOperator().
SecondFieldName [SECOND_FIELD_ NAME_LEN]	char	Value to which the field will be compared for a field to field operator.
ArgValue [ARG_VALUE_ LEN]	char	Value of the comparison (static value).
ArgActive	int	Value of 1 indicates that the argument is active, a value of zero (0) indicates that the argument is inactive. For release 1.0+, all arguments must be active.
ArgEnableDate	NNDate	Defaulted for now, provided for future capability.

Name	Type	Description
ArgDisableDate	NNDate	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Argument Management API Functions

NNRMgrAddArgument

Overview

Note

This functionality is provided for backwards compatibility. Use NNRMgrAddExpression() or NNRMgrUpdateExpression() API instead of NNRMgrAddArgument() or NNRMgrUpdateArgument().

NNRMgrAddArgument() adds one argument to a rule. If a rule has several arguments to be added, this function must be called once for each argument. Prior to adding an argument, the user must define the application group, associated message type, and rule using NNRMgrAddApp(), NNRMgrAddMsg(), and NNRMgrAddRule(). In addition, the user must know the operator information, obtained using NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().

When adding argument information, user permission to update the rule will be checked. If the user is the owner has update permission for the rule, the user can add the argument information. If the user attempting to add an argument does not have update access, an error will be returned indicating that the user does not have update permission and no change will occur.

Note

This functionality adds arguments if the Rule Expression has no arguments or has all arguments ANDed together without nested parentheses.

Syntax

```
const long NNRMgrAddArgument(NNRMgr *pMgr,
                             const NNRArg *pRArg,
                             const NNRArgData *pRArgData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRArg	const NNRArg *	Input	Should be populated prior to this function call. See the Argument Management API structures description.

Name	Type	Input/ Output	Description
pRArgData	const NRArgData *	Input	DateChange, ChangeAction, ArgumentEnableDate, and ArgumentDisableDate should be set to NULL. These are provided only for future enhancements.

Remarks

To store the argument, the application group, message type, and rule information must exist. NNRMgrInit() should be called prior to calling NNRMgrAddArgument(). A call to NNR_CLEAR for both pRArg and pRArgData should be made prior to populating the structures or calling this API. For MQSeries Integrator Rules release 1.0, all arguments must be active. Therefore, no inactive arguments will be added.

Return Value

Returns 1 if the argument was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: NNMgrAddExpression().

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstArgument(), NNRMgrGetNextArgument(), NNRMgrUpdateArgument(), NNRMgrDeleteEntireRule(), NNRMgrAddExpression(), NNRMgrReadExpression(), NNRMgrUpdateExpression()

NNRMgrUpdateArgument

Overview

Note

This functionality change is provided for backwards compatibility. Use the NNRMgrUpdateExpression() API instead of NNRMgrUpdateArgument() API.

NNRMgrUpdateArgument() enables the user to update the nth argument of a previously defined rule. The user provides the unique application group, message type, rule, and position to identify the argument (in the pRArg structure). The new information is in the pRArgUpdate structure.

The argument position represents the sequence number of the argument to be updated, starting from 1 and going to the end of the argument sequence. To change the first argument, set position to 1. To change the fifth argument, set position to 5, and so on.

This function can only update arguments if the Rule Expression has all arguments ANDed together with no nested parentheses.

When updating argument information, user permission to update the rule will be checked. If the user is the owner or has update permission for the rule, the user can update the rule information. If the user attempting to update an argument does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Note

This functionality updates arguments if the Rule Expression has only one argument or has all arguments ANDed together without nested parentheses.

Syntax

```
const long NNRMgrUpdateArgument (NNRMgr *pMgr,
                                 NNRArg * const pRArg,
                                 NNRArgUpdate *pRArgUpdate,
                                 int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRArg	NNRArg * const	Input	Should be populated prior to this function call. See the Argument Management API structures description.

Name	Type	Input/ Output	Description
pRArgUpdate	NNRArgUpdate *	Input	Should be populated prior to this function call. See the Argument Management API structures description.
position	int	Input	Numeric order of the argument to be updated.

Remarks

To update an argument, the rule and expression must exist. NNRMgrInit() should be called prior to any Rules Management API calls. For MQSeries Integrator Rules release 1.0+, all arguments must be active. Therefore, adding any inactive arguments will fail.

Return Value

Returns 1 if the argument was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: NNRMgrUpdateExpression().

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddArgument(), NNRMgrGetFirstArgument(), NNRMgrGetNextArgument(), NNRMgrReadExpression(), NNRMgrUpdateExpression, NNRMgrDeleteEntireRule()

NNRMgrGetFirstArgument

Overview

NNRMgrGetFirstArgument() provides a way of starting to retrieve information for a list of arguments associated with an application group, message type, and rule triplet. This API returns the first argument in the rule in the pRArgData parameter. Prior to retrieving an argument, it must be defined. See NNRMgrAddApp(), NNRMgrAddMsg(), NNRMgrAddRule(), NNRMgrAddArgument(), and NNRMgrAddExpression().

When retrieving argument information, user permission to read the rule will be checked. If the user is the owner or another user with read or update permissions for the rule, the user can see the rule information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

WARNING!

The arguments are not necessarily grouped together with the Boolean ‘&’ operator. If there is more than one argument, use the NNRMgrReadExpression() API to determine the Boolean operators.

Syntax

```
const long NNRMgrGetFirstArgument(NNRMgr *pMgr,
                          const NNRArg * pRArg,
                          NNRArgData * const pRArgData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRArg	const NNRArg *	Input	Should be populated prior to this API call. See the Argument Management API structures description.
pRArgData	`NNRArgData * const`	Output	NNRMgrGetFirstArgument() populates this structure. See the Argument Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetFirstArgument().

A call to NNR_CLEAR for both pRArg and pRArgData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the argument was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_ARGUMENTS, no arguments were found for the application group, message type, and rule name specified in the pRArg structure.

Example

See Sample Program 2: NNRMgrReadExpression().

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetNextArgument(), NNRMgrReadExpression().

NNRMgrGetNextArgument

Overview

NNRMgrGetNextArgument() provides a way of iterating through the arguments after the first argument has been retrieved (see NNRMgrGetFirstArgument()).

When retrieving argument information, user permission to read the rule will be checked. If the user is the owner or another user with read or update permissions for the rule, the user can see the rule information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

WARNING!

The arguments are not necessarily grouped together with the Boolean '&' operator. If there is more than one argument, the user should use the NNRMgrReadExpression() API to retrieve the Boolean operators.

Syntax

```
const long NNRMgrGetNextArgument (NNRMgr *pMgr,
                             NNRArgData * const pRArgData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of current Rules Management object. See NNRMgrInit().
pRArgData	`NNRArgData * const`	Output	NNRMgrGetNextArgument() populates this structure. See the Argument Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetNextArgument().

A call to NNR_CLEAR for both pRArg and pRArgData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the argument was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error returned is RERR_NO_MORE_ARGUMENTS, the end of the arguments list has been reached.

Example

See Sample Program 2: NNRMgrReadExpression().

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstArgument(), NNRMgrReadExpression.

Subscription Management APIs

Subscriptions are added to an Application Group / Message Type Rule Set. After they are added, subscriptions can be associated with multiple rules in the same Application Group / Message Type. The NNRMgrAddSubscription() API is used to add the subscription to the Rule Set (if no rule name is given), as well as associate the subscription to a rule. Subscriptions have permissions that permission work similarly to rule permissions.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Subscription Management API Structures

NNRSubs

Overview

NNRSubs is passed as a pointer as the second parameter of select Subscription Management APIs. This pointer may not be NULL. This structure must be populated by the user prior to calling any of the Subscription Management APIs, and should be initialized by calling NNR_CLEAR prior to populating all of the fields.

Syntax

```
typedef struct NNRSubs{
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       char RuleName[RULE_NAME_LEN];
       char SubsName[SUBS_NAME_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. Using Formatter, the message type is the input format name.
RuleName [RULE_NAME_LEN]	char	Name of the rule to be evaluated within an application group and message name pair. This rule name is defined by the user. This is required only when adding a subscription to a specific rule. It is ignored for action, option, update, and delete functions.
SubsName [SUBS_NAME_LEN]	char	Name of the subscription associated with a message name and application group.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRSubsData

Overview

NNRSubsData is passed as a pointer as the third parameter of select Subscription Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or by Subscription Management API calls). Use of this structure is described in each Subscription Management API section.

Syntax

```
typedef struct NNRSubsData{
       NNDate DateChange;
       int ChangeAction;
       int SubsActive;
       NNDate SubsEnableDate;
       NNDate SubsDisableDate;
       char SubsOwner[SUBS_OWNER_LEN];
       char SubsComment[SUBS_COMMENT_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
SubsActive	int	Provided for future enhancement for activating and inactivating subscriptions, active is defined by a value 1 and inactive is defined by value zero (0).
SubsEnableDate	NNDate	Provided for future functionality, ignored for now.
SubsDisableDate	NNDate	Provided for future functionality, ignored for now.
SubsOwner [SUBS_OWNER_ LEN]	char	Name of the owner of the subscription.
SubsComment [SUBS_COMMENT_ LEN]	char	Information details about the subscription.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRSubsReadData

Overview

NNRSubsReadData is a structure containing subscription information after a subscription read operation.

Syntax

```
typedef struct NNRSubsReadData{
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       char RuleName[RULE_NAME_LEN];
       char SubsName[SUBS_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       int SubsActive;
       NNDate SubsEnableDate;
       NNDate SubsDisableDate;
       char SubsOwner[SUBS_OWNER_LEN];
       char SubsComment[SUBS_COMMENT_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group to identify the subscription.
MsgName [MSG_NAME_LEN]	char	Name of the message type to identify the subscription.
RuleName [RULE_NAME_LEN]	char	Name of the rule to link to the subscription, if provided.
SubsName [SUBS_NAME_LEN]	char	Name of the subscription to be read.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
SubsActive	int	Value of 1 indicates that the subscription is active, a value of zero (0) indicates that the subscription is inactive.
SubsEnableDate	NNDate	Defaulted for now, provided for future capability.
SubsDisableDate	NNDate	Defaulted for now, provided for future capability.

Name	Type	Description
SubsOwner [SUBS_OWNER_ LEN]	char	Contains the name of the subscription owner.
SubsComment [SUBS_COMMENT_ LEN]	char	Contains the subscription owner's comment.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRSubsUpdate

Overview

NNRSubsUpdate contains update information for subscriptions. The pointer must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Subscription Management API calls.

Syntax

```
typedef struct NNRSubsUpdate {
       char SubsName[SUBS_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       int SubsActive;
       NNDate SubsEnableDate;
       NNDate SubsDisableDate;
       char SubsOwner[SUBS_OWNER_LEN];
       char SubsComment[SUBS_COMMENT_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
SubsName [SUBS_NAME_LEN]	char	New name for the subscription to be updated.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
SubsActive	int	Value of 1 indicates that the subscription is active, a value of zero (0) indicates that the subscription is inactive.
SubsEnableDate	NNDate	Defaulted for now, provided for future capability.
SubsDisableDate	NNDate	Defaulted for now, provided for future capability.
SubsOwner [SUBS_OWNER_ LEN]	char	Defaulted for now, provided for future capability.
SubsComment [SUBS_COMMENT_ LEN]	char	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Subscription Management API Functions

NNRMgrAddSubscription

Overview

NNRMgrAddSubscription() adds subscription maintenance information for one subscription. If the user wants more than one subscription for the rule or rule set, this function must be called once for each subscription. The user can either supply a rule name or not. The subscription is created if it does not already exist in the Rule set. If the rule name is provided, the subscription is associated with that rule, if the user has update permission for the rule. The user entering the subscription is identified and stored as its owner and is automatically granted update and read permission for the subscription. PUBLIC is automatically granted read permission for the subscription.

When adding subscription information to a rule, user permission to update the rule will be checked. If the user is the owner or another user with update permission for the rule, the user can add the subscription information. If the user attempting to add a subscription does not have update access, an error will be returned indicating that the user does not have update permission and no change will occur.

Syntax

```
const long NNRMgrAddSubscription(NNRMgr *pMgr,
                         const NNRSubs *pRSubs,
                         const NNRSubsData *pRSubsData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	const NNRSubs *	Input	Should be populated prior to this function call. See the Subscription Management API structures description. Users need not to specify the rule name.
pRSubsData	const NNRSubsData *	Input	Should be populated prior to calling this function. DateChange, ChangeAction, SubsEnableDate and SubsDisableDate should be set to NULL. They are provided only for future enhancements. SubsActive is defaulted to 1.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddSubscription().

A call to NNR_CLEAR for both pRSubs and pRSubsData should be made prior to populating the structures or calling this API.

pRSubs: If a rule name is provided, the function will check to see if the subscription if the subscription already exists in the rule set. If the subscription exists, it then checks to see if the rule already has the subscription. If so, the function will fail and set the error code to RERR_SUBS_NAME_ALREADY_EXISTS. If not, the function adds the subscription to the rule.

If the rule name is provided, and the subscription does not exist in the rule set, the function will create the subscription and automatically add it to the rule.

If the user does not provide the rule name, the function NNRMgrAddSubscription() will check to see if the subscription exists in the rule set. If the subscription already exists, the function will be set to the RERR_SUBS_ALREADY_EXISTS_IN_RULESET error code. If not, the function will create the subscription.

Return Value

Returns 1 if the subscription was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See *Sample Program 2: Rules Management API*.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddRule(), NNRMgrUpdateOwnerPerm(), NNRMgrUpdatePublicPerm(), NNRMgrReadSubscription()

NNRMgrReadSubscription

Overview

NNRMgrReadSubscription() reads subscription maintenance information for one subscription.

When retrieving subscription information, user permission to read the subscription will be checked. If the user is the owner or a user with read or update permissions for the subscription, the user can see the subscription. If the user attempting to access subscription information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission. The subscription read permission is also checked when reading an action or option in the subscription. If the rule name is given, the rule is checked for read permission and association with the subscription.

Syntax

```
const long NNRMgrReadSubscription(NNRMgr *pMgr,
                     const NNRSubs *pRSubs,
                     NNRSubsData* const pRSubsData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	const NNRSubs *	Input	Should be populated prior to this function call. See the Subscription Management API structures description. The rule name does not need to be provided in the NNRSubs structure pointed to by pRSubs.
pRSubsData	NNRSubsData* const	Output	NNRMgrReadSubscription( ) will populate this structure. See the Subscription Management API structures description. If DateChange is non-NULL, the subscription exists.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrReadSubscription().

A call to NNR_CLEAR for both pRSubs and pRSubsData should be made prior to populating the structures or calling this API.

pRSubs: If a rule name is provided, this function will verify whether the subscription exists for the rule name and will check rule permission. If the rule name is not provided, the function will verify whether the subscription exists in the rule set.

Return Value

Returns 1 if the subscription was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR,NNRMgrAddSubscription()

NNRMgrUpdateSubscription

Overview

NNRMgrUpdateSubscription() enables the user to update a subscription. The user provides the unique application group, message type, and subscription name to identify the subscription to be updated (in the pRSubs structure) and the new information in the pRSubsUpdate structure.

When updating subscription information, user permission to update the subscription will be checked. If the user is the owner or another user and he has update permission, the user can update the subscription information. If the user attempting to update a subscription does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Note

The subscription Update permission is also checked when an action or option is either added or updated in the subscription.

Syntax

```
const long NNRMgrUpdateSubscription (NNRMgr *pMgr,
                     const NNRSubs *pRSubs,
                     const NNRSubsUpdate *pRSubsUpdate);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	const NNRSubs *	Input	Should be populated prior to this function call. See the Subscription Management API structures description. Users need not to specify a rule name; the name is ignored.
pRSubsUpdate	const NNRSubsUpdate *	Input	Should be populated prior to this function call. See the Subscription Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

pRSubs: The rule name does not need to be in the NNRSubs structure pointed to by pRSubs; the name is ignored. However, all the changes made to the subscription will be made globally within the rule set.

Return Value

Returns 1 if the subscription was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRSubs          key;
struct NNRSubsUpdate    data;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter subscription name \n>";
cin >> key.SubsName;

cout << "Enter New subscription name \n>";
cin >> data.SubsName;
cout << "Enter new subscription owner \n>";
cin >> data.SubsOwner;
cout << "Enter new subscription comment \n>";
cin >> data.SubsComment;
if (NNRMgrUpdateSubscription(pmgr, &key, &data)) {
        cout    << endl
               << "\tSubs Name: " << key.SubsName << " Changed."
                << endl << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddSubscription(), NNRMgrReadSubscription(), NNRMgrGetFirstSubscription(), NNRMgrGetNextSubscription()

NNRMgrGetFirstSubscription

Overview

NNRMgrGetFirstSubscription() and NNRMgrGetNextSubscription() enable the user to iterate through the subscriptions associated with the application group, message type and, optionally, the rule name. Call NNRMgrGetFirstSubscription(), then NNRMgrGetNextSubscription().

When retrieving subscription information, user permission to read the subscription will be checked. If the user is the owner or another user and he has read permission for the subscription, they can to see the information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission. If the rule name is not provided, all subscriptions will be retrieved for the rule set.

Syntax

```
const long NNRMgrGetFirstSubscription (
              NNRMgr *pMgr,
              const NNRSubs *pRSubs,
              NNRSubsReadData * const pRSubsReadData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	const NNRSubs *	Input	Should be completely populated except for the SubscriptionName field prior to this function call. See the Subscription Management API structures description. User need not to specify a rule name.
pRSubsReadData	NNRSubsRead Data * const	Output	Will be populated by this function call. See the Subscription Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

pRSubs: The rule name does not need to be provided in the NNRSubs structure pointed to by pRSubs. If provided, the function will retrieve the first subscription associated with the rule. If not provided, the function will retrieve the first subscription associated with the rule set.

Return Value

Returns 1 if the subscription was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_SUBSCRIPTIONS, no subscriptions were found for the application group and message type specified in the pRSubs structure.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRSubs          key;
struct NNRSubsReadData  data;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter rule name \n>";
cin >> key.RuleName;

int iret = NNRMgrGetFirstSubscription(pmgr, &key, &data);
if ( iret )
{
        printSubscription( &key, &data );
        while( NNRMgrGetNextSubscription(pmgr, &data) )
        {
                printSubscription( &key, &data );
        }
}
else
{
        cout << endl << "Read failed." << endl << endl << endl;
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddSubscription(), NNRMgrReadSubscription(), NNRMgrGetNextSubscription(), NNRMgrUpdateSubscription()

NNRMgrGetNextSubscription

Overview

NNRMgrGetFirstSubscription() and NNRMgrGetNextSubscription() enable the user to iterate through the subscriptions associated with the application group, message type and, optionally, the rule name. Call GetFirst before GetNext.

When retrieving subscription information, user permission to read both the rule and the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the information. If the user attempting to access subscription information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission. The subscription read permission is also checked when reading an action or option in the subscription

Syntax

```
const long NNRMgrGetNextSubscription (NNRMgr *pMgr,
                NNRSubsReadData * const pRSubsReadData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubsReadData	NNRSubsRead Data * const	Output	Will be populated by this function call. See the Subscription Management API structures description.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the subscription was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_SUBSCRIPTIONS, the end of the subscriptions list has been reached.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
```

```
InitNNRMgrSession(pmgr, session);

struct NNRSubs          key;
struct NNRSubsReadData  data;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter rule name \n>";
cin >> key.RuleName;

int iret = NNRMgrGetFirstSubscription(pmgr, &key, &data);
if ( iret )
{
        printSubscription( &key, &data );
        while( NNRMgrGetNextSubscription(pmgr, &data) )
        {
                printSubscription( &key, &data );
        }
}
else
{
        cout << endl << "Read failed." << endl << endl << endl;
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddSubscription(), NNRMgrReadSubscription(), NNRMgrGetFirstSubscription(), NNRMgrUpdateSubscription()

NNRMgrDuplicateSubscription

Overview

NNRMgrDuplicateSubscription() creates a new subscription based on the subscription name provided. The new subscription will have the name provided in the pNewSubsName and inherit all other properties from the existing subscription provided in pSubs.SubsName.The user must have read permission to the subscription to duplicate it.

Syntax

```
const long NNRMgrDuplicateSubscription (
            NNRMgr *pMgr,
            const NNRSubs* pSubs,
            const char * const pNewSubsName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pSub	const NNRSubs*	Input	Should be populated prior to this function call. See the Subscription Management API structures description.
NewSubsName	const char* const	Input	Name of duplicate specified subscription.

Return Value

Returns 1 if the subscription duplicated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See *Sample Program 2: Rules Management API.*

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstUsingSubs(), NNRMgrGetNextRuleUsingSubs()

NNRMgrDeleteSubscriptionFromRule

Overview

NNRMgrDeleteSubscriptionFromRule disassociates a subscription from its rule if the user has update permission for the rule. Only a subscription that is not associated with any rule can be deleted from the Rule Set by using NNRMgrDeleteEntireSubscription().

Syntax

```
const long NNRMgrDeleteSubscriptionFromRule (
            NNRMgr *pMgr,
            const NNRRule *pRRule,
            const char * SubsName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRRule	pRRule	Input	The unique rule definition.
SubsName	const char* const	Input	Name of subscription.

Remarks

A call to NNR_CLEAR for pRRule should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the user has update permission for the rule and is deleting the subscription and the subscription is successfully deleted. Returns zero (0) if an error occurred. An error will occur if the user does not have update permission.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See *Sample Program 2: Rules Management API.*

See Also

NNRMgrDeleteEntireSubscription()

NNRMgrDeleteEntireSubscription

Overview

NNRMgrDeleteEnireSubscription deletes a subscription and its actions and options from the specified rule. If the subscription is associated with any rules, an error will be returned.

When deleting subscription information, user permission to update the subscription will be checked. If the user is the owner and has update permissions for the subscription, the subscription.will be deleted. If the user is not the owner but does have update access, the subscription will be set to inactive but not deleted. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrDeleteEntireSubscription (
             NNRMgr *pMgr,
             const NNRMSubs *pRSubs,
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	NNRMSubs	Input	The.unique identifier for the subscription with the application group name, message type name, and subscription name.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the subscription was deleted successfully; 2 if the subscription was deactivated; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See *Sample Program 2: Rules Management API.*

See Also

`NNRMgrDeleteSubscriptionFromRule()`

NNRMgrGetFirstRuleUsingSubs

Overview

NNRMgrGetFirstRuleUsingSubs() enables the user to iterate through the rules associated with a subscription. If there are any rules using the subscription, the name of the first rule is returned in NpRSubsReadData:RuleName.

When retrieving subscription information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for subscription, the user can see the information. If the user attempting to access subscription information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission. The subscription read permission is also checked when the user is reading an action or option in the subscription.

Syntax

```
const long NNRMgrGetFirstRuleUsingSubs (
              NNRMgr *pMgr,
              const NNRSubs *pRSubs,
              char* const pRuleName
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRSubs	const NNRSubs *	Input	Should be completely populated except for the Subscription Name field prior to this function call. See the Subscription Management API structures description. User should not specify a rule name.
pRuleName	char* const	Output	Will be populated by this function call.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

pRSubs: The rule name should not be provided in the NNRSubs structure pointed to by pRSubs.

Return Value

Returns 1 if the rules were retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_RULES, no rules were found for the application group, message type, and rule name specified in the pRSubs structure.

Example

See *Sample Program 2: Rules Management API*.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddSubscription(), NNRMgrReadSubscription(), NNRMgrGetFirstSubscription(), NNRMgrUpdateSubscription() NNRMgrGetNextRuleUsingSubs()

NNRMgrGetNextRuleUsingSubs

Overview

NNRMgrGetFirstRuleUsingSubs() and NNRMgrGetNextRuleUsingSubs() enable the user to iterate through the subscriptions associated with a rule. Call NNRMgrGetFirstRuleUsingSubs before NNRMgrGetNextRuleUsingSubs.

When retrieving subscription information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the information. If the user attempting to access subscription information does not have a minimum of read access, an error will be returned indicating that the user does not have read permission. The subscription read permission is also checked when reading an action or option in the subscription

Syntax

```
const long NNRMgrGetNextRuleUsingSubs (NNRMgr *pMgr,
            char* const pRuleName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRuleName	char* const	Output	Will be populated by this function call.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

The rule name should not need to be provided in the NNRSubs structure pointed to by pRSubs.

Return Value

Returns 1 if the rule was retrieved successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_RULES, the end of the rule list has been reached.

Example

See *Sample Program 2: Rules Management API.*

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddSubscription(), NNRMgrReadSubscription(), NNRMgrGetFirstSubscription(), NNRMgrUpdateSubscription() NNRMgrGetFirstRuleUsingSubs()

Action Management APIs

Action are commands that should be used if a rule evaluates as true and the subscription is performed. A subscription includes actions that contain option name-value pairs.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Action Management API Structures

NNRAction

Overview

NNRAction is passed as a pointer as the second parameter of select Action Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Action Management API calls.

Syntax

```
typedef struct NNRAction{
       char AppName[APP_NAME_LEN];
       char MsgName[MSG_NAME_LEN];
       char RuleName[RULE_NAME_LEN];
       char SubsName[SUBS_NAME_LEN];
       char ActionName[ACTION_NAME_LEN];
       char OptionName[OPTION_NAME_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
AppName [APP_NAME_LEN]	char	Name of the application group (defined by the user). Should be the application group in which the user is defining rules for evaluation.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. As long as the user is using Formatter, the message type is the input format name.
RuleName [RULE_NAME_LEN]	char	The rule name is ignored for actions and options.
SubsName [SUBS_NAME_LEN]	char	Name of the subscription associated with a rule name, message name, and application group.
ActionName [ACTION_NAME_ LEN]	char	Name of the action associated with this subscription.
OptionName [OPTION_NAME_LEN]	char	Name of the first option associated with this action.

Name	Type	Description
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRActionData

Overview

NNRActionData is passed as a pointer as the third parameter of the Add Action Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to Add Action Management API calls. Use of this structure is described in the Add Action Management API section.

Syntax

```
typedef struct NNRActionData{
       NNDate DateChange;
       int ChangeAction;
       char OptionValue[OPTION_NAME_LEN];
       long InitFlag;
)
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
OptionValue [OPTION_NAME_ LEN]	char	Value of the first option.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRActionReadData

Overview

NNRActionReadData is passed as a pointer as the third parameter of select Action Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or by Action Management API calls). Use of this structure is described in each Read Action Management API section.

Syntax

```
typedef struct NNRActionReadData{
       NNDate DateChange;
       int ChangeAction;
       int ActionSequence;
       char ActionName[ACTION_NAME_LEN];
       char OptionName[OPTION_NAME_LEN];
       char OptionValue[OPTION_VALUE_LEN];
       long InitFlag;
} NNRMgrGetNextArgument
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
ActionSequence	int	Sequence of this action within its subscription. For example, for the first action, ActionSequence=1.
ActionName [ACTION_NAME_LEN]	char	Name of the action associated with the subscription.
OptionName [OPTION_NAME_LEN]	char	Name of the first option associated with the action.
OptionValue [OPTION_VALUE_LEN]	char	Static value of the first option if there are no actions.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNRActionUpdate

Overview

NNRActionUpdate contains update information for actions. The pointer must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Action Management API calls.

Syntax

```
typedef struct NNRActionUpdate{
       char ActionName[ACTION_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       long InitFlag;
}
```

Parameters

Name	Type	Description
ActionName [ACTION_NAME_LEN]	char	New name of the action to be updated.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Action Management API Functions

NNRMgrAddAction

Overview

NNRMgrAddAction() adds both an action and its first option. All other options must be added using NNRMgrAddOption(). Prior to adding an action, the application group, message type, and subscription must have been added using NNRMgrAddApp(), NNRMgrAddMsg(), and NNRMgrAddSubscription().

When adding action information, user permission to update the subscription will be checked. If the user is the owner or another user with update permission for the subscription, the user can add the action information. If the user attempting to add an action does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrAddAction(NNRMgr *pMgr,
                     const NNRAction *pRAction,
                     const NNRActionData *pRActionData,
                     int *pActionId);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRAction	const NNRAction *	Input	Should be populated prior to this function call. The rule name is ignored. See the Action Management API structures description.
pRActionData	const NNRActionData *	Input	DateChange and ChangeAction should be populated with NULL since they are provided only for future enhancements.

Name	Type	Input/ Output	Description
pActionId	int *	Input	Value of the action identifier used to insert all but the first option for an action (see NNRAddOption()).

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddAction().

A call to NNR_CLEAR for both pRAction and pRActionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the action was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR,NNRMgrGetFirstAction(), NNRMgrGetNextAction()

NNRMgrUpdateAction

Overview

NNRMgrUpdateAction() enables the user to update an action for a previously defined subscription. NNRMgrUpdateAction() only changes the action name. If options must be updated, the Option Management APIs should be used.

The action position represents the sequence number of the action to be updated, starting from 1 and going to the end of the action sequence. To change the first action, set position to 1. To change the fifth action, set position to 5, and so on.

When updating action information, user permission to update the subscription will be checked. If the user is the owner or another user with update permission for the subscription, the user can update the action information. If the user attempting to update an action does not have update access, an error will be returned indicating that the user does not have update permission and no change will occur.

Syntax

```
const long NNRMgrUpdateAction (NNRMgr *pMgr,
                const NNRAction *pRAction,
                const NNRActionUpdate *pRActionUpdate,
                int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRAction	const NNRAction *	Input	Should be populated prior to this function call. The rule name is ignored. See the Action Management API structures description.
pRActionUpdate	const NNRAction Update *	Input	Should be populated prior to this function call. See the Action Management API structures description.
position	int	Input	Numeric order of the action to be updated.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the action was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRAction                key;
struct NNRActionUpdate  data;
int ActionId = -1;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter subscription name \n>";
cin >> key.SubsName;
cout << "Enter action ID \n>";
cin >> ActionId;
cout << "Enter new action name \n>";
cin >> data.ActionName;

if (NNRMgrUpdateAction(pmgr, &key, &data, ActionId)) {
        cout    << endl
                << "\tAction Name: " << key.ActionName << "
Updated."
                << endl;
        cout    << endl
                << "\tAction id: " << ActionId << endl << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddAction(), NNRMgrGetFirstAction(), NNRMgrGetNextAction(), NNRMgrResequenceAction()

NNRMgrResequenceAction

Overview

NNRMgrResequenceAction() enables the user to resequence actions within a subscription. Given the current numeric position of the action, NNRMgrResequenceAction() will move the action to the specified new position. The user provides the unique application group, message type, subscription name, current position for the action to move and the position to move it to.

For example, if the following actions exist:

```
putqueue(TargetQ, MessageType)
reformat(inputformat, outputformat)
```

If you realize that the reformat should occur before the putqueue, you can call NNRMgrResequenceAction(), providing action 2 as the action to be moved and action 1 as the new position. This results in:

```
reformat(inputformat, outputformat)
putqueue(TargetQ, MessageType)
```

To indicate the first action to move in an action sequence, oldPosition can be set to either NNRRB_START or to the number 1. To specify the last action to move in an action sequence, set oldPosition to NNRRB_END.

To move an action to the end of an action sequence, set newPosition to NNRRB_END. To move an action to the start of an action sequence, set newPosition to NNRRB_START, or to the number 1.

If oldPosition or newPosition is greater than the maximum action/option sequence, it is changed to the maximum action sequence.

When updating action information, user permission to update the rule will be checked. If the user is the owner or another user with update permission for the subscription, the user can update the action information. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrResequenceAction (NNRMgr *pMgr,
                                   const NNRAction *pRAction,
                                   int oldPosition,
                                   int newPosition);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().

Name	Type	Input/ Output	Description
pRAction	const NNRAction *	Input	Should be populated prior to this function call. The rule name is ignored. See the Action Management API structures description.
oldPosition	int	Input	Old numeric order of the action to be resequenced.
newPosition	int	Input	New numeric order of the action to be resequenced.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Rules Management resequence boundaries are held in the following structure:

```
typedef enum NNRReseqBounds {
    NNRRB_END= -1,
    NNRRB_START=  1
} NNRReseqBounds;
```

Return Value

Returns 1 if the action was resequenced successfully; zero (0) if an error occurred.

If either oldPosition or newPosition are negative and not equal to NNRRB_END, an error condition is returned and errVal is set to RERR_INVALID_ACTION_PARAM.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNRAction                key;
struct NNRActionUpdate  data;
int oldActionSeq, newActionSeq;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter subscription name \n>";
cin >> key.SubsName;
cout << "Enter old action sequence \n>";
```

```
cin >> oldActionSeq;
cout << "Enter new action sequence \n>";
cin >> newActionSeq;

if (NNRMgrResequenceAction(pmgr, &key, oldActionSeq,
newActionSeq)) {
        cout    << endl
                << "\tAction Name: " << key.ActionName <<
"Resequenced."
                << endl;
        cout    << endl
               << "\tOld Action id: " << oldActionSeq << endl <<
endl;
                CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}
CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddAction(), NNRMgrGetFirstAction(), NNRMgrGetNextAction(), NNRMgrUpdateAction()

NNRMgrGetFirstAction

Overview

NNRMgrGetFirstAction() provides a way of starting to retrieve information for a list of actions associated with an application group, message type, rule and subscription. This API returns the first action in the subscription in the pRActionData parameter. Prior to retrieving an action, actions must be defined, (see NNRMgrAddApp(), NNRMgrAddMsg(), NNRMgrAddRule(), NNRMgrAddSubscription(), NNRMgrAddAction() and NNRMgrAddOption()).

When retrieving action information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the rule information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetFirstAction(
               NNRMgr *pMgr,
               const NNRAction * pRAction,
               NNRActionReadData * const pRActionData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRAction	const NNRAction *	Input	Should be populated prior to this function call. See the Action Management API structures description. Note that RuleName, ActionName, and OptionName do not need to be populated before this call.
pRActionData	NNRActionRead Data * const	Output	NNRMgrGetFirst Action() will populate this structure. See the Action Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetFirstAction().

A call to NNR_CLEAR for both pRAction and pRActionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the action was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_ACTIONS, no actions were found for the application group and message type specified in the pRAction structure.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetNextAction()

NNRMgrGetNextAction

Overview

NNRMgrGetNextArgument() provides a way of iterating through the actions after the first action has been retrieved. See NNRMgrGetFirstAction().

When retrieving action information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the action information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetNextAction(
            NNRMgr *pMgr,
            NNRActionReadData * const pRActionData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pRActionData	`NNRActionRead Data * const`	Output	NNRMgrGetNextAction() will populate this structure. See the Action Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetNextAction(). A call to NNR_CLEAR for both pRAction and pRActionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the action was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_ACTIONS, the end of the actions list has been reached.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstAction()

Option Management APIs

Options are name-value pairs that further define an action. The first option is added with the action, and additional options must be added with NNRMgrAddOption().

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one rule "r1" and another rule "R1". In a case-insensitive environment, you must make each item unique using something other than case differences.

If importing components exported from a context-sensitive database into a context-insensitive database, these differences will cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See the ***MQSeries Integrator System Management Guide*** for information on using NNRie.

Also, case-sensitive operators (see Operator Management APIs) may not work correctly on case-insensitive databases.

See the ***MQSeries Integrator System Management Guide*** for information on how to change a current case-insensitive installation to case sensitive.

Option Management API Structures

NNROption

Overview

NNROption is passed as a pointer as the second parameter of select Option Management APIs. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Option Management API calls.

Syntax

```
typedef struct NNROptionReadData {
       NNDate DateChange;
       int ChangeAction;
       char ActionName[ACTION_NAME_LEN]
       Int ActionSequence;
       char OptionName[OPTION_NAME_LEN];
       char OptionValue[OPTION_VALUE_LEN];
       int OptionSequence;
       long InitFlag;
}
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
ActionName [ACTION_NAME_LEN]	char	Name of action.
ActionSequence	int	Sequence of this action within its subscription. For example, for the first action, ActionSequence=1.
OptionName [Option_NAME_LEN]	char	Name of the first option associated with this action.
OptionValue [OPTION_VALUE_LEN]	char	Value of the option. This must not be NULL since this function adds an option.
OptionSequence	int	Sequence of this option within its action. For example, for the first option, OptionSequence=1.

Name	Type	Description
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNROptionData

Overview

NNROptionData is passed as a pointer as the third parameter of the Option Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to Option Management API calls. Use of this structure is described in each Option Management API section.

Syntax

```
typedef struct NNROptionData{
       NNDate DateChange;
       int ChangeAction;
       char OptionValue[OPTION_VALUE_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
OptionValue [OPTION_VALUE_LEN]	char	Value of the option. This must not be NULL since this function adds an option.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNROptionReadData

Overview

NNROptionReadData is passed as a pointer as a parameter of select Option Management APIs. The pointer may not be NULL and must be cleared (see NNR_CLEAR) prior to being populated (either by the user or by Option Management API calls). Use of this structure is described in each Option Management API section.

Syntax

```
typedef struct NNROptionReadData{
       NNDate DateChange;
       int ChangeAction;
       char ActionName[ACTION_NAME_LEN]
       int ActionSequence;
       char OptionName[OPTION_NAME_LEN]
       char OptionValue[OPTION_VALUE_LEN];
       int OptionSequence
       long InitFlag;
}
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
ActionName [ACTION_NAME_LEN]	char	Name of action.
ActionSequence	int	Sequence of this action within its subscription. For example, for the first action, ActionSequence=1.
OptionName [ACTION_NAME_LEN]	char	Name of option.
OptionValue [OPTION_VALUE_LEN]	char	Static value of the option. If there are no options, this must not be NULL since this function adds an option.
OptionSequence	int	Sequence of this option within its action. For example, for the first option, OptionSequence=1.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

NNROptionUpdate

Overview

NNROptionUpdate is passed as a pointer as a parameter of select functions in the Option Management API. The pointer may not be NULL, must be cleared (using NNR_CLEAR) prior to being populated, and must be populated prior to any Option Management API calls.

Syntax

```
typedef struct NNROptionUpdate{
       char OptionName[OPTION_NAME_LEN];
       NNDate DateChange;
       int ChangeAction;
       char OptionValue[OPTION_VALUE_LEN];
       long InitFlag;
}
```

Parameters

Name	Type	Description
OptionName [OPTION_NAME_LEN]	char	Name of the option to be updated.
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
OptionValue [OPTION_VALUE_LEN]	char	Value of the option to be updated.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API (see NNR_CLEAR).

Option Management API Functions

NNRMgrAddOption

Overview

If an action has more than one option, NNRMgrAddOption() is used to add all but the first option. Prior to adding more options, the user must define the first action and first option pair using NNRMgrAddAction().

When adding option information, user permission to update the subscription will be checked. If the user is the owner or another user with update permission for the subscription, the user can add the option information. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrAddOption(
                  NNRMgr *pMGR,
                  const NNROption *pROption,
                  const NNROptionData *pROptionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
NNROption	const NNROption *	Input	Should be populated prior to this function call.The rule name is ignored. See the Option Management API structures description.
NNROptionData	const NNROptionData *	Input	DateChange and ChangeAction should be populated with NULL since they are provided only for future enhancements.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrAddOption().

A call to NNR_CLEAR for both NNROption and NNROptionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the option was added successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstOption(), NNRMgrGetNextOption()

NNRMgrUpdateOption

Overview

NNRMgrUpdateOption() enables the user to update an action for an existing subscription. The user provides the unique application group, message type, and subscription name, and defines the option to change (in the pROption structure). The new information is provided in the pROptionUpdate structure.

The option position represents the sequence number of the option to be updated, starting from 1 and going to the end of the option sequence. To change the first option, set position to 1. To change the fifth option, set position to 5, and so on.

When updating option information, user permission to update the subscription will be checked. The user or owner has update permission for the rule and will be able to update the rule information. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
Const long NNRMgrUpdateOption (
                  NNRMgr *pMgr,
                  const NNROption *pROption,
                  const NNROptionUpdate *pROptionUpdate,
                  int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pROption	const NNROption *	Input	Should be populated prior to this function call. See the Option Management API structures description.
pROptionUpdate	const NNROption Update *	Input	Should be populated prior to this function call. The rule nmae is ignored. See the Option Management API structures description.
position	int	Input	Numeric order of the action to be updated.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Return Value

Returns 1 if the option was updated successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNROption                key;
struct NNROptionUpdate  data;
int position;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter subscription name \n>";
cin >> key.SubsName;
cout << "Enter action id \n>";
cin >> key.ActionId;
cout << "Enter option id \n>";
cin >> position;
cout << "Enter new option name \n>";
cin >> data.OptionName;
cout << "Enter new option value \n>";
cin >> data.OptionValue;

if (NNRMgrUpdateOption(pmgr, &key, &data, position)) {
        cout    << endl
                << "\tOption Name: " << key.OptionName << "
Changed."
                << endl << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}

CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddOption(), NNRMgrGetFirstOption(), NNRMgrGetNextOption(), NNRMgrResequenceOption()

NNRMgrResequenceOption

Overview

NNRMgrResequenceOption() enables the user to resequence options within an action. Given the current numeric position of the option, NNRMgrResequenceOption() will move the option to the specified new position. The user provides the unique application group, message type, rule name, subscription name, and current position for the option to move and the position to move it to.

For example, the following action/option information exists:

exec(process, argument1, argument2, argument3)

To switch argument2 and argument3, a call to NNRMgrResequenceOption would switch the option in position 4 (argument3) to the option in position 3. The option in position 3 (argument2) would then reside in position 4:

exec(process, argument1, argument3, argument2)

To indicate the first option to move in an option sequence, oldPosition can be set to either NNRRB_START or to the number 1. To specify the last option to move in an option sequence, set oldPosition to NNRRB_END.

To move an option to the end of an option sequence, set newPosition to NNRRB_END. To move an option to the start of an option sequence, set newPosition to NNRRB_START, or to the number 1.

If oldPosition or newPosition is greater than the maximum action/option sequence, it is changed to the maximum option sequence.

When updating option information, user permission to update the subscription will be checked. If the user is the owner or another user with update permission for the subscription, the user can update the option information. If the user does not have update access, an error will be returned indicating that the user does not have update permission, and no change will occur.

Syntax

```
const long NNRMgrResequenceOption (
                             NNRMgr *pMgr,
                             const NNROption *pROption,
                             int oldPosition,
                             int newPosition);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().

Name	Type	Input/ Output	Description
pROption	const NNROption *	Input	Should be populated prior to this function call. The rule name is ignored. See the Option Management API structures description.
oldPosition	int	Input	Old numeric order of the action to be resequenced.
newPosition	int	Input	New numeric order of the action to be resequenced.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls.

Rules Management resequence boundaries are held in the following structure:

```
typedef enum NNRReseqBounds {
    NNRRB_END= -1,
    NNRRB_START=  1
} NNRReseqBounds;
```

Return Value

Returns 1 if the option was resequenced successfully; zero (0) if an error occurred.

If either oldPosition or newPosition are negative and not equal to NNRRB_END, an error condition is returned and errVal is set to RERR_INVALID_OPTION_PARAM.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

Example

```
DbmsSession *session;
NNRMgr *pmgr;
InitNNRMgrSession(pmgr, session);

struct NNROption                key;
struct NNROptionUpdate  data;
int oldPosition, newPosition;
NNR_CLEAR(&key);
NNR_CLEAR(&data);

cout << "Enter app group name \n>";
cin >> key.AppName;
cout << "Enter message type name \n>";
cin >> key.MsgName;
cout << "Enter subscription name \n>";
cin >> key.SubsName;
cout << "Enter action id \n>";
```

```
cin >> key.ActionId;
cout << "Enter old option sequence \n>";
cin >> oldPosition;
cout << "Enter new option sequence \n>";
cin >> newPosition;

if (NNRMgrResequenceOption(pmgr, &key, oldPosition,
newPosition)) {
        cout    << endl
                << "\tOption Name: " << key.OptionName <<
"Resequenced."
                << endl << endl;
        CommitXact(session);
} else {
        DisplayError(pmgr);
        RollbackXact(session);
}

CloseNNRMgr(pmgr, session);
return;
```

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrAddOption(),
NNRMgrGetFirstOption(), NNRMgrGetNextOption(),
NNRMgrUpdateOption()

NNRMgrGetFirstOption

Overview

NNRMgrGetFirstOption() provides a way of starting to retrieve information for a list of options associated with an application group, message type, subscription, and action. This API returns the first option in the action in the pROptionData parameter. Prior to retrieving an option, options must be defined, (see NNRMgrAddApp(), NNRMgrAddMsg(), NNRMgrAddRule(), NNRMgrAddSubscription(), and NNRMgrAddOption()).

When retrieving option information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the option information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetFirstOption(
              NNRMgr *pMgr,
              const NNROption * pROption,
              NNROptionReadData * const pROptionData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pROption	const NNROption *	Input	Should be populated prior to this function call. The rule nmae is ignored. See the Option Management API structures description.
pROptionData	`NNROptionRead Data * const`	Output	NNRMgrGetFirstOption() populates this structure.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetFirstOption().

A call to NNR_CLEAR for both pROption and pROptionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the option was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_OPTIONS, no options were found for the application group and message type specified in the pROption structure.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetNextOption()

NNRMgrGetNextOption

Overview

NNRMgrGetNextOption() provides a way of iterating through the options after the first option has been retrieved (see NNRMgrGetFirstOption()).

When retrieving option information, user permission to read the subscription will be checked. If the user is the owner or another user with read or update permissions for the subscription, the user can see the option information. If the user does not have a minimum of read access, an error will be returned indicating that the user does not have read permission.

Syntax

```
const long NNRMgrGetNextOption(
              NNRMgr *pMgr,
              NNROptionReadData * const pROptionData);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().
pROptionData	`NNROptionReadData * const`	Output	NNRMgrGetNextOption() populates this structure. See the Option Management API structures description.

Remarks

NNRMgrInit() should be called prior to calling NNRMgrGetNextOption().

A call to NNR_CLEAR for both pROption and pROptionData should be made prior to populating the structures or calling this API.

Return Value

Returns 1 if the option was read successfully; zero (0) if an error occurred.

Use NNRMgrGetErrorNo() to retrieve the number for the error that occurred, or use NNRMgrGetError() to retrieve the error message.

If the error number returned is RERR_NO_MORE_OPTIONS, the end of the options list has been reached.

Example

See Sample Program 2: Rules Management API.

See Also

NNRMgrInit(), NNR_CLEAR, NNRMgrGetFirstOption()

Rules Management Error Handling

NNRGetErrorNo

Overview

NNRGetErrorNo() retrieves the error number from previous Rules Management calls.

Syntax

```
const int NNRGetErrorNo(NNRMgr *pRMgr);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().

Return Value

Returns the error number for an error occurring during any of the prior Rules Management calls; returns zero (0) if no Rules Management functions were called prior to this call or NNR_NO_ERR if no error exists. Use NNRGetErrorMessage() to get the associated error message.

Example

See Sample Program 2: Rules Management API.

See Also

NNRGetError()

NNRGetErrorMessage

Overview

NNRGetErrorMessage() retrieves the error message from previous rules management calls.

Syntax

```
const char * NNRGetErrorMessage(NNRMgr *pRMgr);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current Rules Management object. See NNRMgrInit().

Return Value

Returns the error message for an error occurring during any of the previous Rules Management calls.

Example

See Sample Program 2: Rules Management API.

See Also

NNRGetErrorNo()

Chapter 5

Rules Error Messages

This list of errors is subject to change and consists of the following for this release.

Note

Error numbers -10000 to -10099 are Rules Engine Daemon specific and are not included in this list. For more information, see the ***MQSeries Integrator System Management Guide***.

Data Processing Related Errors

Code	Error Name	Error Message	Explanation	Response
-1000		Unknown error code or no error	No matching error code.	
-1001	NO_APPLICATION	Rules configuration missing Application Group	The application group passed into eval() does not exist for the Rules Engine Daemon. This means that the message on the queue did not have a valid OPT_APP_GRP option.	Check the Application Group set in the eval() call OR check the OPT_APP_GRP option for the message in the input queue.
-1002	NO_MESSAGE	Rules configuration missing Message Type	The application group– message type pair passed into eval() does not exist - for the Rules Engine Daemon, this means that the message on the queue did not have a valid OPT_MSG_TYPE option.	Check the Application Group and Message Type set in the eval() call OR check the OPT_APP_GRP and OPT_MSG_TYPE options for the message in the input queue.
-1003	NO_OPERATIONS	Rules not configured and/or Operations missing for message	Rule data in the database was incorrect.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-1004	NO_ARGUMENTS	Rules configuration missing Arguments for message	Rule missing active arguments in the database.	Run Consistency Checker to check data.
-1005	NO_RULES	Rules configuration missing Rules	No active rules defined for the application group–message type pair.	Review the data in the database.
-1006	NO_ SUBSCRIPTIONS	Rules configuration missing Subscriptions	No active subscriptions for the rules in the application group–message type pair.	Review the data in the database.
-1007	NO_SUBSCRIPTION _ACTIONS	Rules configuration missing Subscription Actions	At least one subscription does not have any actions.	Make sure all rules have subscription actions
-1008	NO_BOOLEAN_OPS	Rules configuration missing Boolean Operators	All rules have just a single argument	Call Tech Support
-1009	GET_APP_MSG_SQL _ERROR	Major Database Error Retrieving Application Group / Message Type	Major Database Error	Check database is up and schema is okay
-1010	GET_ARG_SQL_ERR OR	Major Database Error Retrieving Arguments	Major Database Error	Check database is up and schema is okay
-1011	GET_BOOLEAN_OP _SQL_ERROR	Major Database Error Retrieving Boolean Operators	Major Database Error	Check database is up and schema is okay
-1012	GET_OPERN_SQL_ ERROR	Major Database Error Retrieving Operations	Major Database Error	Check database is up and schema is okay
-1013	GET_RULE_SQL_ ERROR	Major Database Error Retrieving Rules	Major Database Error	Check database is up and schema is okay
-1014	GET_SUBACT_SQL_ ERROR	Major Database Error Retrieving Subscription Actions	Major Database Error	Check database is up and schema is okay
-1015	GET_SUBS_SQL_ ERROR	Major Database Error Retrieving Subscriptions	Major Database Error	Check database is up and schema is okay

Client Code Errors

Code	Error Name	Error Message	Explanation	Response
-2000	RULE_MIN_ ERROR	Unknown error code or no error	No error.	
-2001	DBMS_SESSION_ ERROR	Null or dead dbms connection provided to Rules Engine	The Session pointer was invalid.	Check your DBMS and run OpenDbmsSession () again.
-2002	EMPTY_INPUT_ MESSAGE_TYPE	Null or missing message type provided to Rules Engine	No message type name set in eval().	Send in a valid message type.
-2003	ERROR_LOAD_ ARGUMENTS_ ADDARG	Error adding an argument to Rules Engine	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2004	ERROR_LOAD_ ARGUMENTS_CC	Wrong number of argument columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2005	ERROR_LOAD_ ARGUMENTS_ NOCOL	Unexpected argument column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2006	ERROR_LOAD_ ARGUMENTS_ NULL	Null argument column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2007	ERROR_LOAD_ OPERATIONS_ ADDOP	Error adding an operation to Rules Engine	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2008	ERROR_LOAD_ OPERATIONS_CC	Wrong number of operation columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2009	ERROR_LOAD_ OPERATIONS_ NOCOL	Unexpected operation column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2010	ERROR_LOAD_ OPERATIONS_NU LL	Null operation column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2011	ERROR_LOAD_ RULES_ADD_ RULE	Error adding a rule to Rules Engine	A rule in the database has an argument count of zero (0) which is invalid. Rules must have at least one active argument.	Run the consistency checker to find the rule and fix the problem.

Code	Error Name	Error Message	Explanation	Response
-2012	ERROR_LOAD_ RULES_CC	Wrong number of rule columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2013	ERROR_LOAD_ RULES_NOCOL	Unexpected rule column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2014	ERROR_LOAD_ RULES_NULL	Null rule column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2015	ERROR_LOAD_ SUBS_ADD_SUB	Error adding a subscription to Rules Engine	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2016	ERROR_LOAD_ SUBS_CC	Wrong number of subscription columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2017	ERROR_LOAD_ SUBS_NOCOL	Unexpected subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2018	ERROR_LOAD_ SUBS_NULL	Null subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2019	ERROR_LOAD_ SUBSLIST_ADD_ SUBSL	Error adding a rule subscription to Rules Engine	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2020	ERROR_LOAD_ SUBSLIST_CC	Wrong number of rule subscription columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2021	ERROR_LOAD_ SUBSLIST_ NOCOL	Unexpected rule subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2022	ERROR_LOAD_ SUBSLIST_NULL	Null rule subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2023	ERROR_ NEGATIVE_OP_ COUNT	INTERNAL ERROR - failed to resize operations	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2024	ERROR_NEGATIV E_RULE_COUNT	INTERNAL ERROR - failed to resize rules	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2025	FORMATTER_ PARSE_FAILED	Formatter failed to parse input message	The message type may not match the format of the input message.	Check both the Input Format Name (MsgType) and message.
-2026	IE_TOO_MANY_ OPERATIONS	INTERNAL ERROR - incorrect operation count	(Should never see) Memory may be low.	Shut down Rules Engine and restart.

Code	Error Name	Error Message	Explanation	Response
-2027	INVALID_ ARGUMENT_ OPERATION	Invalid Argument loaded - operation id too high	Data in the database is incorrect.	Run Consistency Checker to check data.
-2028	INVALID_INPUT_ MESSAGE_LEN	Input message had an invalid length	Call to eval() had an invalid msglen parameter.	Check the parameters sent to eval().
-2029	INVALID_RULE_ ARG_COUNT	Rule argument count is invalid - check table	Data in the database is incorrect.	Run Consistency Checker to check data.
-2030	NULL_ FORMATTER_ INSTANCE	Formatter instance is null	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2031	INPUT_MESSAGE _NULL	Null input message	The message sent through eval() is empty.	Check the call to eval() or the message in the queue when running the Rules Engine Daemon.
-2032	OPERATION_ EVALUATION_ FAILED	Internal Error - Evaluation failure #1	Problem evaluating part of a rule – operator may be invalid.	Run Consistency Checker to check data.
-2033	OP_ADD_ARG_ FAILED (operation add argument failed	Internal Error - Load failure #1	Problem loading arguments.	Run Consistency Checker to check data.
-2034	OP_CONS_ FAILED (Operator Constructor detected	Internal Error - Load failure #2	Problem loading operator.	Run Consistency Checker to check data.
-2035	RULE_ OPERATION_ MISSING (rule operation array error)	Internal Error - Evaluation failure #2	Problem evaluating part of a rule – operator may be invalid.	Run Consistency Checker to check data.
-2036	UNSUPPORTED_ DBMS_ INTERFACE	Database type not supported	Invalid DbmsType in the Session variable used to create Rules Engine.	Check call to OpenDbmsSession ().
-2037	INVALID_RULE_ SUBSCRIPTIO	Internal Error - Load failure #3	Problem loading subscriptions.	Run Consistency Checker to check data.
-2038	FAILED_ADD_ SUBSCRIPTION	Internal Error - Load failure #4	Problem loading subscriptions.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2039	EMPTY_ APPLICATION_ GROUP_NAME	Empty Input Value for Application Group Name	No application group name passed into eval().	Check call to eval().
-2040	EMPTY_ MESSAGE_NAME	Empty Input Value for Message Name	No message type name passed into eval().	Check call to eval().
-2041	IE_NULL_ MESSAGE_ GROUP	Internal Error - Lookup failure #1	Problem loading message type.	Run Consistency Checker to check data.
-2042	IE_NULL_ APPLICATION_ GROUP	Internal Error - Lookup failure #2	Problem loading application group.	Run Consistency Checker to check data.
-2043	IE_NULL_ ENGINE_ INSTANCE	Internal Error - Null Engine Instance	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2044	ERROR_SETTING _HITLIST	Error setting HitList	gethitrule() had problems retrieving hit rules.	Run Consistency Checker to check data.
-2045	ERROR_SETTING _HITLIST	Error setting NoHitList	getnohitrule() had problems retrieving no hit rules.	Run Consistency Checker to check data.
-2046	IE_NO_ERROR_ HANDLER	Internal Error - No error handler	(Should never see) Memory may be low.	Shut down Rules Engine and restart.
-2047	IE_CANNOT_SET _TSD	Internal Error - Error Setting Thread Specific Data	Problem with threading - maybe too many threads.	Shut down process immediately, check system, and restart.
-2048	ERROR_LOAD_ BOOLEAN_ OPERATORS	Internal Error - Error Loading Boolean Operators	Problem loading Boolean operators.	Run Consistency Checker to check data.

Rules Management Data Errors

For all these errors check that the DBMS is still running properly.

Code	Error Name	Error Message	Explanation	Response
-2500	NNR_NO_ERR	No rules management error	No error.	
-2501	RERROR_DB	DB error	Not in use.	(Should never see)
-2502	RERR_COUNTER _ADD	DB error Counter Insert	Data may be incorrect to add new Application Group.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2503	RERR_COUNTER _UPDATE	DB error Counter Update	Data may be incorrect to add new Application Group.	Run Consistency Checker to check data.
-2504	RERR_COUNTER _INSTANCE_ ADD	DB error Counter Instance Insert	Data may be incorrect to add new Rule, Subscription, etc.	Run Consistency Checker to check data.
-2505	RERR_COUNTER _INSTANCE_ UPDATE	DB error Counter Instance Update	Data may be incorrect to add new Rule, Subscription, etc.	Run Consistency Checker to check data.
-2506	RERR_APP_ GROUP_ADD	DB error Application Group Insert	Problem inserting Application Group. May be duplicate.	Run Consistency Checker to check data.
-2507	RERR_MSG_TYPE _ADD_FORMAT	DB error message type insert (format)	Problem inserting Message Type. May not be valid format.	Run Consistency Checker to check data.
-2508	RERR_R_ MESSAGES_ADD	DB error message type insert	Problem inserting Message Type. May be duplicate.	Run Consistency Checker to check data.
-2509	RERR_RULE_ ADD	DB error rule insert	Problem inserting Rule. May be duplicate.	Run Consistency Checker to check data.
-2510	RERR_RULE_ UPDATE	DB error rule update	Problem updating Rule. Rule may not exist.	Run Consistency Checker to check data.
-2511	RERR_ OPERATION_ ADD	DB error argument op insert	Problem inserting operator for rule.	Run Consistency Checker to check data.
-2512	RERR_ARG_ADD	DB error argument insert (Arg)	Problem inserting argument for rule.	Run Consistency Checker to check data.
-2513	RERR_ OPERATION_ UPDATE	DB error argument op update	Problem updating argument for rule.	Run Consistency Checker to check data.
-2514	RERR_R_ SUBSCRIPTION_ LIST_ADD	DB error subscription list insert	Problem inserting subscription. May be duplicate.	Run Consistency Checker to check data.
-2515	RERR_R_ SUBSCRIPTION_ MASTER_ADD	DB error subscription master insert	Problem inserting subscription. May be duplicate.	Run Consistency Checker to check data.
-2516	RERR_R_ SUBSCRIPTION_ ACTION_ADD	DB error action insert	Problem inserting action.	Run Consistency Checker to check data.
-2517	RERR_ APPLICATION_ GROUP_READ	DB error application group read	Problem retrieving application group. May have wrong name.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2518	RERR_MESSAGE_ TYPE_READ	DB error message type read	Problem retrieving message type. May have wrong parameters.	Run Consistency Checker to check data.
-2519	RERR_RULE_ READ	DB error rule read	Problem retrieving rule. May have wrong parameters.	Run Consistency Checker to check data.
-2520	RERR_ SUBSCRIPTION_ LIST_READ	DB error subscription list read	Problem retrieving subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2521	RERR_ SUBSCRIPTION_ MASTER_READ	DB error subscription master read	Problem retrieving subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2522	RERR_ SUBSCRIPTION_ ACTION_READ	DB error subscription action read	Problem retrieving subscription action. May have wrong parameters.	Run Consistency Checker to check data.
-2523	RERR_MESSAGE_ TYPE_READ_ MESSAGE_ID	DB error message type read (message id)	Problem retrieving message type - format. May have wrong parameters.	Run Consistency Checker to check data.
-2524	RERR_ OPERATOR_ READ	DB error operator read	Problem retrieving operator. May have wrong parameters.	Run Consistency Checker to check data.
-2525	RERR_ OPERATOR_ TYPE_READ	DB error operator type read	Problem retrieving operator type. May have invalid operator.	Run Consistency Checker to check data.
-2526	RERR_ARG_ READ	DB error argument read	Problem retrieving rule action. May have wrong parameters.	Run Consistency Checker to check data.
-2527	RERR_COUNTER _READ	DB error counter read	Problem retrieving new application id. May have wrong parameters.	Run Consistency Checker to check data.
-2528	RERR_COUNTER _INSTANCE_ READ	DB error counter instance read	Problem retrieving new ids for rule, subscription, etc. May have wrong parameters.	Run Consistency Checker to check data.
-2529	RERR_ OPERATION_ READ	DB error operation read	Problem retrieving argument info. May have wrong parameters.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2530	RERR_STALE_ OPERATION_ EXISTS	DB error unreferenced operations	Arguments still exist that are not used in a rule.	Run Consistency Checker to check data.
-2531	RERR_ ARGUMENT_ UPDATE	DB error argument update	Could not update argument.	Run Consistency Checker to check data.
-2532	RERR_ SUBSCRIPTION_ COMBINED_ READ	DB error subscription multi-read	Problem retrieving subscription info. May have wrong parameters.	Run Consistency Checker to check data.
-2533	RERR_NO_ OPTIONS_READ	DB error options not found	No options found for subscription action.	Run Consistency Checker to check data.
-2534	RERR_DELETE_ OPTION_FAILED	DB error option delete	Could not delete option.	Run Consistency Checker to check data.
-2535	RERR_ RESEQUENCE_ ACTION_FAILED	DB error action resequence	Could not resequence actions. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2536	RERR_ RESEQUENCE_ OPTION_FAILED	DB error option resequence	Could not resequence options. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2537	RERR_DELETE_ ALL_ ARGUMENTS_ FAILED	DB error delete all arguments failed	Could not delete all arguments for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2538	RERR_DELETE_ ALL_LIST_SUBS_ FAILED	DB error delete all list subscriptions failed	Could not delete all subscriptions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2539	RERR_DELETE_ ALL_MASTER_ SUBS_FAILED	DB error delete all subscription masters failed	Could not delete all subscriptions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2540	RERR_DELETE_ ALL_ACTIONS_ FAILED	DB error delete all actions failed	Could not delete all actions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2541	RERR_ DECREMENT_ OPERATION_ FAILED	DB error operation decrement	Could not reduce the number of arguments using a specific operator.	Run Consistency Checker to check data.
-2542	RERR_DELETE_ RULE_FAILED	DB error delete rule	Could not delete rule. May have wrong parameters.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2543	RERR_DELETE_ ARGUMENTS_ FAILED	DB error delete arguments	Could not delete argument. May have wrong parameters.	Run Consistency Checker to check data.
-2544	RERR_DELETE_ OPERATION_ FAILED	DB error delete operation	Could not delete argument information for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2545	RERR_DELETE_ ACTIONS_FAILE D	DB error delete actions	Could not delete action. May have wrong parameters.	Run Consistency Checker to check data.
-2546	RERR_DELETE_ SUBS_FAILED	DB error delete subscriptions	Could not delete subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2547	RERR_RESEQ_ OPTION_RANGE _FAILED	DB error resequence multiple options	Could not resequence options. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2548	RERR_INSERT_ OPTION_FAILED	DB error option insert	Could not insert option. May have wrong parameters.	Run Consistency Checker to check data.
-2549	RERR_GET_MAX _ACTION_ FAILED	DB error get max action	Could not retrieve the maximum number of actions. May not have any actions.	Run Consistency Checker to check data.
-2550	RERR_GET_MAX _OPTION_ FAILED	DB error get max option	Could not retrieve the maximum number of options. May not have any options.	Run Consistency Checker to check data.
-2551	RERR_MOVE_ ACTION_FAILED	DB error move action	Could not resequence action. May have invalid sequence parameter.	Run Consistency Checker to check data.
-2552	RERR_MOVE_ OPTION_FAILED	DB error move option	Could not resequence option. May have invalid sequence parameter.	Run Consistency Checker to check data.
-2553	RERR_RESEQ_ ACTION_RANGE _FAILED	DB error resequence multiple actions	Could not resequence actions. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2554	RERR_UPDATE_ ACTION_FAILED	DB error update action	Could not update action. May have wrong parameters.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2555	RERR_UPDATE_ OPTION_FAILED	DB error update option	Could not update option. May have wrong parameters.	Run Consistency Checker to check data.
-2556	RERR_UPDATE_ SUBSCRIPTION_ FAILED	DB error update subscription	Could not update subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2557	RERR_OPTION_ READ_FAILED	DB error option read	Could not retrieve option. May have wrong parameters	Run Consistency Checker to check data.
-2558	RERR_GET_MAX _ARG_FAILED	DB error get max argument	Could not retrieve the maximum number of arguments. May not have any arguments.	Run Consistency Checker to check data.
-2559	RERR_APP_ GROUP_UPDATE	DB error application group update	Could not update application name. May have wrong old name.	Run Consistency Checker to check data.
-2560	RERR_GET_ VERSION_FAILE D	DB error get version failed	Could not retrieve version information for import / export.	Run Consistency Checker to check data.
-2561	RERR_CANNOT_ UPDATE_FIELD	DB error update field name failed	Could not update the old name to the new field name.	Run Consistency Checker to check data.
-2562	RERR_GET_MAX _BOOLEAN_ OPER_FAILED	DB error get max boolean operator	Could not retrieve the maximum number of Boolean operators. May have wrong parameters.	Run Consistency Checker to check data.
-2563	RERR_BOOLEAN _OP_ADD	DB error boolean operator add failed	Could not insert Boolean operato. May have wrong parameters.	Run Consistency Checker to check data.
-2564	RERR_BOOLEAN _OP_INCR	DB error boolean operator update failed	Could not update Boolean operator. May have wrong parameters.	Run Consistency Checker to check data.

General Rules Management Errors

Code	Error Name	Error Message	Explanation	Response
-2600	RERR_INVALID_ APP_PARAM	Invalid application group parameters	Invalid application group name.	Check passed-in application group name.
-2601	RERR_APP_GROUP _NAME_ALREADY_ EXISTS	Error application group already exists	Cannot add application with duplicate name.	Check passed-in application group name.
-2602	RERR_APP_GROUP _NAME_DOES_NOT _EXIST	Error application group does not exist	Invalid application group name.	Check passed-in application group name.
-2603	RERR_INVALID_ MSG_PARAM	Invalid message type parameters	Invalid application group / message type pair.	Check passed-in application group / message type name.
-2604	RERR_MSG_TYPE_ NAME_ALREADY_ EXISTS	Error message type already exists	Application group already has the message type.	Check passed-in application group / message type name.
-2605	RERR_MSG_TYPE_ NAME_DOES_NOT_ EXIST	Error message type does not exist	Invalid application group / message type pair.	Check passed-in application group / message type name.
-2606	RERR_FORMAT_ NAME_DOES_NOT_ EXIST	Error format name does not exist	Message type name must match an input format name.	Check passed-in a message type name against format names.
-2607	RERR_INVALID_ RULE_PARAM	Invalid rule parameters	Invalid application group / message type / rule name.	Check passed-in parameters.
-2608	RERR_RULE_NAME _ALREADY_EXISTS	Error rule name already exists	Application group / message type pairs can not have duplicate rule names.	Check passed-in parameters.
-2609	RERR_RULE_NAME _DOES_NOT_EXIST	Error rule name does not exist	Invalid application group / message type / rule name.	Check passed-in parameters.
-2610	RERR_INVALID_ OPERATOR_ PARAM	Invalid operator parameters	Invalid operator ID.	Check passed-in parameter.
-2611	RERR_INVALID_ ARG_PARAM	Invalid argument parameters	Invalid parameters to create / update / retrieve argument.	Check passed-in parameters.

Code	Error Name	Error Message	Explanation	Response
-2612	RERR_INVALID_ SUBS_PARAM	Invalid subscription parameters	Invalid parameters to create / update / retrieve subscription.	Check passed-in parameters.
-2613	RERR_SUBS_NAME _ALREADY_EXISTS	Error subscription name already exists	Subscription names cannot be duplicated within a rule.	Check passed-in parameters.
-2614	RERR_SUBS_NAME _DOES_NOT_EXIST	Error subscription name does not exist	Application group / message type / rule name / subscription name not found.	Check passed-in parameters.
-2615	RERR_INVALID_ ACTION_PARAM	Invalid action parameters	Invalid parameters to create / update / retrieve action.	Check passed-in parameters.
-2616	RERR_ACTION_SEQ _DOES_NOT_EXIST	Error action does not exist	Application group / message type / rule name / subscription name / action name not found.	Check passed-in parameters.
-2617	RERR_INVALID_ OPTION_PARAM	Invalid option parameters	Invalid parameters to create / update / retrieve action	Check passed-in parameters.
-2618	RERR_ CONVERSION_ ERROR	Error during conversion	Conversion of static argument value failed.	Check passed-in parameters. Run Consistency Checker.
-2619	RERR_NO_MORE_ ACTIONS	No more actions	Not really error unless returned from GetFirst...	Subscription must have at least one action.
-2620	RERR_NO_MORE_ OPERATORS	No more operators	Not really error.	
-2621	RERR_NO_MORE_ ARGUMENTS	No more arguments	Not really error unless returned from GetFirst...	Rule must have at least one argument.
-2622	RERR_INVALID_ RULES_PARAM	Invalid rules management object passed in	Must call NNRMgrInit() before calling any other functions.	Call NNRMgrInit() prior to calling any other functions.
-2623	RERR_FEATURE_ NOT_ IMPLEMENTED	Feature not implemented	Feature is not implemented at this time.	
-2624	RERR_ARGUMENT_ DOES_NOT_EXIST	Argument does not exist	Invalid parameters to update / retrieve argument.	Check passed-in parameters: AppGrp / MsgType / RuleName / ArgSeq / Fields / Operator.

Code	Error Name	Error Message	Explanation	Response
-2625	RERR_OPERATION_ DOES_NOT_EXIST	Operation does not exist	Invalid parameters to update / retrieve argument information.	Check passed-in parameters: AppGrp / MsgType / RuleName / ArgSeq / Fields / Operator.
-2626	RERR_UNKNOWN_ OPERATOR_TYPE	Unknown operator type	Operator may be invalid.	Check passed-in parameters.
-2627	RERR_NO_MORE_ SUBSCRIPTIONS	No more subscriptions	Not really error unless returned from GetFirst...	Rule must have at least one subscription.
-2628	RERR_NO_MORE_ RULES	No more rules	Not really error.	
-2629	RERR_ACTION_ DOES_NOT_EXIST	Action does not exist	Invalid parameters to update / retrieve action.	Check passed-in parameters: AppGrp / MsgType / RuleName / SubName / ActSeq.
-2630	RERR_OPTION_DO ES_NOT_EXIST	Option does not exist	Invalid parameters to update / retrieve option.	Check passed-in parameters: AppGrp / MsgType / RuleName / SubName / ActSeq / OptSeq.
-2631	RERR_APP_ID_ CORRUPTED	App id corrupted	Data for Application Group may be incorrect.	Run Consistency Checker to check data.
-2632	RERR_MSG_ID_ CORRUPTED	Msg id corrupted	Data for Message Type may be incorrect.	Run Consistency Checker to check data.
-2633	RERR_NO_MORE_ OPTIONS	No more options	Not really error unless returned from GetFirst...	Action must currently have at least one option.
-2634	RERR_EXPORT_APP _FAILURE	Export app name failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2635	RERR_EXPORT_ MSG_FAILURE	Export message name failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2636	RERR_EXPORT_ RULE_FAILURE	Export rule failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2637	RERR_EXPORT_ ARG_FAILURE	Export argument failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-2638	RERR_EXPORT_SUB _FAILURE	Export subscription failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2639	RERR_EXPORT_ACT _FAILURE	Export action failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2640	RERR_EXPORT_OPT _FAILURE	Export option failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2641	RERR_NO_MORE_ MESSAGES	No more messages	Not really error.	
-2642	RERR_NO_MORE_ APPLICATIONS	No more applications	Not really error.	
-2643	RERR_IMPORT_ FILE_READ	Error reading import file	Import failed to read from file.	Check file. Recreate file by exporting again.
-2644	RERR_IMPORT_APP	Error importing application	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data. Try importing with overwrite flag.
-2645	RERR_INVALID_IE_ TYPE	Invalid import / export type	Can only import / export Rules components.	Should never see this error.
-2646	RERR_IMPORT_ MSG	Error importing message type	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data. Try importing with overwrite flag.
-2647	RERR_IMPORT_ RULE	Error importing rule	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data. Try importing with overwrite flag
-2648	RERR_MEMORY_ ALLOCATION_ FAILURE	Memory allocation failure	Could not allocate memory.	Shut down excess items. Restart import / export.
-2649	RERR_IMPORT_ ARGUMENT	Error importing argument	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2650	RERR_IMPORT_ SUBSCRIPTION	Error importing subscription	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data. Try importing with overwrite flag

Code	Error Name	Error Message	Explanation	Response
-2651	RERR_IMPORT_ ACTION	Error importing action	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2652	RERR_IMPORT_ OPTION	Error importing option	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2653	RERR_ UNSUPPORTED_ VERSION	Unsupported version	Can only export release 1.0. Can only import release 1.0.	Check release of MQSeries Integrator Rules.
-2654	RERR_DECODE_ FAILURE	Decoding failure	Could not decode line in file.	Export File may be corrupt. Recreate file by exporting again.
-2655	RERR_NONOWNER _CANNOT_ADD_ PERMISSION	Cannot add permission if not owner	Rule old owner may not be a valid user of the current database.	Check database users.
-2656	RERR_NO_ PERMISSION_TO_ READ	No permission to read	Cannot read permission. Read permission not granted.	Assign permissions to rules.
-2657	RERR_NO_ PERMISSION_TO_ UPDATE	No permission to update	Current user does not have update permission for the rule.	Have rule owner change update permissions for himself and / or PUBLIC.
-2658	RERR_PERMISSION _LIST_READ_ FAILURE	Permission list read failure	Could not read permission list.	Run Consistency Checker to check data.
-2659	RERR_NO_MORE_ PERMISSIONS	No more permissions	Not really an error.	
-2660	RERR_EXPORT_ VERSION_FAILURE	Error exporting version	Could not retrieve version for export. Can only export from release 1.0 and higher.	Check MQSeries Integrator install.
-2661	RERR_EXPORT_ PERMISSIONS_ FAILURE	Error exporting permissions	Could not export rule permissions.	Run Consistency Checker to check data.
-2662	RERR_INVALID_ FIELD_NAME_ PARAM	Invalid field name parameter	The field name provided is invalid.	Check parameters to function call.

Code	Error Name	Error Message	Explanation	Response
-2666	RERR_INVALID_ DATE_TIME_ FORMAT_IN_ARG	Invalid date/time format in argument	Bad format of static date/time value	Check input parameter. Verify that the Time portion of a Date value or the Date portion of a Time value is zero padded.
-2667	RERR_NON_ NUMERIC_DATE_ TIME_IN_ARG	Invalid non-numeric date/time value in argument	Bad format of static date/time value	Check input parameter
-2668	RERR_INVALID_ YEAR_IN_ARG	Invalid year in argument	Bad format of static date/time value	Check input parameter
-2669	RERR_INVALID_ MONTH_IN_ARG	Invalid month in argument	Bad format of static date/time value	Check input parameter
-2670	RERR_INVALID_ DAY_IN_ARG	Invalid day in argument	Bad format of static date/time value	Check input parameter
-2671	RERR_INVALID_ HOUR_IN_ARG	Invalid hour in argument	Bad format of static date/time value	Check input parameter
-2672	RERR_INVALID_ MINUTE_IN_ARG	Invalid minute in argument	Bad format of static date/time value	Check input parameter
-2673	RERR_INVALID_ SECOND_IN_ARG	Invalid second in argument	Bad format of static date/time value	Check input parameter
-2674	RERR_ UNBALANCED_ QUOTES	Unbalanced quotes in expression after	Invalid Boolean expression - quotes must be balanced	Check input expression parameter
-2675	RERR_INVALID_ RULES_OPERATO	Invalid Rules	Operator in expression in Invalid Rules operator	Check the Operator list for spelling / case
-2676	RERR_MISSING_ RULES_OPERATOR	Expression missing Rules Operator	Rules expression must have a Rules Operator	Check input expression parameter
-2677	RERR_NEED_ SECOND_FIELD_OR _VALUE	Rules Operator missing comparison value or field name in expression	All Rules operators need a second argument except those checking for existence	Check input expression parameter
-2678	RERR_ UNBALANCED_ PARENS	Unbalanced parentheses in expression	Parentheses must be balanced in Rules expression	Check input expression parameter
-2679	RERR_EXPECTED_ TERMINAL	Expected terminal in expression	Expression ended incorrectly	Check input parameter
-2680	RERR_ARG_MUST_ BE_ACTIVE	Arguments must be active for NEONet 4.0+	Arguments can no longer be Inactive	Change input expression parameter

Code	Error Name	Error Message	Explanation	Response
-2681	RERR_USE_ UPDATE_EXPR	Must Use NNRMgrUpdateExpression to perform desired update	Cannot use NNRMgrAddArgument unless all arguments are just ANDed together	Use NNRMgrUpdateExpression
-2682	RERR_TRAILING_ CHARS	Trailing characters found in expression	Extra characters in the expression	Make sure you are using '&' and '|' for Boolean operators
-2683	RERR_MISSING_ OPERAND	Missing operand in boolean expression before / after	2 Operands are needed around a Boolean operator	Check input expression parameter
-2684	RERR_NONOWNER _CANNOT_DELETE	Cannot delete item if not owner.	User not the owner of the sub / rule Cannot delete	Delete as owner.
-2685	RERR_ SUBSCRIPTION_IS_ USED	Subscription is used by a rule - cannot delete	Subscription is used by a rule and cannot be deleted	Remove subscription from all associated rules
-2686	RERR_INVALID_ COMPONENT_ TYPE	Invalid component type as parameter	Invalid component type parameter	Check component type - input parameter
-2687	RERR_INVALID_ COMPONENT_ PARAM	Invalid or missing parameter	May have invalid parameter	Check passed in parameters (i.e., NULL values)
-2688	RERR_INVALID_ CHANGE_OWNER_ PARAM	Invalid or missing change owner parameter	may have invalid parameter	Check passed in parameter
-2689	RERR_INVALID_ COMPONENT_ OWNER_PARAM	Invalid or missing component owner parameter	May have invalid parameter	Check passed in parameter for NULL value
-2690	RERR_ SUBSCRIPTION_ LIST_READ_ FAILURE	Subscription list read failure	Failure reading subscription list	Run Consistency Checker and check data
-2691	RERR_RULE_LIST_ READ_FAILURE	Rule list read failure	Failure reading rule list	Run Consistency Checker and check data
-2692	RERR_IMPORT_ PERM	Error importing permission	Error importing permission	Check file. Run Consistency Checker to check data
-2693	RERR_USE_ EXISTENCE_OPS	Cannot compare against empty strings - use existence operator	Cannot do a comparison against an empty string	To compare against an empty field, use the EXIST or NOT_EXIST operator

Code	Error Name	Error Message	Explanation	Response
-2694	RERR_OPT_PUT_ FMT_INVALID	Invalid option value for putqueue MQS_FORMAT option	Option can be only 8 characters long	Change the parameters sent into NNRMgrAddOption or NNRMgrUpdateOpti on
-2695	RERR_OPT_PUT_ PROP_INVALID	Invalid option value for putqueue MQS_PROPAGAT E option	Must be PROPAGATE OR NO_PROPAGATE	Change the parameters sent into NNRMgrAddOption or NNRMgrUpdateOpti on
-2696	RERR_OPT_PUT_ PER_INVALID	Invalid option value for putqueue MQS_PERSIST option	Must be PERSIST OR NO_PERSIST	Change the parameters sent into NNRMgrAddOption or NNRMgrUpdateOpti on
-2697	REERR_OPT_PUT_ EXP_INVALID	Invalid option value for putqueue MQS_EXPIRY option	Must be PROPAGATE OR NO_PROPAGATE	Change the parameters sent into NNRMgrAddOption or NNRMgrUpdateOpti on
-2698	RERR_OPT_FMT_ FMT_INVALID	Invalid option value for reformat option	INPUT_FORMAT must be a valid input format name and TARGET_FORMAT must be a valid output format name	Change the parameters sent into NNRMgrAddOption or NNRMgrUpdateOpti on or add the needed formats

Permission Data Errors

Component refers to any item with its own permissions i.e., Rules and Subscription for MQSeries Integrator 1.0.

Code	Error Name	Error Message	Explanation	Response
-5500	NN_NO_DB_ERR	No NEONet database error	No error.	
-5501	NN_ID_INSERT_ FAILURE	Get next id insert error	Error getting new ids for user / permission.	Run Consistency Checker to check data.
-5502	NN_ID_UPDATE_ FAILURE	Get next id update error	Error getting new ids for user / permission.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-5503	NN_NODE_DOES_ NOT_EXIST	Node does not exist	Must run on valid 1.0 MQSeries Integrator database with Node data saved.	Check MQSeries Integrator install.
-5504	NN_HIERARCHY_ DOES_NOT_EXIST	Hierarchy does not exist	Must run on valid 1.0 MQSeries Integrator database with Hierarchy data saved.	Check MQSeries Integrator install. Run Consistency Checker to check data.
-5505	NN_COMPONENT_ ADD_FAILURE	Component add failure	Could not add rule component to permission system - may be duplicate.	Run Consistency Checker to check data.
-5506	NN_COMPONENT_ LOAD_FAILURE	Component load failure	Could not retrieve rule component information from permission system. May not exist.	Run Consistency Checker to check data.
-5507	NN_DELETE_ COMPONENT_ FAILURE	Delete component failure	Could not delete rule component information from permission system. May not exist.	Run Consistency Checker to check data.
-5508	NN_UNABLE_TO_ DETERMINE_USER	Unable to determine user	Permission user not a valid database user.	Run Consistency Checker to check data.
-5509	NN_UNABLE_TO_ FIND_USER	Unable to find user in database	Permission user not a valid database user.	Run Consistency Checker to check data.
-5510	NN_UNABLE_TO_ FIND_USER_IN_ NEONET	Unable to find user in NEONet	Permission user not a valid permission user in MQSeries Integrator.	Run Consistency Checker to check data.
-5511	NN_UNABLE_TO_ ADD_USER_TO_ NEONET	Unable to add user to NEONet	Cannot add permission user. May not be a valid database user.	Run Consistency Checker to check data.
-5512	NN_UNABLE_TO_ ADD_PERMISSION_ SET	Unable to add permission	Cannot add permission - may be a duplicate.	Run Consistency Checker to check data.
-5513	NN_UNABLE_TO_ FIND_PERMISSION	Unable to find permission	Cannot find permission. May have invalid parameters.	Run Consistency Checker to check data.
-5514	NN_UNABLE_TO_ LOAD_PERMISSION _LIST	Unable to read permission	Cannot retrieve permission. May have invalid parameters.	Run Consistency Checker to check data.

Code	Error Name	Error Message	Explanation	Response
-5515	NN_UNABLE_TO_ UPDATE_ PERMISSION	Unable to update permission	Cannot update permission. May have invalid parameters.	Run Consistency Checker to check data.
-5516	NN_ADD_USER_ NOT_DB_USER	User is not a valid user of the database instance	Permission user not a valid database user.	Run Consistency Checker to check data.
-5517	NN_UNABLE_TO_ CHANGE_ PERMISSION_USER	Unable to change the user for the permissions	The new user may not be valid or caused a duplicate permission.	Run Consistency Checker to check data.
-5518	NN_UNABLE_TO_ DELETE_ PERMISSIONSET	Unable to delete the permission set	Invalid parameters to delete permission set for a user - rule pair.	Run Consistency Checker to check data.
-5519	NN_ NOPERMISSIONS_ FOUND	No permissions were found	Indicates no more permissions to read for rule or subscription	Rule or subscription must have at least two permissions
5520	NN_COMPONENT_ UPDATE_FAILURE	Component update failure	Cannot update permission. May have invalid parameter	Run Consistency Checker to run data

General Permission Errors

Component refers to any item with its own permissions i.e., Rules and Subscription for MQSeries Integrator 1.0.

Code	Error Name	Error Message	Explanation	Response
-5000	NN_NO_ERR	No Errors	No error.	
-5001	NN_GET_NEXT_ID_ INVALID_PARAM	Next id invalid parameters	Invalid parameters to get new user / component id for permission system.	Check passed-in parameters.
-5002	NN_UPDATE_ PERMISSION_ INVALID_PARAM	Update permission invalid parameters	Invalid parameters to update permission.	Check passed-in parameters.
-5003	NN_GET_NODE_ID _INVALID_PARAM	Get node invalid parameters	Invalid parameters to retrieve node information.	Check passed-in parameters.

Code	Error Name	Error Message	Explanation	Response
-5004	NN_HIERARCHY_ LEVEL_INVALID_ PARAM	Get hierarchy level invalid parameters	Invalid parameters to retrieve hierarchy level information.	Check passed-in parameters.
-5005	NN_HIERARCHY_ INVALID_PARAM	Get hierarchy invalid parameters	Invalid parameters to retrieve hierarchy information.	Check passed-in parameters.
-5006	NN_ADD_ COMPONENT_ INVALID_PARAM	Add component invalid parameters	Invalid parameters to add component to permission system.	Check passed-in parameters.
-5007	NN_COMPONENT_ LOAD_INVALID_ PARAM	Load component invalid parameters	Invalid parameters to retrieve component from permission system.	Check passed-in parameters.
-5008	NN_DELETE_ COMPONENT_ INVALID_PARAM	Delete component invalid parameters	Invalid parameters to delete component from permission system.	Check passed-in parameters.
-5009	NN_LOAD_USER_ INVALID_PARAM	Load user invalid parameters	Invalid parameters to retrieve user from permission system.	Check passed-in parameters.
-5010	NN_ADD_USER_IN VALID_PARAM	Add user invalid parameters	Invalid parameters to add user to permission system.	Check passed-in parameters.
-5011	NN_ADD_ PERMISSION_ INVALID_PARAM	Add permission invalid parameters	Invalid parameters to add permission to permission system.	Check passed-in parameters.
-5012	NN_LOAD_ PERMISSION_ INVALID_PARAM	Load permission invalid parameters	Invalid parameters to retrieve permission from permission system.	Check passed-in parameters.
-5013	NN_PERMISSION_ ALREADY_EXISTS	Adding permission that already exists	Duplicate permissions not allowed for user / component/ permission.	Check passed-in parameters.
-5014	NN_CHANGE_ USER_PERM_ INVALID_PARAM	Changing user invalid parameters	Invalid parameters to change the owner for a certain component.	Check passed-in parameters.
-5015	NN_DELETE_ PERMSET_INVALID _PARAM	Deleting permission set invalid parameters	Invalid parameters to delete all permissions for a user / component.	Check passed-in parameters.

Code	Error Name	Error Message	Explanation	Response
-5016	NN_NONOWNER_ CANNOT_ADD_ PERMISSION	Cannot add permission if not owner	User is not the owner of the component. Cannot add / update permission.	Add as owner of component
-5017	NN_NO_ PERMISSION_TO_ READ	No permission to read	Read permission not granted to PUBLIC or User.	Grant read permission for component
-5018	NN_PERMISSION_ LIST_READ_ FAILURE	Permission list read failure	Cannot read permission list	Run Consistency Checker to check data
-5019	NN_NO_MORE_ PERMISSIONS	No more permissions	Indicates no more permissions to read for rule or subscription	Rules and subscriptions must have at least two permissions
-5020	NN_NO_MORE_ ITEMS	No more components.	Not really an error	
-5021	NN_ NOPERMISSION_ TO_UPDATE	No permission to update	Update permission not granted to PUBLIC or User.	Grant update permission for component
-5022	NN_NONOWNER_ CANNOT_DELETE	Cannot delete item if not owner	User is not the owner of the component.Cannot delete item.	Delete as owner of component

Appendix A

Sample Programs

Makefile

To compile the Rules APIs, system-dependent database libraries, system/compiler specific standard C/C++ libraries, and system-dependent thread libraries for multi-threaded applications must exist. In addition, the MQSeries Integrator Formatter (libformat.a), Rules (librules.a), and generic tool set (libntools.a) libraries must exist.

The following is an example of a makefile for using the Rules APIs. The variable LIB_DIR must be set with the path to the directory containing the library files (libformat.a, librules.a, and libntools.a). The variable DBMS_DIR must be set with the path to the directory containing the system-dependent database libraries.

Note

For multithreading, you must also link with the appropriate thread library matching the MQSeries Integrator release. For example, link with the thread library for UI threads, pthread for POSIX threads, and so on.

The following is an example Makefile:

```
include ./Makeinfo

kit_make: $(RULESOBJ.o)
touch kit_make

ruletest: ruletest.o $(LIB_DIR)/libneonet.a
g++ -g -O    testrules.o \
-L$(LIB_DIR)\
-L$(DBMS_DIR)\
-lneonet\
-lrules\
-lformat \
-lsqlobj\
-lntools\
-libqmon\
-lsybdb\
-lm\
-o ruletest
```

The following is an example of the associated output using the preceding Makefile. The LIB_DIR in the Makefile is the directory /export/u/users/sja/aig/lib, and the SYBASE directory is /usr/sybase/lib as shown below:

```
g++ -g -Oruletest.o \
-L/export/u/users/sja/aig/lib\
-L/usr/sybase/lib\
-lneonet\
-lrules\
-lformat\
-lsqlobj\
-libqmon\
-lntools\
-lsybdb\
-lm\
-o ruletest
```

Sample Program 1: Rules API

This program reads a file containing a message. The filename is "testdata.txt." The application group and message type are Bravo and "HL7 Message" respectively. Once the file has been read, this program evaluates the message. After evaluation, the subscriptions and options for rules that evaluated to "true" are retrieved and written to standard out.

```
#include <stdio.h>
#include <stdlib.h>

#if defined(_MS_SQL_NT)
#include "interface.h"
#include <sqlfront.h>
#include <sqldb.h>
#elif defined(sybase)
extern "C" {
#include <sybfront.h>
#include <sybdb.h>
}
#endif

#include <iostream.h>
#include <fstream.h>
#include "dbtypes.h"
#include "ses.h"
#include "sqlapi.h"
#include "rerror.h"
#include "ruleuser.h"
#include "vrule.h"

#include "neobuf.h"

extern int err_handler( DBPROCESS*dbproc, intseverity,
   int dberr, intoserr, char*dberrstr, char*oserrstr);

extern int msg_handler( DBPROCESS*dbproc, DBINTmsgno,
   int msgstate, intseverity, char*msgtext,
   char*foo, char* baz, short unsigned int bar);

int
main(int argc,char*argv[])
{
#ifdef sybase
   dberrhandle(err_handler);
   dbmsghandle(msg_handler);
#else
```

```
    dberrhandle((int(__cdecl *)(void))err_handler);
    dbmsghandle((int(__cdecl *)(void))msg_handler);
#endif

    DbmsSession * session = OpenDbmsSession("fred",SYBASE49);

    if ( !session || !session->Ok() ){
        cout << "Failed to open rules database session" << endl;
        exit(1);
    }

    VRule * rules = CreateRulesEngine(session);

    if ( !rules ){ // only happens on a bad session.
        cout << "Error, unable to create VRule object" << endl;
        exit(2);
    }

    ifstream infile("testdata.txt");

    int c;
    size_t pos=0;
    CBuf buf;
    SUBSCRIPTION * p=NULL;
    OPTIONPAIR * popt;

    while (!infile.eof()){
        while (!infile.eof() && (c=infile.get()) != ’M’) ;
        if ( !infile.eof() ){
            buf[0] = c;
            for ( pos = 1; !infile.eof() && (c=infile.get())!=0x0a;pos++){
                buf[pos]=c;
            }
            pos--;
            if (pos>1){
                cout << "New message" << endl;
                cout.write(buf.ptr(),pos);
                cout << endl << "End of message " << endl;
                if ( !rules->eval("Bravo","HL7 Message",buf.ptr(),pos) ){
                    cout << "Fail, errno = " << rules->GetErrorNo();
                    cout << " - " << rules->GetErrorMessage() << endl;
                    ;
                } else{
                    cout << "ActionList: " ;
                    while ( (p=rules->getsubscription()) ){
                        cout << "SubId: " << p->SubId << " ";
                        cout << p->action << endl;
                        while ( (popt=rules->getopt()) ){
                         cout << popt->Sequence << " : ";
                         cout << popt->Name << " - ";
                         cout << popt->Value << endl;
                        }
                    }
                    cout << endl;
                }
            }

        }
    }
```

```
   return 0;
}

#include "interface.h"

extern int err_handler();
extern int msg_handler();

int err_handler(
DBPROCESS*dbproc,
int severity,
int dberr,
int oserr,
char*dberrstr,
char*oserrstr)
{
 fprintf(stderr,"DB-LIBRARY error:\n\t%s\n", dberrstr);

   if (oserr != DBNOERR)
      fprintf(stderr,"Operating-system error:\n\t%s\n", oserrstr);

   fflush(stdout);

   if ((dbproc == NULL) || (DBDEAD(dbproc)))
      return(INT_EXIT);

   return(INT_CANCEL);
}

extern int msg_handler( DBPROCESS*dbproc, DBINTmsgno,
   int msgstate, intseverity, char*msgtext,
   char*foo, char* baz, short unsigned int bar)
{
   fprintf
   (stderr,"SQL Server message %ld, state %d, severity %d:\n\t%s\n",
   msgno, msgstate, severity, msgtext);
   fflush(stdout);
   return(0);
}
```

Sample Program 2: Rules Management API

```
// This example adds the following Rules data:
// Application Group: MyAppGroup
//
// Message Type: f1
//
// Rule Name:   MyRuleName
//    Expression:F1 int= 5 & F2 int= 33
//    Subscription:MySubsName
//    Permissions:
```

```
//        Owner:    update/read
//        PUBLIC:   update/read
//
// Rule Name2:  MyRuleName2
//     Expression:F2 int= 33 | F3 int= 55
//     Subscription:MySubsName2
//     Permissions:
//        Owner:    update/read
//        PUBLIC:   read
//
// Subscription Name:MySubsName
//     Action Name:MyActionName
//        Option Names:OptionName1, OptionName2
//        Option Values:OptionValue1, OptionValue2
//     Permissions:
//        Owner:    update/read
//        PUBLIC:   read
//
//
// Subscription Name2:
//     Action Name:MyActionName
//        Option Names:OptionName1, OptionName2
//        Option Values:OptionValue1, OptionValue2
//     Permissions:
//        Owner:    read
//        PUBLIC:   read
//
//
// This example then reads the rule
//
#include <stdlib.h>
#include <fstream.h>
#include <string.h>
#include "dbtypes.h"
#include "ses.h"
#include "sqlapi.h"
#include <nnrmgr.h>
int
main(int argc, char **argv)
{
   //
   // This example has explicit variables for reading and writing
   // to allow the reader to see exactly what is being populated in
   // which structures.
   //
   DbmsSession  *session;
   NNRMgr           *pmgr;
   struct NNROperator sOper;
   struct NNRApp aakey;
   struct NNRAppData aadata;
   struct NNRApp rakey;
   struct NNRAppData radata;
   struct NNRMsg amkey;
   struct NNRMsgData amdata;
   struct NNRMsg rmkey;
   struct NNRMsgData rmdata;
   struct NNRRule arkey;
   struct NNRRuleData ardata;
   struct NNRRule rrkey;
   struct NNRRuleData rrdata;
   struct NNPermissionDataPermissionData;
   struct NNPermissionDataPublicPermissionData;
   struct NNRExp       rekey;
    struct NNRExpData   redata;
   struct NNRArg aarkey;
   struct NNRArgData aardata;
   struct NNRArg rarkey;
   struct NNRArgData rardata;
   struct NNRSubsaskey;
   struct NNRSubsData asdata;
   struct NNRSubsrskey;
   struct NNRSubsData rsdata;
   struct NNRAction aactkey;
   struct NNRActionData aactdata;
   struct NNRAction ractkey;
   struct NNRActionReadData ractdata;
   struct NNROptionoptkey;
   struct NNROptionData optdata;
   struct NNROption roptkey;
   struct NNROptionReadData roptdata;
   struct NNRComponent Component;
   struct NNUserPermissionData UserPermission;
   int             iret;
   int             ActionId = -1;
   // As usual, first you must open the session
#if defined(oracle)
   session = OpenDbmsSession("rapi",ORACLE7);
#elif defined(sybase) || defined(_MS_SQL_NT)
   session = OpenDbmsSession("rapi",SYBASE49);
#endif
   if ( (!session) || (!session->Ok()) ){
      cerr << "No session was created or was not ok" << endl;
      exit(1);
   }
   // Next, you must initialize the rules management
   pmgr = NNRMgrInit(session);
   // Now, let’s begin a transaction
   BeginXact(session);
   //
   // First let’s read the operators in preparation for adding
   // arguments using the rules management APIs.  Remember you
   // need this information to add an argument.
   //
   if (NNRMgrGetFirstOperator(pmgr, &sOper)){
      cout << "Handle, Symbol, Type: " << endl;
      cout << endl
         << "\t SUPPORTED OPERATORS" << endl;
      cout << endl
         << "\t\t" << sOper.OperatorHandle
         << "\t\t" << sOper.OperatorType
         << "\t\t" << sOper.OperatorSymbol << endl << endl;
      while (NNRMgrGetNextOperator(pmgr, &sOper)){
         cout << endl
            << "\t\t" << sOper.OperatorHandle
            << "\t\t" << sOper.OperatorType
            << "\t\t" << sOper.OperatorSymbol << endl << endl;
      }
   }
```

```
//
// Now, let’s add an application
//
NNR_CLEAR(&aakey);
NNR_CLEAR(&aadata);
strcpy(aakey.AppName, "MyAppGroup");
iret = NNRMgrAddApp(pmgr, &aakey, &aadata);
if ( iret ){
   cout << endl
      << "\tApp Group Name: " << aakey.AppName << " Added."
      << endl << endl;
} else {
   cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
   cout << " Message is: " << endl;
   cout << NNRGetErrorMessage(pmgr) << endl;
   RollbackXact(session);
}
//
// Now, if we’ve been successful up to here, add
// the message type f1.  In this case assume that
// the input format f1 has fields: F1, F2 and F3
// each delimited by commas.
//
if ( iret ){
   NNR_CLEAR(&amkey);
   NNR_CLEAR(&amdata);
   strcpy(amkey.AppName, "MyAppGroup");
   strcpy(amkey.MsgName, "f1");
   iret = NNRMgrAddMsg(pmgr, &amkey, &amdata);
   if ( iret ){
      cout << endl
         << "\tMessage Type: " << amkey.MsgName << " Added."
         << endl << endl;
   } else {
      cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
      cout << " Message is: " << endl;
      cout << NNRGetErrorMessage(pmgr) << endl;
      RollbackXact(session);
   }
}
//
// Now if we’ve been successful up to here add a rule
// The NNRMgrAddRule will add the following default (rule) permissions:
//  Owner: read/update
//  Public: read
//
if ( iret ) {
   NNR_CLEAR(&arkey);
   NNR_CLEAR(&ardata);
   strcpy(arkey.AppName, "MyAppGroup");
   strcpy(arkey.MsgName, "f1");
   strcpy(arkey.RuleName, "MyRuleName");
   ardata.RuleActive = 1;
   if (NNRMgrAddRule(pmgr, &arkey, &ardata)){
      cout << endl
         << "\tRule Name: " << arkey.RuleName << " Added."
         << endl << endl;
   } else {
      cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
```

```
            cout << " Message is: " << endl;
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    //
    // Now, if we’ve been successful, grant update permission for Public
    //
    if ( iret ) {
        NNR_CLEAR(&Component);
        Component.ComponentType = NNRCOMP_RULE;
        Component.ComponentUnion.NNRRule = &arkey;
        NN_CLEAR(&PublicPermissionData);
        strcpy(PublicPermissionData.PermissionName, "Update");
        strcpy(PublicPermissionData.PermissionValue, "Granted");
        iret = NNRMgrUpdatePublicPerm(pmgr,&Component,&PublicPermissionData);
        if ( iret ) {
            cout << "Permission updated: " << endl;
            cout << "\tName: " << PublicPermissionData.PermissionName << endl;
            cout << "\tValue: " << PublicPermissionData.PermissionValue << endl <<
endl;
        } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Error message is: " << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    //
    // Now, if we’ve been successful, add expression to rule
    //
    if ( iret ) {
        NNR_CLEAR(&rekey);
        NNR_CLEAR(&redata);
        strcpy(rekey.AppName, "MyAppGroup");
        strcpy(rekey.MsgName, "f1");
        strcpy(rekey.RuleName, "MyRuleName");
        strcpy(redata.Expression, "F1 INT= 5 & F2 INT= 33");
        iret = NNRMgrAddExpression(pmgr, &rekey, &redata);
        if ( iret ) {
              cout << "\tExpression: " << redata.Expression << " Added." << endl;
        } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl << endl;
            RollbackXact(session);
        }
    }
    //
    // Now, if we’re successful up to here, add a subscription to Rule Set and
    // associate with rule
    //
    if ( iret ) {
        NNR_CLEAR(&askey);
        NNR_CLEAR(&asdata);
        strcpy(askey.AppName, "MyAppGroup");
        strcpy(askey.MsgName, "f1");
        //
        // Because rule name is populated and the subscription was not previously
        // added to the Rule Set, the subscription will be added to the Rule Set
        // and associated with the rule.
```

```
      //
      strcpy(askey.RuleName, "MyRuleName");
      strcpy(askey.SubsName, "MySubsName");
      asdata.SubsActive = 1;
      strcpy(asdata.SubsOwner, "Me");
      strcpy(asdata.SubsComment, "MyComment");
      if (NNRMgrAddSubscription(pmgr, &askey, &asdata)) {
         cout << endl
            << "\tSubs Name: " << askey.SubsName << " Added."
            << endl << endl;
         CommitXact(session);
      } else {
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // If we’ve been successful, let’s add an action and a few options
   // to the subscription
   //
   if ( iret ) {
      NNR_CLEAR(&aactkey);
      NNR_CLEAR(&aactdata);
      strcpy(aactkey.AppName, "MyAppGroup");
      strcpy(aactkey.MsgName, "f1");
      // Rule name does not need to be populated because
      // the subscription is "owned" by the Rule Set, not a rule
      strcpy(aactkey.SubsName, "MySubsName");
      strcpy(aactkey.ActionName, "MyActionName");
      strcpy(aactkey.OptionName, "OptionName1");
      strcpy(aactdata.OptionValue, "OptionValue1");
      iret = NNRMgrAddAction(pmgr, &aactkey, &aactdata, &ActionId);
      if ( iret ) {
         cout << endl
            << "\tAction Name: " << aactkey.ActionName << " Added."
            << endl;
         cout << endl
            << "\tAction id: " << ActionId << endl << endl;
         //
         // Here’s where we actually add the second option, the
         // first option as actually added in NNRMgrAddAction above
         //
         NNR_CLEAR(&optkey);
         NNR_CLEAR(&optdata);
         strcpy(optkey.AppName, "MyAppGroup");
         strcpy(optkey.MsgName, "f1");
         strcpy(optkey.RuleName, "MyRuleName");
         strcpy(optkey.SubsName, "MySubsName");
         optkey.ActionId = ActionId;
         strcpy(optkey.OptionName, "OptionName2");
         strcpy(optdata.OptionValue, "OptionValue2");
         iret = NNRMgrAddOption(pmgr, &optkey, &optdata);
         if ( iret ) {
            cout << endl
               << "\tOption Name: " << optkey.OptionName << " Added."
               << endl << endl;
         } else {
            cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
            cout << " Message is: " << endl;
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
         }
      } else {
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << " Message is: " << endl;
         cout << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // If we’ve been successful, change the owner of the subscription
   //
   if ( iret ) {
      NNR_CLEAR(&askey);

      strcpy(askey.AppName, "MyAppGroup");
      strcpy(askey.MsgName, "f1");
      strcpy(askey.SubsName, "MySubsName");

      NNR_CLEAR(&Component);
      Component.ComponentType = NNRCOMP_SUBS;
      Component.ComponentUnion.NNRSubs = &askey;

      char NewOwner[33];
      memset( (void *)NewOwner, ’\0’, 33) );
      strcpy(NewOwner, "You");

      if (NNRMgrChangeOwner(pmgr, &Component, NewOwner)) {
         cout << endl << "\tSubscription Name: " << key.SubsName << endl;
         cout << " owner changed to " << NewOwner << endl << endl;
      } else {
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << " Message is: " << endl;
         cout << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // Now if we’ve been successful up to here. Add the second rule
   // The NNRMgrAddRule will add the following default permissions:
   //  Owner: read/update
   //  Public: read
   //
   if ( iret ) {
      NNR_CLEAR(&arkey);
      NNR_CLEAR(&ardata);
      strcpy(arkey.AppName, "MyAppGroup");
      strcpy(arkey.MsgName, "f1");
      strcpy(arkey.RuleName, "MyRuleName2");
      ardata.RuleActive = 1;
      if (NNRMgrAddRule(pmgr, &arkey, &ardata)){
         cout << endl
            << "\tRule Name: " << arkey.RuleName << " Added."
            << endl << endl;
      } else {
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << " Message is: " << endl;
```

```
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    //
    // Now, if we’ve been successful, add expression to the second rule
    //
    if ( iret ) {
        NNR_CLEAR(&rekey);
        NNR_CLEAR(&redata);
        strcpy(rekey.AppName, "MyAppGroup");
        strcpy(rekey.MsgName, "f1");
        strcpy(rekey.RuleName, "MyRuleName2");
        strcpy(redata.Expression, "F2 int= 33 | F3 int= 55");
        iret = NNRMgrAddExpression(pmgr, &rekey, &redata);
        if ( iret ) {
              cout << "\tExpression: " << redata.Expression << " Added." << endl;
        } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl << endl;
            RollbackXact(session);
        }
    }
    //
    // Now, if we’re successful, add a subscription to the Rule Set
    //
    if ( iret ) {
        NNR_CLEAR(&askey);
        NNR_CLEAR(&asdata);
        strcpy(askey.AppName, "MyAppGroup");
        strcpy(askey.MsgName, "f1");
        //
        // Do not populate the rule name, leave NULL
        // This will cause the rule to be added to the Rule Set
        //
        strcpy(askey.SubsName, "MySubsName2");
        asdata.SubsActive = 1;
        strcpy(asdata.SubsOwner, "Me");
        strcpy(asdata.SubsComment, "MyComment");
        if (NNRMgrAddSubscription(pmgr, &askey, &asdata)) {
            cout << endl
                << "\tSubs Name: " << askey.SubsName << " Added."
                << endl << endl;
            CommitXact(session);
        } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    //
    // If we’ve been successful, let’s add an action and a few options
    // to the subscription
    //
    if ( iret ) {
        NNR_CLEAR(&aactkey);
        NNR_CLEAR(&aactdata);
        strcpy(aactkey.AppName, "MyAppGroup");
        strcpy(aactkey.MsgName, "f1");
```

```
      strcpy(aactkey.SubsName, "MySubsName2");
      strcpy(aactkey.ActionName, "MyActionName");
      strcpy(aactkey.OptionName, "OptionName1");
      strcpy(aactdata.OptionValue, "OptionValue1");
      iret = NNRMgrAddAction(pmgr, &aactkey, &aactdata, &ActionId);
      if ( iret ) {
         cout << endl
            << "\tAction Name: " << aactkey.ActionName << " Added."
            << endl;
         cout << endl
            << "\tAction id: " << ActionId << endl << endl;
         //
         // Here’s where we actually add the second option, the
         // first option as actually added in NNRMgrAddAction above
         //
         NNR_CLEAR(&optkey);
         NNR_CLEAR(&optdata);
         strcpy(optkey.AppName, "MyAppGroup");
         strcpy(optkey.MsgName, "f1");
         strcpy(optkey.RuleName, "MyRuleName");
         strcpy(optkey.SubsName, "MySubsName");
         optkey.ActionId = ActionId;
         strcpy(optkey.OptionName, "OptionName2");
         strcpy(optdata.OptionValue, "OptionValue2");
         iret = NNRMgrAddOption(pmgr, &optkey, &optdata);
         if ( iret ) {
            cout << endl
               << "\tOption Name: " << optkey.OptionName << " Added."
               << endl << endl;
         } else {
            cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
            cout << " Message is: " << endl;
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
         }
      } else {
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << " Message is: " << endl;
         cout << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // Now, if we’re successful, associate the subscription to the rule
   //
   if ( iret ) {
      NNR_CLEAR(&askey);
      NNR_CLEAR(&asdata);
      strcpy(askey.AppName, "MyAppGroup");
      strcpy(askey.MsgName, "f1");
      // Rule name MUST be populated in order to associate the
      // subscription to the rule
      strcpy(askey.RuleName, "MyRuleName2");
      strcpy(askey.SubsName, "MySubsName2");
      asdata.SubsActive = 1;
      strcpy(asdata.SubsOwner, "Me");
      strcpy(asdata.SubsComment, "MyComment");
      if (NNRMgrAddSubscription(pmgr, &askey, &asdata)) {
         cout << endl
```

```
               << "\tSubs Name: " << askey.SubsName << " Added."
               << endl << endl;
            CommitXact(session);
         } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
         }
      }
      //
      // Now, if we’ve been successful, deny update permission for the Owner
      // of the subscription
      //
      if ( iret ) {
         NNR_CLEAR(&Component);
         Component.ComponentType = NNRCOMP_SUBS;
         Component.ComponentUnion.NNRSubs = &arkey;
         NN_CLEAR(&PermissionData);
         strcpy(PermissionData.PermissionName, "Update");
         strcpy(PermissionData.PermissionValue, "DenyAll");
         iret = NNRMgrUpdateOwnerPermission(pmgr,&Component,&PermissionData);
         if ( iret ) {
            cout << "Permission updated: " << endl;
            cout << "\tName: " << PermissionData.PermissionName << endl;
            cout << "\tValue: " << PermissionData.PermissionValue << endl << endl;
         } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Error message is: " << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
         }
      }
      //
      // Now, if we’re successful, duplicate MySubsName2
      //
      if ( iret ) {
         NNR_CLEAR(&askey);
         strcpy(askey.AppName, "MyAppGroup");
         strcpy(askey.MsgName, "f1");
         strcpy(askey.SubsName, "MySubsName2");

         char newSubsName[ SUBS_NAME_LEN + 1 ];
         memset( (void *)newSubsName, ’\0’, ( SUBS_NAME_LEN + 1 ) );
         strcpy(newSubsName, "MyDuplicateSub");

         iret = NNRMgrDuplicateSubscription(pmgr, &askey, newSubsName);
         if ( iret)
            cout << endl
               << "\tSubs Name: " << askey.SubsName << " Added."
               << endl << endl;
         } else {
            cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
            cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
         }
      }
      //
      // Now, if we’re successful, associate the MyDuplicateSub subscription to the
rule
      //
```

```
   if ( iret ) {
      NNR_CLEAR(&askey);
      NNR_CLEAR(&asdata);
      strcpy(askey.AppName, "MyAppGroup");
      strcpy(askey.MsgName, "f1");
      // Rule name MUST be populated in order to associate the
      // subscription to the rule
      strcpy(askey.RuleName, "MyRuleName2");
      strcpy(askey.SubsName, "MyDuplicateSub");
      asdata.SubsActive = 1;
      strcpy(asdata.SubsOwner, "Me");
      strcpy(asdata.SubsComment, "MyComment");
      if (NNRMgrAddSubscription(pmgr, &askey, &asdata)) {
         cout << endl
            << "\tSubs Name: " << askey.SubsName << " Added."
            << endl << endl;
      } else {
         cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
         cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // Now, if we’re successful, delete/dis-associate subscription from rule
   //
   if ( iret ) {
      NNR_CLEAR(&arkey);

      strcpy(arkey.AppName, "MyAppGroup");
      strcpy(arkey.MsgName, "f1");
      strcpy(arkey.RuleName, "MyRuleName2");

      char SubsName[ SUBS_NAME_LEN + 1 ];
      memset( (void *)SubsName, ’\0’, ( SUBS_NAME_LEN + 1 ) );
      strcpy(SubsName, "MyDuplicateSub");

      iret = NNRMgrDeleteSubscriptionFromRule(pmgr, &arkey, SubsName);
      if ( iret ) {
         cout << endl;
         cout << "Subscription " << SubsName << " removed from Rule ";
         cout << arkey.RuleName << endl << endl;
      } else {
         cout << "Error removing subscription from rule" << endl;
         cout << "Error number is: " << NNRGetErrorNo(pmgr) << endl;
         cout << "Message is: " << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }
   //
   // Now, if we’re successful, delete the subscription and its actions and
options
   //
   if ( iret ) {
      NNR_CLEAR(&askey);
      NNR_CLEAR(&asdata);
      strcpy(askey.AppName, "MyAppGroup");
      strcpy(askey.MsgName, "f1");
      strcpy(askey.SubsName, "MyDuplicateSub");
      iret = NNRMgrDeleteEntireSubscription(pmgr, &askey);
```

```
        if (iret == 2) {

                    cout << endl;
              cout << "User is Not the Owner - Subscription Deactivated" << endl <<
endl;
                 } else {
             if ( iret == 1 ) {
                  cout << "Subscription Deleted from Rule Set" << endl << endl;
             } else {
                  // Failure
                  cout << endl;
                    cout << "Failure deleting subscription from Rule Set" <<
                         endl << endl;
                  RollbackXact(session);
             }
        }
                }

          }
   //
   // If we’ve been successful all components of rule "MyRuleName"
   // have been added, now let’s try to read them.  First, let’s
   // read the application group
   //
   if ( iret ) {
        // Commit to get the information saved before trying to read it
        CommitXact(session);
        // Now start another transaction boundary
        BeginXact(session);
        NNR_CLEAR(&rakey);
        NNR_CLEAR(&radata);
        strcpy(rakey.AppName, "MyAppGroup");
        iret = NNRMgrReadApp(pmgr, &rakey, &radata);
        if ( iret ) {
             cout << endl
                  << "\tApp Name:\t" << rakey.AppName << endl
                  << "\tDate Change:\t" << radata.DateChange  << endl
                  << "\tChangeAction:\t" << radata.ChangeAction << endl;
        } else {
             cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
             cout << " Message is: " << endl;
             cout << NNRGetErrorMessage(pmgr) << endl;
             RollbackXact(session);
        }
   }
   //
   // Now let’s read the message type
   //
   if ( iret ) {
        NNR_CLEAR(&rmkey);
        NNR_CLEAR(&rmdata);
        strcpy(rmkey.AppName, "MyAppGroup");
        strcpy(rmkey.MsgName, "f1");
        iret = NNRMgrReadMsg(pmgr, &rmkey, &rmdata);
        if ( iret ) {
             cout << endl
                  << "\tApp Name:\t" << rmkey.AppName << endl
                  << "\tMsg Name:\t" << rmkey.MsgName << endl
                  << "\tDate Change:\t" << rmdata.DateChange  << endl
                << "\tChangeAction:\t" << rmdata.ChangeAction << endl;
        } else {
            cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
            cout << " Message is: " << endl;
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    // Now, let’s read all the rules in the Rule Set
    if ( iret ) {
        NNR_CLEAR(&rrkey);
        NNR_CLEAR(&rrdata);
        strcpy(rrkey.AppName, "MyAppGroup");
        strcpy(rrkey.MsgName, "f1");

        iret = NNRMgrGetFirstRule(pmgr, &rrkey, &rrdata);
        if ( iret ) {
            cout << endl
                << "\tApp Name:\t\t"  << rrkey.AppName << endl
                << "\tMsg Name:\t\t"  << rrkey.MsgName << endl
                << "\tRule Name:\t\t" << rrrrdata.RuleName << endl
                << "\tDate Change:\t\t"<< rrdata.DateChange  << endl
                << "\tChange Action:\t\t" << rrdata.ChangeAction << endl
                << "\tArgument Count:\t\t" << rrdata.ArgumentCount << endl
                << "\tOr Condition:\t\t" << rrdata.OrCondition << endl
                << "\tSubscriber Index:\t" << rrdata.SubscriberIndex << endl
                << "\tRule Active:\t\t" << rrdata.RuleActive << endl
                << "\tRule Enable Date:\t" << rrdata.RuleEnableDate << endl
                << "\tRule Disable Date:\t" << rrdata.RuleDisableDate << endl;
            cout << endl << endl;
            NNR_CLEAR(&rrdata);
            while (NNRMgrGetNextRule(pmgr, &rrdata) {
                cout << endl
                    << "\tApp Name:\t\t"  << rrkey.AppName << endl
                    << "\tMsg Name:\t\t"  << rrkey.MsgName << endl
                    << "\tRule Name:\t\t" << rrdata.RuleName << endl
                    << "\tDate Change:\t\t"<< rrdata.DateChange  << endl
                    << "\tChange Action:\t\t" << rrdata.ChangeAction << endl
                    << "\tArgument Count:\t\t" << rrdata.ArgumentCount << endl
                    << "\tOr Condition:\t\t" << rrdata.OrCondition << endl
                    << "\tSubscriber Index:\t" << rrdata.SubscriberIndex <<  endl
                    << "\tRule Active:\t\t" << rrdata.RuleActive << endl
                    << "\tRule Enable Date:\t" << rrdata.RuleEnableDate << endl
                    << "\tRule Disable Date:\t" << rrdata.RuleDisableDate << endl;
                cout << endl << endl;
                NNR_CLEAR(&rrdata);
            }
        } else {
            cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
            cout << " Message is: " << endl;
            cout << NNRGetErrorMessage(pmgr) << endl;
            RollbackXact(session);
        }
    }
    // Now, let’s read the expression from the rule
    if ( iret ) {
        NNR_CLEAR(&rekey);
        NNR_CLEAR(&redata);
        strcpy(rekey.AppName, "MyAppGroup");
```

```
      strcpy(rekey.MsgName, "f1");
      strcpy(rekey.RuleName, "MyRuleName");
      iret = NNRMgrReadExpression(pmgr, &rekey, &redata);
      if ( iret ) {
         cout << "App Name:\t\t"  << rekey->AppName << endl
                  << "Msg Name:\t\t"      << rekey->MsgName << endl
                  << "Rule Name:\t\t"     << rekey->RuleName << endl
               << "Expression:\t\t"    << redata->Expression << endl;
      } else {
         cout << "Read failed for rule " << rekey.RuleName << endl;
         cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
         cout << " Message is: " << NNRGetErrorMessage(pmgr) << endl;
         RollbackXact(session);
      }
   }

   // Now, let’s read the subscriptions
   if ( iret ) {
      NNR_CLEAR(&rskey);
      NNR_CLEAR(&rsdata);
      strcpy(rskey.AppName, "MyAppGroup");
      strcpy(rskey.MsgName, "f1");
      strcpy(rskey.SubsName, "MySubsName");
      iret = NNRMgrReadSubscription(pmgr, &rskey, &rsdata);
      if ( iret ) {
         cout << endl
            << "\tApp Name:\t\t"  << rskey.AppName << endl
            << "\tMsg Name:\t\t"  << rskey.MsgName << endl
            << "\tRule Name:\t\t" << rskey.RuleName << endl
            << "\tSubs Name:\t\t"<< rskey.SubsName << endl
            << "\tDate Change:\t\t"<< rsdata.DateChange  << endl
            << "\tChange Action:\t\t" << rsdata.ChangeAction << endl
            << "\tSubs Active:\t\t" << rsdata.SubsActive << endl
            << "\tSubs Enable Date:\t" << rsdata.SubsEnableDate << endl
            << "\tSubs Disable Date:\t" << rsdata.SubsDisableDate << endl
            << "\tSubs Owner:\t\t"<< rsdata.SubsOwner << endl
            << "\tSubsComment\t\t"<< rsdata.SubsComment << endl;
         cout << endl << endl;

         NNR_CLEAR(&Component);
         Component.ComponentType = NNRCOMP_SUBS;
         Component.ComponentUnion.NNRSubs = &rskey;
         NN_CLEAR(&UserPermission);

         if (NNRMgrGetFirstPerm(pmgr, &Component, &UserPermission)) {
            cout << endl
                  << "\t Name:\t\t" << UserPermission.ParticipantName << endl
            << "\tPerm Name:\t\t"
            << UserPermission.Permission.PermissionName
            << endl << "\t Permission Value:\t\t"
                      <<UserPermission.Permission.PermissionValue<<endl<<endl;
            NN_CLEAR(&UserPermission);
                 while( NNRMgrGetNextPerm(pmgr, &UserPermission) ) {
               cout << endl
                  << "\t Name:\t\t" << UserPermission.ParticipantName
               << endl
               << "\tPerm Name:\t\t"
               << UserPermission.Permission.PermissionName
               << endl << "\t Permission Value:\t\t"
```

```
                        << UserPermission.Permission.PermissionValue
                     << endl << endl;
                     NN_CLEAR(&UserPermission);
                 }
             } else {
                 cout << "Error number is " << NNRGetErrorNo(pmgr) << endl;
                 cout << " Message is: " << endl;
                 cout << NNRGetErrorMessage(pmgr) << endl;
                 RollbackXact(session);
             }
         }
     }
     // Now, let’s read the actions
     if ( iret ) {
         NNR_CLEAR(&ractkey);
         NNR_CLEAR(&ractdata);
         strcpy(ractkey.AppName, "MyAppGroup");
         strcpy(ractkey.MsgName, "f1");
         strcpy(ractkey.SubsName, "MySubsName");
         iret = NNRMgrGetFirstAction(pmgr, &ractkey, &ractdata);
         if ( iret ) {
             cout << endl
                 << "\tApp Name:\t\t"  << ractkey.AppName << endl
                 << "\tMsg Name:\t\t"  << ractkey.MsgName << endl
                 << "\tRule Name:\t\t" << ractkey.RuleName << endl
                 << "\tSubscription Name: \t"<< ractkey.SubsName << endl
                 << "\tActionName:\t\t"<< ractdata.ActionName << endl
                 << "\tDate Change:\t\t"<< ractdata.DateChange  << endl
                 << "\tChange Action:\t\t" << ractdata.ChangeAction << endl
                 << "\tOption Name:\t\t"<< ractdata.OptionName << endl
                 << "\tOption Value:\t\t"<< ractdata.OptionValue << endl;
             cout << endl << endl;
             while( NNRMgrGetNextAction(pmgr, &ractdata) ) {
                 cout << endl
                     << "\tApp Name:\t\t"  << ractkey.AppName << endl
                     << "\tMsg Name:\t\t"  << ractkey.MsgName << endl
                     << "\tRule Name:\t\t" << ractkey.RuleName << endl
                     << "\tSubscription Name: \t"<< ractkey.SubsName << endl
                     << "\tActionName:\t\t"<< ractdata.ActionName << endl
                     << "\tDate Change:\t\t"<< ractdata.DateChange  << endl
                     << "\tChange Action:\t\t" << ractdata.ChangeAction << endl
                     << "\tOption Name:\t\t"<< ractdata.OptionName << endl
                     << "\tOption Value:\t\t"<< ractdata.OptionValue << endl;
                 cout << endl << endl;
             }
         }
     }
     // Now, let’s read the action/option pairs using the option routines
     if ( iret ) {
         NNR_CLEAR(&roptkey);
         NNR_CLEAR(&roptdata);
         strcpy(roptkey.AppName, "MyAppGroup");
         strcpy(roptkey.MsgName, "f1");
         strcpy(roptkey.SubsName, "MySubsName");
         iret = NNRMgrGetFirstOption(pmgr, &roptkey, &roptdata);
         if ( iret ) {
             cout << endl
                 << "\tApp Name:\t\t"  << roptkey.AppName << endl
                 << "\tMsg Name:\t\t"  << roptkey.MsgName << endl
```

```
            << "\tRule Name:\t\t" << roptkey.RuleName << endl
            << "\tSubscription Name: \t"<< roptkey.SubsName << endl
            << "\tActionName:\t\t"<< roptdata.ActionName << endl
            << "\tDate Change:\t\t"<< roptdata.DateChange  << endl
            << "\tChange Action:\t\t" << roptdata.ChangeAction << endl
            << "\tOption Name:\t\t"<< roptdata.OptionName << endl
            << "\tOption Value:\t\t"<< roptdata.OptionValue << endl;
         cout << endl << endl;
         while( NNRMgrGetNextOption(pmgr, &roptdata) ) {
            cout << endl
               << "\tApp Name:\t\t"  << roptkey.AppName << endl
               << "\tMsg Name:\t\t"  << roptkey.MsgName << endl
               << "\tRule Name:\t\t" << roptkey.RuleName << endl
               << "\tSubscription Name: \t"<< roptkey.SubsName << endl
               << "\tActionName:\t\t"<< roptdata.ActionName << endl
               << "\tDate Change:\t\t"<< roptdata.DateChange  << endl
               << "\tChange Action:\t\t" << roptdata.ChangeAction << endl
               << "\tOption Name:\t\t"<< roptdata.OptionName << endl
               << "\tOption Value:\t\t"<< roptdata.OptionValue << endl;
            cout << endl << endl;
         }
      }
   }
   cout << endl << "\t\t NOW YOU’VE DONE RULES MANAGEMENT!" << endl << endl;

   NNRMgrClose(pmgr);
   CloseDbmsSession(session);
   return;
}
```

Appendix B
Notices

Trademarks and Service Marks

Index

Sending your comments to IBM

MQSeries Integrator

Programming Reference for NEONRules

SC34-5506-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book only and the way in which the information is presented.

To request additional publications or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By fax:
  - From outside the U.K., use your international access code followed by 44 1962 870229
  - From within the U.K., use 01962 870229

Electronically, use the appropriate network ID:

- IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
- IBMLink: HURSLEY(IDRCF)
- Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic number to which your comment applies
- Your name / address / telephone number / fax number / network ID

SC34-5506-00