IBM

MQSeries® Integrator

Programming Reference for NEONRules

Version 1.1

SC34-5506-01

Note: Before using this information, and the product it supports, be sure to read the general information under *Notices* on page 327**.**

Second edition (June 1999)
This edition applies to IBM® MQSeries Integrator, Version 1.1 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 5

Product Documentation Set 6
Summary of Changes 7
Supported Platforms and Compilers 8
Disk Space and Memory Requirements 9
MQSeries Integrator Disk Space Requirements 9
Year 2000 Readiness Disclosure 10

Chapter 2: Rules Overview 11

NEONRules Components 11
Application Groups 11
Message Types 11
Rules 12
Suggested Flow of Calls for Rules Evaluation 21
Thread-Safe Rule Evaluation 23
APIs and Header Files 27
Libraries 36

Chapter 3: Rules APIs 39

Class/Type Definitions 39
VRule Supporting Functions 46
VRule Member Functions 52
Rules Error Handling 81

Chapter 4: Rules Management APIs 85

Rules Management API Structures 87
Overall Rules Management APIs and Macros 89
Application Group Management APIs 92
Application Group Management API Structures 93
Application Group Management API Functions 97
Message Type Management APIs 111
Message Type Management API Structures 112
Message Type Management API Functions 115
Rule Management APIs 127

Rule Management API Structures....127
Rule Management API Functions....135
Permissions APIs....151
Permission Management API Structures....151
Overall Permission Macro....156
Permission API Functions....157
Operator Management APIs....169
Operator Management API Structures....169
Operator Management API Functions....170
Expression Management APIs....174
Expression Management API Structures....175
Expression Management API Functions....178
Argument Management APIs....184
Argument Management API Structures....185
Argument Management API Functions....190
Subscription Management APIs....194
Subscription Management API Structures....195
Subscription Management API Functions....203
Action Management APIs....227
Action Management API Structures....228
Action Management API Functions....234
Option Management APIs....249
Option Management API Structures....250
Option Management API Functions....256
Rules Management Error Handling....271

Chapter 5: Rules Error Messages....273

Appendix A: Operator Types....317

Appendix B: Notices....327

Trademarks and Service Marks....329

Index....331

Chapter 1
Introduction

The ***MQSeries Integrator Programming Reference for NEONRules*** provides descriptions and examples for each function in 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 to install and configure MQSeries Integrator Version 1.1.
- ***MQSeries Integrator User's Guide*** helps users understand and apply the program through its graphical user interfaces (GUIs).
- ***System Management*** is intended for system administrators and database administrators who work with MQSeries Integrator on a day-to-day basis.
- ***MQSeries Integrator Programming References*** are intended for those users who build and maintain the links between MQSeries Integrator and other applications. These documents include:
  - ***Programming Reference for NEONFormatter*** is a reference to NEONFormatter APIs for those who write applications to translate messages from one format to another.
  - ***Programming Reference for NEONRules*** is a reference to NEONRules APIs for those who write applications to perform actions based on message contents.
  - ***Application Development Guide*** assists programmers in writing applications that use MQSeries Integrator APIs.

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

Summary of Changes

This document is a major revision in support of the functional changes introduced with Version 1.1. This revision also includes maintenance and editorial changes.

Chapter 4: Rules Management APIs has been updated to include the following new APIs:

- NNRMgrGetFirstApp
- NNRMgrGetNextApp
- NNRMgrDuplicateApp
- NNRMgrDeleteEntireApp
- NNRMgrGetFirstMsg
- NNRMgrGetNextMsg
- NNRMgrDeleteEntireMsg
- NNRMgrDuplicateRule
- NNRMgrDeleteAction
- NNRMgrDeleteOption
- NNRMgrUpdateUserPerm

Chapter 4: Rules Management APIs also includes modifications to the following APIs:

- NNRExpData structure
- NNRMgrAddExpression
- NNRMgrUpdateExpression
- NNRMgrReadExpression

Chapter 5: Rules Error Messages on page 273 has been updated to include enhanced error handling. Error messages now include contextual information.

Supported Platforms and Compilers

Operating System	DBMS	Compiler
AIX 4.2, 4.3	DB2 5.0 DB2 5.2 Oracle 7.3.4 Oracle 8.0.5 Sybase 11.5 Sybase 11.9	IBM C Set ++ version 3 or later
HP-UX 10.20	DB2 5.0 DB2 5.2 Oracle 7.3.4 Oracle 8.0.5 Sybase 11.5 Sybase 11.9	HP C++ version 10.40 (HP-UX 10.20)
Solaris 2.5.1, 2.6	DB2 5.0 DB2 5.2 Oracle 7.3.4 Oracle 8.0.5 Sybase 11.5 Sybase 11.9	Sparcworks C++ compiler version 4.2
Windows NT 4.0	DB2 5.0 DB2 5.2 Oracle 7.3.4 Oracle 8.0.5 SQL Server 6.5 Sybase 11.5 Sybase 11.9	Microsoft Visual C++ version 6.0

Disk Space and Memory Requirements

Required disk space is dependent on the number of queues, formats, and rules. Recommended memory for satisfactory performance depends on message rates, message sizes, and application-specific factors. For Windows NT/SQLServer, the recommended memory is 128 MB; for other platforms, the recommended memory is 256 MB.

MQSeries Integrator Disk Space Requirements

For Solaris, the /var/tmp file system requires at least 250 MB of free space to unpack the MQSeries and MQSeries Integrator products.

The minimum database allocation requires 20 MB.

MQSeries Integrator binaries require 150 MB.

MQSeries base code and server require a minimum of 25-30 MB of disk space to be available for the product code and data.

MQSeries documentation requires 50 MB of disk space (HTML files — 35 MB, PDF files — 15 MB).

The GUI requires 40 MB.

Year 2000 Readiness Disclosure

MQSeries Integrator, when used in accordance with its associated documentation, is capable of correctly processing, providing, and/or receiving date information within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software, and firmware) used with this IBM program properly exchange accurate date information with it.

Customers should contact third party owners or vendors regarding the readiness status of their products.

IBM reserves the right to update the information shown here. For the latest information regarding levels of supported software, refer to: http://www.software.ibm.com/ts/mqseries/platforms/supported.html.

For the latest IBM statement regarding Year 2000 readiness, refer to: http://www.ibm.com/ibm/year2000/.

Chapter 2

Rules Overview

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

NEONRules Components

NEONRules components consist of the following:

- Application groups
- Message types
- Rules

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 example, you might want rules for the accounting department and the application development department separated into two groups. You might 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 NEONFormatter, a message type is the same as an input format name.

This format name is used by NEONFormatter to parse input messages for rules evaluation.

Rules

To create rules, users 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 define the following items:

- Expressions: evaluation criteria containing arguments and operators
- Subscription information: subscriptions, actions, and options
- Permission information

Expressions

The evaluation criteria is an expression that consists of fields, associated operators, and associated comparison data, connected with Boolean operators. An argument is a combination of a field name, Rules comparison operator, and comparison data that is either a static value or other field name. Field names depend on the message type or input format name, and are defined using NEONFormatter. Rules comparison operators are 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 control the evaluation priority.

Arguments

An argument is 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 Rules operators contain a type in uppercase characters and an operator, concatenated with no spaces, for example, STRING=. See Appendix B*: Operator Types* on page 317.

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 a least one space before a parenthesis or a Boolean operator.

Data types of comparison values are only checked for DATE, TIME, DATETIME, INT, FLOAT, and STRING operators.

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 cannot 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 Rules 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.

Field Names

A field name is defined by the user when an input format is defined. A rule 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 have). A field name cannot contain both single and double quotes. Field names are not checked for validity.

For a more detailed explanation of a field, see the ***Programming Reference for NEONFormatter APIs***.

Rules Comparison Operators

An operator is defined by type and associated symbol. See Appendix B *: Operator Types* on page 317.

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, for example, comparing field1 to field2, and case-sensitive string comparisons, for example, where "a" does not equal "A", are also possible.

Note:
Use EXIST_TRIM operators and STRING_TRIM operators to trim trailing blanks prior to evaluating or comparing fields. The EXIST and STRING operators will not trim trailing blanks.

Existence Operators

Existence operators enable a user to determine if a field exists and is not empty in a message. Integer, string, 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.

Operators, except for NOT_EXIST and NOT_EXIST_TRIM, will not hit if a field does not exist or is empty.

Existence operators determine if a field exists or is empty in a message. Existence operators have a TRIM option that trims trailing blanks prior to determining whether a field exists or is empty, thus making a string of blanks a nonexistent field.

Integer Operators

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

INT comparison values are valid if there are whole numbers in the integer range for the platform used, which is usually from -2.1 billion to 2.1 billion. Non-numeric characters are not allowed except for a plus sign (+) or minus sign (-) as the first character. Do not use a decimal point.

String Operators

String operators compare strings of characters. Case-sensitive operators evaluate the characters 'a' and 'A' differently. Rules can work differently on different platforms. For example, on an EBCDIC machine, the order of characters is: 'a'- 'z' < 'A'- 'Z' < '0' - '9'. In ASCII, the order of characters is: '0'

- ‘9’ < ‘A’ - ‘Z’ < ‘a’ - ‘z’. String operators (including field to field, case sensitive, and field to field case sensitive operators) can have a TRIM option that trims trailing blanks prior to comparing fields. For the TRIM operators, trailing blanks are truncated from message fields and comparison values. Therefore, a field containing a string of trailing blanks is considered empty.

STRING comparison values are valid if they are composed of NULL-terminated strings with a maximum of 64 characters.

Float Operators

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

FLOAT comparison values are valid if there is a whole number in the integer range for the platform used. The range is usually from -2.1 billion to 2.1 billion, and a decimal mantissa being a whole number with the maximum of 32 digits. Non-numeric characters not are allowed except for a plus sign (+) or minus sign (-) as the first character. A decimal point must be used.

NEONRules Date, Time, and DateTime Operators

The International ISO-8601:1988 standard date notation is used as the standard format. This format 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.

Static Date, Time, or DateTime comparison values are valid if they comply with the ISO-8601:1988 standard notation. Date, Time, and DateTime static values appearing 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.

The NEONRules Date, Time, and DateTime operators are used 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 (YYYYMMDD), the Time operators, the time portion (hhmmss) and DateTime operators, the entire value (YYYYMMDDhhmmss).

In the following example, an argument using a DATE operator compares a Date against a DateTime:

F1 DATE=F2, where F1 is a Date and F2 is a DateTime

The value of the first field (F1) is compared against only the Date portion of the second field (F2).

Note:
The visual representation of dates in the GUI does not adhere to the standard DateTime format, for example, YYYYMMDD and hhmmss. However, the Management APIs must receive Date, Time, and DateTime values in the standard DateTime format.

Specifying a Year Cutoff Value

The internal application functions of MQSeries Integrator use DateTime information for archiving, time stamping, logging, and so on. These functions use the standard C++ class libraries and use four-digit notation or Universal Coordinated Time (UTC for time stamps.These functions are Y2K compliant, given that the underlying hardware is compliant. The function and libraries used with MQSeries Integrator include the logic for correct processing of leap year before, during, and after 1/1/2000.

Within the message handling and processing functionality, date information can be embedded and reformatted. MQSeries Integrator provides Date and DateTime comparison, parsing, and reformatting functions. Date and DateTime parsing and reformatting and supported Date and DateTime rules facilities are Y2K compliant for accepting input and providing output date information. Default Date and DateTime formats use four-digit years and are Y2K compliant. MQSeries Integrator also supports two-digit years as custom field definitions. These custom formats are Y2K compliant if used as described in the following paragraphs.

MQSeries Integrator products provide the facility to resolve the century ambiguity through a Year Cutoff Number for Input field data definitions, or Input Controls, using Custom Date and Time and Custom Date definitions, which include a two-digit year notation, such as MM/DD/Y HH:MM:SS or MM/DD/YY. You must specify a Year Cutoff Number from 0 to 100 (inclusive). Using this cutoff number, NEONFormatter converts a two-digit year (YY) to a four-digit year (YYYY).

The Year Cutoff algorithm is as follows:

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

With this method, any year 00 to 100 is converted to either 19XX or 20XX.

The following are some examples of how NEONFormatter interprets the Year Cutoff number:

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

You can use the NEONFormatter API or the NEONFormatter GUI to define date-related formats. Both facilities use the same underlying libraries and both are Y2K compliant.

NEONFormatter API

For an input control that specifies a data type of custom date or date-time with a two-digit year format string, you must specify a Year Cutoff value (regardless of the output Date or DateTime string). NEONFormatter uses this value to convert the two-digit year date value to a four-digit year date value. When NEONFormatter does the conversion, it compares the year value of the input data to the specified year Cutoff value and assigns the century designation as required. For example, based on the comparison, NEONFormatter converts the year value "XX" to "20XX" (21st century year) or "19XX" (20th century year) as appropriate.

NEONFormatter GUI

In the NEONFormatter GUI, you must specify a Year Cutoff value for all input formats with a two-digit year date string. The GUI provides a field for this and defaults the field to a Year Cutoff of ’101’, which is an invalid number. You must enter a valid Year Cutoff value to continue.

Boolean Operators

A Boolean expression is a single argument or more than one argument connected by Boolean operators. Boolean algebra defines the AND operator as having higher precedence than the OR operator if no parentheses are present. Parentheses change the order of evaluation from the standard Boolean operator precedence. The implementation of Rules Boolean expressions complies with this algebraic definition.

For example, 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 around the second set of values:

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.

Note:
All arguments must be active. Therefore, all inactive arguments must be activated or deleted during the database upgrade. NNRie automatically deletes inactive rules.

Grouping Arguments

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

1. Parentheses can surround a single complete argument.

   (F1 INT= 1).

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

   (F1 INT= 1 & F2 INT= 2)

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

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

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

Permissions

Rule and Subscription permissions restrict user access to individual complete rules or subscriptions or their components in the NEONRules database. Permissions only apply to managing rule and subscription contents, not rule evaluation.

A 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 subscriptions.

A 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.

The Rules component owner 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.

Subscriptions, Actions, and Options

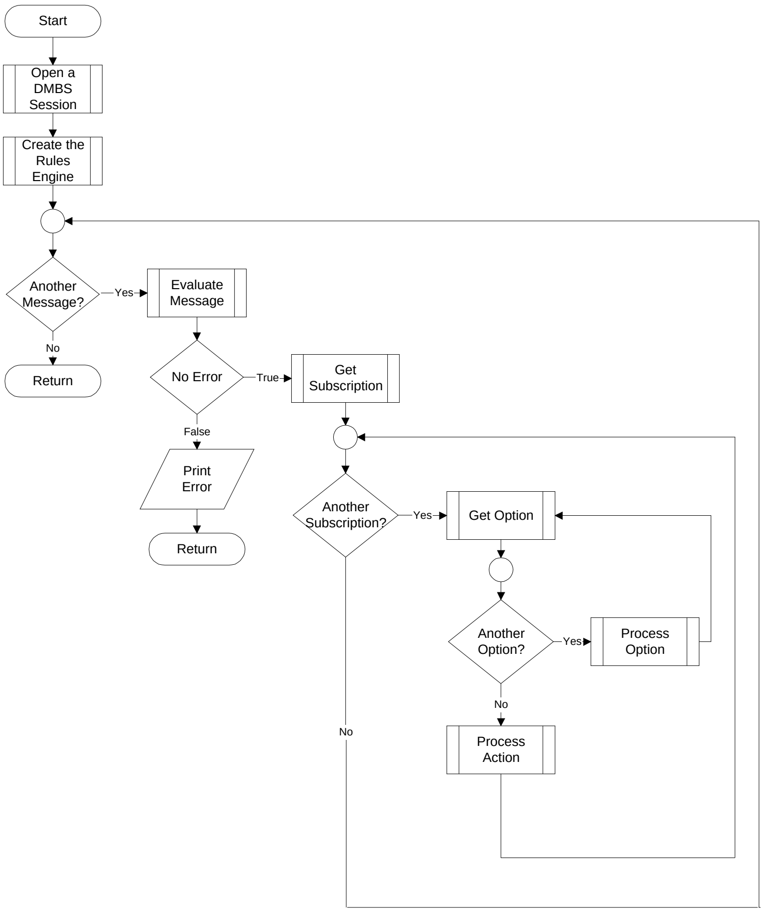
When a rule evaluates to true, it is considered a *hit*. If the rule does not evaluate to true, it is considered a *no-hit*. When a rule hits, NEONRules lets you retrieve associated subscriptions to be taken by the application. These subscriptions are the actions or commands and the associated parameters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

Suggested Flow of Calls for Rules Evaluation

Using eval(), Rules 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, DBIdentifier);
```

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 message and print it:

```
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 to determine when a new subscription has been reached each time an action is retrieved .

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 is overwritten the next time getopt is called. The user must save 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 can 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.

Evaluating Messages in a Non-Threaded Environment

The preceding *Suggested Flow of Calls for Rules Evaluation* 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.
```

Evaluating Messages in a Multi-Threaded Environment

To evaluate messages concurrently, thread-safe NEONRules APIs can be called in a multi-threaded environment. Only the Rules daemon and NEONFormatter 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()
- 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.
```

Threads A, B, and C all do the same

```
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
```

Threads B and 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

Two types of APIs exist for NEONRules: 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 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.

The APIs are 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	Evaluation structures
—	nnrmerr.h	Rules Management errors
—	rerror.h	Rules error handling

VRule Supporting Functions

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

VRule Member Functions

Return Type	Function	Arguments
int	eval	(char *AppName, char *MsgName, char *msg, int msglen, int log=0)
RULE*	gethitrule	None
RULE*	getnohitrule	None
SUBSCRIPTION*	getsubscription	None
Formatter	getformatterobject	None
OPTIONPAIR*	getopt	None
void	ThreadCleanup	None
char *	getlog	None
int	LoadRuleSet	(char *AppGrp, char*MsgType, int LoadNow=0)

Return Type	Function	Arguments
int	LoadRuleComponent	(char *AppGrp, char*MsgType, NNRComponentTypes ComponentType, char* ComponentType, int LoadNow=0)

Rules Error Handling Functions

Return Type	Function	Arguments
char*	GetErrorNo	None
char*	GetErrorMessage	None

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	NNRMgrAddApp	(NNRMgr *pMgr, const NNRApp *pRApp, const NNRAppData *pRAppData)
const long	NNRMgrReadApp	(NNRMgr *pMgr, const NNRApp *pRApp, NNRAppData *const pRAppData)
const long	NNRMgrGetFirst App	(NNRMgr *pMgr, NNRAppReadData *const RAppData)

Return Type	Function	Arguments
const long	NNRMgrGetNext App	(NNRMgr *pMgr, NNRAppReadData *const RAppData)
const long	NNRMgrDuplicate App	(NNRMgr *pMgr, const NNRApp* pRApp, const char* NewAppName)
const long	NNRMgrUpdateApp	(NNRMgr *pMgr, const NNRApp *pRApp, const NNRAppUpdate *pRAppUpdate)
const long	NNRMgrDelete EntireApp	(NNRMgr *pMgr, const NNRApp *pRApp)
const long	NNRMgrAddMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg, const NNRMsgData *pRMsgData)
const long	NNRMgrReadMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg, NNRMsgData *const pRMsgData)
const long	NNRMgrGetFirst Msg	(NNRMgr *pMgr, const NNRMsg *pRMsg, NNRMsgReadData *const pRMsgData)
const long	NNRMgrGetNext Msg	(NNRMgr *pMgr, NNRMsgReadData *const pRMsgData)
const long	NNRMgrDuplicate Msg	(NNRMgr *pMgr, const NNRMsg *pRMsg, const char *NewAppName)
const long	NNRMgrDelete EntireMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg)

Return Type	Function	Arguments
const long	NNRMgrAddRule	(NNRMgr *pMgr, const NNRRule *pRRule, const NNRRuleData *pRRuleData)
const long	NNRMgrReadRule	(NNRMgr *pMgr, const NNRRule *pRRule, NNRRuleData* const pRRuleData)
const long	NNRMgrGetFirst Rule	(NNRMgr *pMgr, const NNRRule *pRRule, NNRRuleReadData * const pRRuleData)
const long	NNRMgrGetNext Rule	(NNRMgr *pMgr, NNRRuleReadData * const pRRuleData)
const long	NNRMgrDuplicate Rule	(NNRMgr *pMgr, const NNRRule *pRRule, const char *NewRuleName)
const long	NNRMgrUpdateRule	(NNRMgr *pMgr, const NNRRule *pRule, const NNRRuleUpdate *pRRuleUpdate)
const long	NNRMgrDelete EntireRule	(NNRMgr *pMgr, const NNRRule *pRRule)
const long	NNRMgrGetFirst Perm	(NNRMgr *pRMgr, const NNRComponent * pRComponent, NNUserPermissionData const * pPermissionData)
const long	NNRMgrGetNext Perm	(NNRMgr *pRMgr, NNUserPermissionData const * pPermissionData)

Return Type	Function	Arguments
const long	NNRMgrUpdate UserPerm	(NNRMgr *pRMgr, const NNRComponent * pRComponent, const NNPermissionData * pPermission Data)
const long	NNRMgrChange Owner	(NNRMgr *pRMgr, const NNRComponent * pRComponent, char *pNewOwner)
const long	NNRMgrUpdate OwnerPerm	(NNRMgr *pRMgr, const NNRComponent * pRComponent, const NNPermissionData * pPermission Data)
const long	NNRMgrUpdate PublicPerm	(NNRMgr *pRMgr const NNRComponent * pRComponent, const NNPermission Data * pPermission Update)
const long	NNRMgrGetFirst Operator	(NNRMgr *pMgr, NNROperator * const pOperator)
const long	NNRMgrGetNext Operator	(NNRMgr *pMgr, NNROperator * const pOperator)
const long	NNRMgrAdd Expression	(NNRMgr *pMgr, const NNRExp * pRExp, NNRExpData * pRExpData)
const long	NNRMgrRead Expression	(NNRMgr *pMgr, const NNRExp * pRExp, NNRExpData * pRExpData)
const long	NNRMgrUpdate Expression	(NNRMgr *pMgr, const NNRExp *pRExp, const NNRExpData *pRExpData)

Return Type	Function	Arguments
const long	NNRMgrGetFirst Argument	(NNRMgr *pMgr, const NNRArg * pRArg, NNRArgData * const pRArgData)
const long	NNRMgrGetNext Argument	(NNRMgr *pMgr, NNRArgData * const pRArgData)
const long	NNRMgrAdd Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, const NNRSubsData *pRSubsData)
const long	NNRMgrRead Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, NNRSubsData * const pRSubsData)
const long	NNRMgrGetFirst Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, NNRSubsReadData * const pRSubsReadData)
const long	NNRMgrGetNext Subscription	(NNRMgr *pMgr, NNRSubsReadData * const pRSubsReadData)
const long	NNRMgrDuplicate Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, const char * const pNewSubsName)
const long	NNRMgrUpdate Subscription	(NNRMgr *pMgr, const NNRSubs *pRSubs, const NNRSubsUpdate *pRSubsUpdate)
const long	NNRMgrDelete SubscriptionFrom Rule	(NNRMgr *pMgr, const NNRRule * pRRule, const char * SubsName)
const long	NNRMgrDelete EntireSubscription	(NNRMgr *pMgr, const NNRRule * pRRule)

Return Type	Function	Arguments
const long	NNRMgrGetFirst RuleUsingSubs	(NNRMgr *pMgr, const NNRSubs *pRSubs, char* const pRuleName)
const long	NNRMgrGetNext RuleUsingSubs	(NNRMgr *pMgr, char* const pRuleName)
const long	NNRMgrAddAction	(NNRMgr *pMgr, const NNRAction *pRAction, const NNRActionData *pRActionData, int *pActionId)
const long	NNRMgrGetFirst Action	(NNRMgr *pMgr, const NNRAction * pRAction, NNRActionReadData * const pRActionData)
const long	NNRMgrGetNext Action	(NNRMgr *pMgr, NNRActionReadData * const pRActionData)
const long	NNRMgrResequence Action	(NNRMgr *pMgr, const NNRAction *pRAction, int oldPosition, int newPosition)
const long	NNRMgrUpdate Action	(NNRMgr *pMgr, const NNRAction *pRAction, const NNRActionUpdate *pRActionUpdate, int position)
const long	NNRMgrDelete Action	(NNRMgr *pMgr, const NNRAction *pRAction, int position)
const long	NNRMgrAddOption	(NNRMgr *pMGR, const NNROption *pROption, const NNROptionData *pROptionData)

Return Type	Function	Arguments
const long	NNRMgrGetFirst Option	(NNRMgr *pMgr, const NNROption * pROption, NNROptionReadData * const pROptionData)
const long	NNRMgrGetNext Option	(NNRMgr *pMgr, NNROptionReadData * const pROptionData)
const long	NNRMgrResequence Option	(NNRMgr *pMgr, const NNROption *pROption, int oldPosition, int newPosition)
const long	NNRMgrUpdate Option	(NNRMgr *pMgr, const NNROption *pROption, const NNROptionUpdate *pROptionUpdate, int position)
const long	NNRMgrDelete Option	(NNRMgr *pMgr, const NNROption *pROption, int Position)

Rules Management Error Handling Functions

Return Type	Function	Arguments
const int	NNRGetErrorNo	NNRMgr *pRMgr
const char*	NNRGetErrorMessage	NNRMgr *pRMgr

Libraries

NEONRules APIs must be linked with the following libraries:

Link Libraries for Rules APIs

UNIX Library	Description
libnnrulesfmt.so	NEONRules and NEONFormatter library
libnnfmgr.so	NEONRules Manager library
libnncmpntmgr.so	NEONRules Permission Management library
libnntools.so	MQSeries Integrator generic tool set
libnnaim.so	High-Level MQSeries Integrator library
libnnsql.so	MQSeries Integrator SQL Object Interface library
libnnses.so	MQSeries Integrator session-specific library
libnnsesdbold.so	MQSeries Integrator session-specific library
—	System / compiler-specific libraries
—	Database dependent libraries

Notes:

- Library file extensions are .so or .sl for UNIX, .DLL for NT, and .a for AIX. NT library names are not preceded by lib.
- For MQSeries Integrator, link with libnnmqs.so.1 and libnnMQSruleng.so.1.
- THREAD SAFETY: For multi-threading, 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.

- For MQSeries, link with libnnmqs.so, libnnsesmqs.so, and libnnsesdboldboth.so.
- You must use libnnfmgr.so and libnncmpntmgr.so when using libnnrmgr.so.

Chapter 3
Rules APIs

This chapter details NEONRules Supporting and Member Functions.

Class/Type Definitions

VRule

A VRule object is a Virtual Rules Engine instance. This class provides a standard interface for handling Rules API calls and allows the user to perform all 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

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 (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
action	char*	Action name
SubName	char*	Subscription 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

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 can be unique to an action. A NULL OPTIONPAIR in a subscription 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);
      }
      else 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

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 a list of rules that did not hit and a list of rules that hit. It should be noted that these APIs are 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;
}
```

VRule Supporting Functions

To use NEONRules 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

Link with the following libraries (UNIX) in the MQSeries Integrator library directory:

- libnnrulesfmt.so
- libnnrmrg.so

Note:
THREAD SAFETY: 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

Note:
THREAD SAFETY: 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 ***System Management***.

Parameters

Name	**Type**	**Input/ Output**	**Description**
Session	DbmsSession *	Input	Name of the 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 open MQSeries Integrator session. See OpenDbmsSession() in the ***MQSeries Integrator Application Development Guide***.
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	Errors are logged to the logfile 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 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

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pEngine	VRule*	Input	Name of the open VRule object.

Remarks

DeleteRuleEngine() must be called after CreateRulesEngine() and after all 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

Using the application group and message type, eval(), retrieves all associated active rules, parses the message into fields, and evaluates those fields based on 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. This should be the Application Group in which the user defined rules for evaluating this message. This string should not be empty.
MsgName	char*	Input	Type of message to be evaluated. If NEONFormatter 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.

Name	Type	Input/ Output	Description
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: 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 evaluate completely, regardless of the outcome; zero (0) if the evaluation fails.

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.

Note:
If this is the first eval() call for the specified Application Group / Message Type, all the rules and subscriptions for this rule set will be read into cache. Subsequent calls to eval() will not reload the data unless LoadRuleSet() or LoadRuleComponent() were called previously with LoadNow set to FALSE. Modifications to the data will only be reflected if one of the Load APIs is called prior to the eval() API. See *LoadRuleSet* on page 71 or *LoadRuleComponent* on page 74 for more information.

Example

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

See Also

CreateRulesEngine

DeleteRuleEngine

getaction

getsubscription

gethitrule

getnohitrule

GetErrorNo

GetRerror

GetErrorMessage

LoadRuleSet

LoadRuleComponent

gethitrule

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

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

Note:
THREAD SAFETY: 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 are 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

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: 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 are 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

getsubscription() gets an action within a subscription associated with a rule that evaluated 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.

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: 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 subscriptions 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

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

Syntax

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

Parameters

None

Remarks

Note:
THREAD SAFETY: 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. Each time this API is called, the returned action is removed from the list. When the return value is NULL, the list of actions has been exhausted.

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

Each subscription can 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: 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. Each time this function is called, the option is removed from the list. When the return value is NULL, the list of options for the subscription action has been exhausted.

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 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

getlog() retrieves a list of Rules error messages and returns the list in a string format. This string usually contains more information than GetErrorMessage() because 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

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

Syntax

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

Parameters

None

Return Value

Returns a 1 if an error occurs; 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

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(). It can be called before VRule::eval(). However, it should never be called after an eval() and before getsubscription(), getopt(), gethitrule(), and so on.

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 NEONFormatter is 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 reloads 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() completes 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() loads 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 cannot 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

```
// OpenDbmsSession and CreateRulesEngine called already
// Rules (VRule object) has been used for evaluations and this
// call reloads the named RuleSet

    char appgrp[APP_NAME_LEN] = "TestApp";
    char msgtype[MSG_NAME_LEN] = "TestFmt";
    int LoadImmed = 0;
    int ReloadResult = 0;

    if ( (!ReloadResult = Rules->LoadRuleSet(appgrp,msgtype,
           LoadImmed)) ) {
    cerr << "Error reloading rule set: " << appgrp << ", ";
    cerr << msgtype << endl;
    cerr << "Rules Error String > " ;
    cerr << "NNR" << Rules->GetErrorNo() << " <" ;
    cerr << Rules->GetErrorMessage() << " <" << endl;
    } else if (ReloadResult == 2) {
           cerr << "Rule Set has not been loaded yet. It will
           be when eval is called." << endl;
           } else {
           cerr << "Rule Set Reload succeeded for:
```

```
        " << appgr <<
        ", "
         << msgtype << endl;
       }

// subsequent calls to VRule::eval will use the new Rules data
```

Note:
The LoadRuleSet API returns a value of 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 does not load the rule set, instead, the load occurs when the eval() API is invoked.

See Also

CreateRulesEngine

DeleteRuleEngine

eval

GetErrorNo

GetRerror

GetErrorMessage

LoadRuleComponent

Using the application name, message type name, component type to reload, component name to reload, and the LoadNow parameter, the LoadRuleComponent() reloads the specified rule component stored in the Rules memory with the modified component data stored in the database. The MSG component type reloads the entire rule set (all rules and subscriptions for the application group / message type) and the SUB component type reloads the specified subscription. When a single subscription is reloaded, the data reloaded by the LoadRuleComponent API includes the subscription information, the subscription actions, options, and links to rules.

WARNING!
LoadRuleComponent() 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(), and so on.

Syntax

```
int VRule::LoadRuleComponent(char* AppGrp,
                             char* MsgType,
                             NNRComponentTypes ComponentType,
                             char* ComponentName,
                             int LoadNow=0);
```

Parameters

Name	Type	Input/ Output	Description
AppGrp	char*	Input	Application Group Name. Should be the Application Group for the rule set to load. If loading a subscription, the subscription being loaded must reside in the rule set defined by the application. This string should not be empty.

Name	Type	Input/ Output	Description
MsgType	char*	Input	Type of message to be evaluated. If NEONFormatter is used, message type is the input format name. Should be the message type for the rule set to load. If loading a subscription, the subscription must reside in the rule set defined by the message. This string should not be empty.
Component Type	NNR Component Types	Input	Component Type. If NNRCOMP_ MSG is used, the entire rule set is loaded; if NNRCOMP_SUBS is used, the given subscription is loaded. See *Permissions APIs* on page 151 for the NNRComponent Types definition.
Component Name	char*	Input	Component Name. If ComponentType is NNRCOMP_SUBS, this parameter is the subscription name. If the ComponentType is NNRCOMP_MSG, this parameter is the MsgType name.
LoadNow	int	Input	Indicates when to reload the rule set or subscription information.

Remarks

If you specify a subscription that does not exist in the database, the LoadRuleComponent API removes the designated subscription, along with the subscription's actions, options, and rule links, from the rules cache.

If the subscription in the database contains zero actions, it is still cached. If an associated rule does not exist in the rules cache then the subscription is loaded without that rule link.

If the LoadNow parameter is set (value equals 1), and the rule set is loaded when the reload request is received, the LoadRuleComponent API immediately reads the specified subscription from the database and updates the rules cache. If the rule set is not loaded when the reload request is received, then the entire rule set loads (performance hit).

If the LoadNow parameter is not set (value equals zero (0)), the rule set is stored and reloads the next time eval() is called. When eval() is called for the rule set, each of the stored reload requests are completed before the eval is executed. This is the suggested method.

Return Value

Returns 2 if the subscription in the LoadRuleComponent API call resides in a rule set that has not been loaded into the rules cache or does not exist in the database. This applies if the LoadNow parameter is not set (equal to 0), because the information is not checked until eval() is called. Also returns 2 if the component is not found in the database or cache and LoadNow is set.

Returns 1 if the LoadRuleComponent() succeeds. Returns 0 if the LoadRuleComponent fails, or if the reload of the rule set fails and removes the rules from cache. If the LoadNow parameter is set to 1, returns zero (0).

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

```
// OpenDbmsSession and CreateRulesEngine called already
// Rules (VRule object) has been used for evaluations and
// this call reloads the named Rule Set or Component

     char appgrp[APP_NAME _LEN] = "TestApp";
     char msgtype[MSG_NAME_LEN] = "TestFmt";
     NNRComponentTypes CompType;  // fill in
     char ComponentName[SUB_NAME_LEN];    // fill in
     char ComponentType[15];
     int LoadImmed = 0;
     int ReloadResult = 0;

      switch (CompType) {
          case NNRCOMP_MSG:
                strcpy (ComponentName, msgtype);
                strcpy (ComponentType, "Message Type");
                break;
          case NNRCOMP_SUB:
                strcpy (ComponentType, "Subscription");
                break;
          case NNRCOMP_RULE:
          case NNRCOMP_APP:
          default:
                cerr < "invalid component type" << endl;
                return 0;
                break;
        }

     if ( !(ReloadResult = Rules->LoadRuleComponent(appgrp,
            msgtype,CompType,ComponentName,LoadImmed)) ) {
     cerr << "Error reloading rule component: ";
     if (CompType == NNRCOMP_MSG) {
     cerr << "Message Type = "<< appgrp << ", " << msgtype <<
            endl;
            } else {
            cerr << ComponentType << " = "<< appgrp << ", ";
            cerr << msgtype << ", " << ComponentName << end1;
            }
            cerr << "Rules Error String > " ;
            cerr <<  "NNR" << Rules->GetErrorNo() << " <" ;
            cerr <<  Rules->GetErrorMessage() << " <" <<endl;
            } else {
            cerr << "Reload succeeded for component: ";
            if (CompType == NNRCOMP_MSG) {
            cerr <<"Message Type = "<< appgrp << ", ";
            cerr << msgtype << endl;
            } else {
            cerr << ComponentType << " = "<< appgrp << ", ";
            cerr << msgtype << ", " << ComponentName << endl;
               }
               if (ReloadResult == 2) {
               cerr << "Component not found OR rule set not
                        currently loaded. ";
               cerr << "Reload request ignored." << endl;
               }
      }
```

```
// subsequent calls to VRule::eval will use the new Rules data
```

Note:
The LoadRuleComponent API returns a value of 2 if the Rules Engine instance has never evaluated a message using the specified application group / message name pair and LoadNow is not set. In this case, the LoadRuleComponent API does not load the rule set, instead, the load occurs when the eval() API is invoked.

See Also

CreateRulesEngine

DeleteRuleEngine

eval

GetErrorNo

GetRerror

GetErrorMessage

getformatterobject

getformatterobject is a formatter object retrieval function that takes no parameters and returns the instance of the formatter that the VRule::eval() function used to parse the last input message. A user may want to use this function to retrieve the parsed fields and, therefore, not have to parse before a reformat done after the eval().

This formatter object is destroyed when the DeleteRuleEngine() destroys the VRule object. Do not access the formatter object after the VRule is deleted

Syntax

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

Parameters

None

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

```
char *appname;
char *msgname;
char *msg;
int msglen;

DbmsSession *session = OpenDbmsSession("rules", DbType);

VRule *rule = CreateRulesEngine(session);
Formatter *gFormatter = rule->getformatterobject();
```

```
if (!rule->eval(appname, msgname, msg, msglen) { // error
           if (gFormatter->GetErrorCode() ) {
               // Formatter Error.
               cerr << "Formatter Error:"
                    << gFormatter->GetErrorCode() << endl;
               cerr << "Error Message:"
                    << gFormatter->GetErrorMessage() << endl;
           }
```

Rules Error Handling

GetErrorNo

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). Zero (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

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

GetRerror() returns the description for the error number relating to a SQL or NEONRules processing error. SQL and NEONRules 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 returns the following message:

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

Note:
GetErrorMessage() returns the last error message including additional information replacing the "%s".

Syntax

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

Parameters

Name	Type	Input/ Output	Description
ErrorNo	int	Input	Determines 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, include the following header files located in the MQSeries Integrator include directory:

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

Link with the following libraries located in the MQSeries Integrator library directory (use the .a extension for AIX, .so or .sl for UNIX, and .DLL for NT):

- libnnfmgr.a
- libnnsql.a
- libnntools.so

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 NEONFormatter*** 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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises between two components named the same with only case differences.

See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. For more information, see Appendix B *: Operator Types* on page 317.

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

Rules Management API Structures

NNDate

NNDate is passed as part of an argument in several Rules Management functions and should be cleared using 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;
} NNDate;
```

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).

Name	Type	Description
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

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 open database session.

Remarks

NNRMgrInit() should be called prior to any Rules Management API calls. For information about the DbmsSession Type to use, see OpenDbmsSession() in ***MQSeries Integrator Application Development Guide***.

Return Value

Returns a pointer to an instance of a NNRMgr object.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrClose

NNRMgrClose

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	Valid Rules Management object returned from call to 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

When using Rules Management APIs, user must clear structures prior to invoking each function. Use the NNR_CLEAR macro to clear structures. NNR_CLEAR clears a structure in such a way that the Rules Management APIs can alert the user to a noninitialized 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.

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. Application Management APIs are how applications are created and associated with rules, 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 into a case-insensitive database that were exported from a case-sensitive 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 information on using NNRie.

Also, case-sensitive operators (see *Operator Management APIs* on page 169) may not work correctly on case-insensitive databases.

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

Application Group Management API Structures

NNRApp

NNRApp is passed as a pointer as the second parameter of the Application Group Management APIs. The pointer cannot 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 Also

NNR_CLEAR

NNRAppData

NNRAppData is passed as a pointer as the third parameter of some of the Application Group Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to being populated by the user or 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 Also

NNR_CLEAR

NNRAppReadData

NNRAppReadData is passed as a pointer to select functions in the Application Group Management API. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to any Application Group Management API read calls.

Syntax

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

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.
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 Also

NNR_CLEAR

NNRAppUpdate

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 Also

NNR_CLEAR

Application Group Management API Functions

NNRMgrAddApp

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.
pRApp	const NNRApp *	Input	Must be populated prior to this function call.
pRAppData	const NNRAppData *	Input	Must be populated prior to this function call. DateChange and ChangeAction should be populated with NULL values because they are provided only for future enhancements.

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 is added successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrReadApp

NNRMgrUpdateApp

NNRMgrReadApp

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.
pRApp	const NNRApp *	Input	Should be populated prior to this function call.
pRAppData	NNRAppData * const	Output	NNRMgrReadApp populates this structure. If DateChange is not NULL, it is assumed that the application group exists.

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 is read successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddApp

NNRMgrUpdateApp

NNRMgrGetFirstApp

NNRMgrGetFirstApp() provides a way to start iterating through the application groups that exist in a database. NNRMgrGetFirstApp() must be called before NNRMgrGetNextApp().

Syntax

```
const long NNRMgrGetFirstApp (
      NNRMgr *pMgr,
      NNRAppReadData *const RAppData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
RAppData	NNRAppReadData *const	Output	NNRMgrGetFirstApp populates this structure.

Remarks

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

Return Value

Returns 1 if the application is retrieved; returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message. If the error message returned is RERR_NO_MORE_APPLICATIONS, the end of the application group list was reached.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrDuplicateApp

NNRMgrDeleteEntireApp

NNRMgrGetNextApp

NNRMgrGetNextApp

NNRMgrGetNextApp() provides a way of iterating through the application groups after the first application group has been retrieved. NNRMgrGetFirstApp() must be called before NNRMgrGetNextApp().

Syntax

```
const long NNRMgrGetFirstApp (
      NNRMgr *pMgr,
      NNRAppReadData *const RAppData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
RAppData	NNRAppReadData *const	Output	NNRMgrGetNextApp populates this structure.

Remarks

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

Return Value

Returns 1 if the application is retrieved; returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message. If the error message returned is RERR_NO_MORE_APPLICATIONS, the end of the application group list was reached.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrDuplicateApp

NNRMgrDeleteEntireApp

NNRMgrGetFirstApp

NNRMgrDuplicateApp

NNRMgrDuplicateApp() creates a new application group with the name specified in the NewAppName syntax.

NNRMgrDuplicateApp() creates the message type in the specified application group, accesses each subscription in the original application group / message type pair, and duplicates the subscription and its components. The rules are then duplicated into the new application / message type pair in a similar way.

The current user is the owner of the new application group. Read permission must exist for the application group to be duplicated.

Syntax

```
const long NNRMgrDuplicateApp (
      NNRMgr *pMgr,
      const NNRApp* pRApp,
      const char* NewAppName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRApp	const NNRApp*	Input	This structure must be populated prior to this function call.
NewAppName	const char*	Input	Name of the new application group.

Return Value

Returns 1 if the application group is duplicated successfully; returns zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrDuplicateApp

NNRMgrDeleteEntireApp

NNRMgrGetFirstApp

NNRMgrGetNextApp

NNRMgrUpdateApp

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.
pRApp	const NNRApp *	Input	Must be populated prior to this function call.
pRAppUpdate	const NNRAppUpdate *	Input	Must be populated prior to this function call.

Remarks

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

Return Value

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

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteEntireApp

NNRMgrDeleteEntireApp() deletes an application group by deleting each component for the application group, including application, message type, rule, expression, and associations with subscriptions. This call depends on permissions. If the user does not have permission for each component in the application group, that component and the application group will not be deleted. However, the components that the user does have permission for will be deleted.

NNRMgrDeleteEntireApp() automatically calls NNRMgrDeleteEntireRule() and NNRMgrDeleteEntireSubscription(). NNRMgrDeleteEntireRule() deletes the rule if the current user owns and has Update permission for it. If the user is not the owner but has Update permission, the rule is deactivated. If the user does not have Update permission, the rule is not changed. Deleting a rule unlinks all the related subscriptions. NNRMgrDeleteEntireSubscription() cannot delete subscriptions that are linked to rules that were not deleted.

There may be some active and inactive rules or subscriptions left in the message type. The message type will only delete if there are not rules and subscriptions left. The application group will only delete if there are no message types left.

WARNING!
NNRMgrDeleteEntireApp() deletes all components contained within an application group.

Syntax

```
const long NNRMgrDeleteEntireApp (
      NNRMgr *pMgr,
      const NNRApp *pRApp);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRApp	NNRApp	Input	The unique identifier for the application with the message type name and subscription name.

Remarks

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

Return Value

Returns 1 if the application group and its contents are completely removed. Returns 2 if the application group still remains, but some rules or subscriptions remain due to mismatched permissions. Returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message. This does not report which rules or subscriptions could not be deleted. The user must retrieve the lists of items to find this information.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrDeleteEntireRule

NNRMgrDeleteEntireSubscription

NNRMgrDuplicateApp

NNRMgrGetFirstApp

NNRMgrGetNextApp

Message Type Management APIs

A message type identifies the type of data to which the rules apply. Message type is the same as the input format name in NEONFormatter.

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

Also, case-sensitive operators may not work correctly on case-insensitive databases. For more information, see Appendix B *: Operator Types* on page 317.

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

Message Type Management API Structures

NNRMsg

NNRMsg is passed as a pointer as the second parameter of the Message Type Management APIs. The pointer cannot 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;
} NNRMsg;
```

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 Also

NNR_CLEAR

NNRMsgData

NNRMsgData is passed as a pointer as the third parameter of the Message Type Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to being populated 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 {
        char EvalType[EVAL_TYPE_LEN];
        NNDate DateChange;
        int ChangeAction;
        long InitFlag;
}  NNRMsgData;
```

Members

Name	Type	Description
EvalType [EVAL_TYPE_LEN]	char	Defaulted for now, provided for future capability.
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 Also

NNR_CLEAR

NNRMsgReadData

NNRMsgReadData is passed as a pointer to select functions in the Message Type Management API. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to any Message Type Management API read calls.

Syntax

```
typedef struct NNRMsgReadData(
        char AppName[APP_NAME_LEN];
        char MsgName[MSG_NAME_LEN];
        NNDate DateChange;
        int ChangeAction;
        long InitFlag;
} NNRMsgReadData;
```

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 NEONFormatter.
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 Also

NNR_CLEAR

Message Type Management API Functions

NNRMgrAddMsg

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 must exist prior to associating the message type to an application group using NNRMgrAddMsg(). If you are using NEONFormatter, an input format of this name must exist. 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	Valid Rules Management object returned from call to NNRMgrInit().
pRMsg	const NNRMsg *	Input	Must be populated prior to this function call.
pRMsgData	const NNRMsgData *	Input	Default the DateChange and ChangeAction parameters to NULL This is provided only for future enhancements.

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 is added successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrReadMsg

NNRMgrReadMsg

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	Valid Rules Management object returned from call to NNRMgrInit().
pRMsg	const NNRMsg *	Input	Must be populated prior to this function call.
pRMsgData	NNRMsgData *const	Output	NNRMgrReadMsg() populates this structure. If DateChange is not NULL, the user can assume a message exists.

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 is read successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrGetFirstMsg

NNRMgrGetFirstMsg() provides a way to start iterating through the message types that exist in a database. NNRMgrGetFirstMsg() must be called before NNRMgrGetNextMsg().

Syntax

```
const long NNRMgrGetFirstMsg(
      NNRMgr *pMgr,
      const NNRMsg *pRMsg,
      NNRMsgReadData *const pRMsgData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRMsg	const NNRMsg *	Input	Should be populated prior to this function call. This must contain the correct application group name.
pRMsgData	NNRMsgData *const	Output	NNRMgrGetFirstMsg() populates this structure. If DateChange is non-NULL, the user should assume a message exists.

Remarks

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

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 a message type is retrieved; returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message. If the error number returned is RERR_NO_MORE_MESSAGES, the end of the message type list was reached.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrDeleteEntireMsg

NNRMgrDuplicateMsg

NNRMgrGetNextMsg

NNRMgrReadMsg

NNRMgrGetNextMsg

NNRMgrGetNextMsg() provides a way of iterating through the message types after the first message type has been retrieved. NNRMgrGetFirstMsg() must be called before NNRMgrGetNextMsg().

Syntax

```
const long NNRMgrGetNextMsg(
      NNRMgr *pMgr,
      NNRMsgReadData *const pRMsgData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRMsgData	NNRMsgData *const	Output	NNRMgrGetNextMsg() populates this structure. If DateChange is not NULL, the user can assume a message exists.

Remarks

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

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 a message type is retrieved; returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message. If the error message returned is RERR_NO_MORE_MESSAGES, the end of the message type list was reached.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit
NNR_CLEAR

NNRMgrAddMsg

NNRMgrDeleteEntireMsg

NNRMgrDuplicateMsg

NNRMgrGetFirstMsg

NNRMgrReadMsg

NNRMgrDuplicateMsg

NNRMgrDuplicateMsg() creates a new message type under the application group specified in the NewAppName syntax. If the application group entered in NewAppName does not exist, NNRMgrDuplicateMsg() also creates the application group.

NNRMgrDuplicateMsg() creates the message type in the application group specified in the NewAppName syntax, accesses each subscription in the original application group / message type pair, and duplicates the subscription and its components.The rules are then duplicated into the new application / message type pair in a similar way.

The current user is the owner of the new message type. Read permission must exist for the message type to be duplicated.

Syntax

```
const long NNRMgrDuplicateMsg(
      NNRMgr *pMgr,
      const NNRMsg *pRMsg,
      const char *NewAppName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRMsg	const NNRMsg *	Input	Must be populated prior to this function call.
NewAppName	const char *	Input	Enter the application group name for the message type to be duplicated in.

Remarks

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

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 type and its contents are completely duplicated. Returns zero (0) if an error occurs, for example, the message type already exists in the new application group.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrDeleteEntireMsg

NNRMgrReadMsg

NNRMgrDeleteEntireMsg

NNRMgrDeleteEntireMsg() deletes a message type by deleting each component for the message type, including message type, rule, expression, and associations with subscriptions. This call depends on permissions. If the user does not have permission for each component of the message type, that component and the message type are not deleted. However, the components that the user does have permission for will delete.

NNRMgrDeleteEntireMsg() automatically calls NNRMgrDeleteEntireRule() and NNRMgrDeleteEntireSubscription(). NNRMgrDeleteEntireRule() deletes the rule if the current user owns and has Update permission for it. If the user is not the owner but has Update permission, the rule is deactivated. If the user does not have Update permission, the rule is not changed. Deleting a rule unlinks all the related subscriptions. NNRMgrDeleteEntireSubscription() cannot delete subscriptions that are linked to rules that were not deleted.

There may be some active and inactive rules or subscriptions left in the message type. The message type will only delete if there are not rules and subscriptions left.

Syntax

```
const long NNRMgrDeleteEntireMsg(
      NNRMgr *pMgr,
      const NNRMsg *pRMsg);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRMsg	const NNRMsg *	Input	Should be populated prior to this function call.

Remarks

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

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 type and its contents are completely removed; returns 2 if the message type still remains, but some rules or subscription remain due to mismatched permissions; returns zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrDuplicateMsg

NNRMgrReadMsg

Rule Management APIs

Rule Management APIs are used to create rules that contain expressions and associate rules 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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises between two components named the same with only case differences. See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. For more information, see Appendix B *: Operator Types* on page 317.

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

Rule Management API Structures

NNRRule

NNRRule is passed as a pointer as the second parameter for some of the Rule Management APIs. The pointer cannot 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;
} NNRRule;
```

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. If the user is using NEONFormatter, 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 Also

NNR_CLEAR

NNRRuleData

NNRRuleData is passed as a pointer as the third parameter of the Rule Management APIs. The pointer cannot be NULL and must be cleared using 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;
} NNRRuleData;
```

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.

Name	Type	Description
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

See Also

NNR_CLEAR

NNRRuleReadData

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;
} NNRRuleReadData;
```

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.

Name	Type	Description
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 Also

NNR_CLEAR

NNRRuleUpdate

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;
} NNRRuleUpdate;
```

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.

Name	Type	Description
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 Also

NNR_CLEAR

Rule Management API Functions

NNRMgrAddRule

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 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 the 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	Valid Rules Management object returned from call to NNRMgrInit().
pPRRule	const NNRRule *	Input	Should be populated prior to this function call.

Name	Type	Input/ Output	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).

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 is added successfully; zero (0) if an error occurs. An error can occur if the component cannot be stored, if either the owner or PUBLIC cannot be stored, or if the Read or Update permissions for both the owner and PUBLIC cannot be stored.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrReadRule

NNRMgrUpdateOwnerPerm

NNRMgrUpdatePublicPerm

NNRMgrReadRule

NNRMgrReadRule() enables the user to retrieve rule management information. Note that this API 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 is checked. If the user is the owner or another user with Read permissions for the rule, the user can see the rule information. If the user attempting to access rule information does not have a minimum of Read access, an error is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRRule	const NNRRule *	Input	Should be populated prior to this function call.
pRRuleData	NNRRuleData* const	Output	NNRMgrReadRule() populates this structure. If DateChange is not NULL, this rule exists.

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 is read successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit
NNR_CLEAR

NNRMgrAddRule

NNRMgrGetFirstRule

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 is 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 attempting to access rule information does not have a minimum of Read access, an error is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRRule	const NNRRule *	Input	Must be completely populated except for the RuleName field prior to this function call.
pRRuleData	NNRRule Read Data *const	Output	NNRMgrGetFirstRule populates this structure.

Remarks

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

Return Value

Returns 1 if the rule is retrieved successfully; zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrUpdateRule

NNRMgrAddRule

NNRMgrReadRule

NNRMgrDeleteEntireRule

NNRMgrGetNextRule

NNRMgrGetNextRule

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 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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRRuleRead Data	NNRRuleRead Data const *	Output	NNRMgrGetFirstRule populates this structure.

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 is retrieved successfully; zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrUpdateRule

NNRMgrAddRule

NNRMgrReadRule

NNRMgrDeleteEntireRule

NNRMgrGetFirstRule

NNRMgrDuplicateRule

NNRMgrDuplicateRule() creates a new rule under the same application group / message type pair. Specify the new rule name in the NewRuleName syntax.

The current user is the owner of the new rule. Read permission must exist for the rule to be duplicated.

Syntax

```
const long NNRMgrDuplicateRule(
      NNRMgr *pMgr,
      const NNRRule *pRRule,
      const char *NewRuleName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pPRRule	const NNRRule *	Input	Should be populated prior to this function call.
NewRule Name	const char	Input	Enter the new rule name. The duplicated rule is created under the same application group / message type pair.

Remarks

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

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 and its contents are completely duplicated; returns zero (0) if an error occurs; for example, the new rule exists.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrReadRule

NNRMgrUpdateOwnerPerm

NNRMgrUpdatePublicPerm

NNRMgrUpdateRule

NNRMgrUpdateRule() enables the user to update a rule associated with a specific application and 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 with Update permission for the rule, the user can update the rule information. If the user does not have Update access, an error is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRule	const NNRRule *	Input	Must be populated prior to this function call.
pRRuleUpdate	const NNRRuleUpdate *	Input	Should be populated prior to this function call.

Remarks

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

Return Value

Returns 1 if the rule is updated successfully; zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteEntireRule() deletes a rule by deleting each component for the rule, including 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.

WARNING!
NNRMgrDeleteEntireRule() deletes all components associated with a rule. The user should only call this API to delete a rule.

When deleting rule management information, user permission to update the rule is checked. If the user is the owner and has Update permissions for the rule, the rule can be deleted. If the user is not the owner but does have Update permission, the rule is set to inactive but not deleted. If the user does not have Update permission, an error is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRRule	const NNRRule *	Input	Must be populated prior to this function call.

Remarks

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

Return Value

Returns 1 if the rule is deleted successfully; returns 2 if the rule is deactivated; returns zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

Permissions APIs

When a rule is added using NNRMgrAddRule(), 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 PUBLIC using other Permissions APIs.

Permission Management API Structures

NNUserPermissionData

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

Syntax

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

Parameters

Name	Type	Description
Permission	NNPermission Data	Specifies the permission for this specific participant.
ParticipantName [NN_PARTICIPANT _NAME_LEN]	char	Logon name of the user to whom the permission is being assigned. This parameter must be all capital letters for Oracle; and 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 Also

NNR_CLEAR

NNPermissionData

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

Syntax

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

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. Only Update is valid.
PermissionValue [NN_PERMISSION _NAME_LEN]	char	Value for the permission being defined for the rule and user permission. Only the Granted and DenyAll values associated with Update are valid.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

See Also

NN_CLEAR

NNRComponent

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

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

The NNRComponent is then called into a Permission API. NNRComponent can be initialized by calling NN_CLEAR before populating.

Syntax

```
typedef enum NNRComponentTypes{
        NNRCOMP_RULE   =1,
        NNRCOMP_SUBS   =2,
        NNRCOMP_APP    =3,
        NNRCOMP_MSG    =4
  }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.

Name	Type	Description
ComponentType	NNRComponentTypes	Identifies the type of component used in ComponentUnion; must be either NNRCOMP_RULE or NNRCOMP_SUBS.
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.

See Also

NNR_CLEAR

Overall Permission Macro

NN_CLEAR

When using Rules Management APIs affecting permissions, users are expected to clear structures prior to invoking each function. Structures should be cleared 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.

Return Value

None

Example

```
struct NNPermission permit;

NN_CLEAR(&permit);
```

Permission API Functions

NNRMgrGetFirstPerm

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	Valid Rules Management object returned from call to NNRMgrInit().
pRComponent	const NNR Component *	Input	Must populate prior to this function call.
pPermission Data	NNUser PermissionData* const	Output	Populated by the call to NNRMgrGetFirstPerm().

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 occurs.

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

If the error message 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NN_CLEAR

NNRMgrGetNextPerm

NNRMgrGetNextPerm

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	Valid Rules Management object returned from call to NNRMgrInit().
pPermission Data	const NNUserPermission Data *	Output	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 is read from the list successfully; zero (0) if an error occurs.

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

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NN_CLEAR

NNRMgrGetFirstPerm

NNRMgrUpdateUserPerm

NNRMgrUpdateUserPerm() enables the user to add or change permissions for a specific user. Only the owner of the permission can call NNRMgrUpdateUserPerm().

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRComponent	const NNRComponent *	Input	Must be populated prior to this function call.
pPermissionData	constNNUser PermissionData *	Input	Must be populated prior to this function call. This must include a valid database user name and a valid permission name / value pair (Name = Owner, Update; Value = Granted, DenyAll).

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 permission is added or updated. Returns zero (0) if the input parameters are not initialized with NNR_CLEAR and NN_CLEAR, the current user is not the owner of the item, the given user is invalid, the permission name/value is invalid, or a different error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NN_CLEAR

NNRMgrUpdatePublicPerm

NNRMgrChangeOwner

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	Valid Rules Management object returned from call to NNRMgrInit().
pRComponent	const NNRComponent *	Input	Must be populated prior to this function call.
pNewOwner	char *	Input	Must be populated with the new owner's logon 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 logon name.

Return Value

Returns 1 if the owner is changed successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit
NN_CLEAR

NNRMgrUpdateOwnerPerm

NNRMgrUpdatePublicPerm

NNRMgrUpdateOwnerPerm

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	Valid Rules Management object returned from call to NNRMgrInit().
pRComponent	const NNRComponent *	Input	Must be populated prior to this function call.
pPermissionData	const NNPermission Data *	Input	Must be populated prior to this function call.

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 are updated successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NN_CLEAR

NNRMgrUpdatePublicPerm

NNRMgrUpdatePublicPerm

NNRMgrUpdatePublicPerm() enables the owner to change permissions for another user. Only the owner can change permissions for other users. By default, other users (PUBLIC) are 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	Valid Rules Management object returned from call to NNRMgrInit().
pRComponent	const NNRComponent *	Input	Should be populated prior to this function call.
pPermissionData	const NNPermission Data *	Input	Should be populated prior to this function call.

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 is added successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NN_CLEAR

NNR_CLEAR

NNRMgrUpdateOwnerPerm

Operator Management APIs

Operator Management API Structures

NNROperator

NNROperator is passed as a pointer to the second parameter of the Operator Management APIs. The pointer cannot be NULL and must be cleared using 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

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. After using NNRMgrGetFirstOperator() to return the first operator in the pOperator parameter, the user should call NNRMgrGetNextOperator().

The pOperator structure contains a unique operator specified by a symbol, type, and handle. The operator type and operator symbol provide a means for the user to choose the operator symbol to provide the expression addition and update functions: NNRMgrAddExpression() 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.
pOperator	NNROperator * const	Output	Populated by NNRMgrGetFirstOperator().

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 number 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

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. After using NNRMgrGetFirstOperator() to return the first operator in the pOperator parameter, the user should call NNRMgrGetNextOperator().

The pOperator structure contains a unique operator specified by a symbol, type, and handle. The operator type and operator symbol provide a means for the user to choose the operator symbol to provide the expression addition and update functions: NNRMgrAddExpression() 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.
pOperator	NNROperator * const	Output	Populated by NNRMgrGetFirstOperator().

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 number 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

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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises when two components are named the same with only case differences. See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. For more information, see *Operator Types* on page 317.

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

Expression Management API Structures

NNRExp

NNRExp is passed as an argument in several Rules Management APIs to identify what rule owns the Expression. It should be cleared using 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.

NNRExpData

NNRExpData is passed as an argument in several Rules Management APIs affecting Rule expressions. It should be cleared using 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
        } NNRExpData;
```

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.

Name	Type	Description
Expression [EXPRESSION_ LEN]	char	Boolean expression containing arguments and Boolean operators AND (&) and OR (|) with parentheses to determine order of evaluation. Allows the user to add, update, and read rule expressions up to 4096 characters long plus the terminating NULL.

Expression Management API Functions

NNRMgrAddExpression

NNRMgrAddExpression() adds an expression to a rule. A rule can have only one expression containing any number of arguments. 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 is checked. If the user is the owner or has update permission for the rule, the user can add the expression information. If the user does not have update access, an error is returned indicating that the user does not have update permission and no change occurs.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRExp	const NNRExp *	Input	Must be populated prior to this function call.

Name	Type	Input/ Output	Description
pRExpData	const NNRExpData *	Input	DateChange, ChangeAction, EnableDate and DisableDate must be set to NULL; 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.

Return Value

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

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrDeleteEntireRule

NNRMgrReadExpression

NNRMgrUpdateExpression

NNRMgrReadExpression

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, the user can see the rule information. If the user attempting to access rule information does not have read access, an error is returned, indicating 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	Valid Rules Management object returned from call to NNRMgrInit().
pRExp	const NNRExp *	Input	Must be populated prior to this function call.
pRExpData	const NNRExpData *	Output	Populate this structure using NNRMgrReadExpression().

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 occurred.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrDeleteEntireRule
NNRMgrAddExpression

NNRMgrUpdateExpression

NNRMgrUpdateExpression

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 is 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 is returned indicating that the user does not have update permission and no change will occur.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRExp	const NNRExp *	Input	Must be populated prior to this function call.
pRExpData	const NNRExpData *	Input	DateChange, ChangeAction, EnableDate and DisableDate must be set to NULL; 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.

Return Value

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

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises when two components are named the same with only case differences. See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. See *Appendix B: Operator Types*.

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

Argument Management API Structures

NNRArg

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;
} NNRArg;
```

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.

NNRArgData

NNRArgData is passed as a pointer as the third parameter of select Argument Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to being populated 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;
} NNRArgData;
```

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().

Name	Type	Description
SecondFieldName [SECOND_FIELD_ NAME_LEN]	char	Value to which the field will be compared for a field to field operator.
ArgValue[ARG_V ALUE_LEN]	char	Value of the comparison (static value).
ArgActive	int	Specifies whether the argument is active (value of 1). For release 4.0 and later, all arguments MUST be active.
ArgEnableDate	NNDate	For future enhancements, ignore for now.
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.

NNRArgUpdate

NNRArgUpdate 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;
} NNRArgUpdate;
```

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 is compared for a field to field operator.

Name	Type	Description
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 4.0 and later, all arguments must be active.
ArgEnableDate	NNDate	Defaulted for now, provided for future capability.
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.

Argument Management API Functions

NNRMgrGetFirstArgument

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

When retrieving argument information, user permission to read the rule is 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 is returned indicating that the user does not have Read permission.

Note:
The arguments are not necessarily grouped together with the Boolean AND (&) 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
pMgr	NNRMgr *	Input	Name of a current Rules Management object.
pRArg	const NNRArg *	Input	Must be populated prior to this API call.

Name	Type	Input/ Output	Description
pRArgData	NNRArgData * const	Output	NNRMgrGetFirstArgument() populates this structure.

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 is read successfully; zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetNextArgument

NNRMgrReadExpression

NNRMgrAddApp()

NNRMgrAddMsg()

NNRMgrAddRule()

NNRMgrAddExpression()

NNRMgrGetNextArgument

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 is checked. If the user is the owner or another user and 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 is returned indicating that the user does not have Read permission.

WARNING!
The arguments are not necessarily grouped together with the Boolean AND () 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
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRArgData	NNRArgData * const	Output	NNRMgrGetNextArgument() populates this structure.

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 is read successfully; zero (0) if an error occurs.

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

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, and to associate the subscription to a rule. Subscription permissions 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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises when two components are named the same with only case differences. See ***System Management*** for information on using NNRie.

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

Subscription Management API Structures

NNRSubs

NNRSubs is passed as a pointer as the second parameter of select Subscription Management APIs. This pointer cannot 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;
        } NNRSubs;
```

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.

Name	Type	Description
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.

NNRSubsData

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;
        } NNRSubsData;
```

Parameters

Name	Type	Description
DateChange	NNDate	Defaulted for now, provided for future capability.
ChangeAction	int	Defaulted for now, provided for future capability.
SubsActive	int	Value of 1indicates that the subscription is active, a value of zero (0) indicates that the subscription is inactive.
SubsEnableDate	NNDate	Provided for future functionality, ignored for now.
SubsDisableDate	NNDate	Provided for future functionality, ignored for now.

Name	Type	Description
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 Also

NNR_CLEAR

NNRSubsReadData

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;
        } NNRSubsReadData;
```

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.

Name	Type	Description
ChangeAction	int	Defaulted for now, provided for future capability.
SubsActive	int	Value of 1indicates 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	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 Also

NNR_CLEAR

NNRSubsUpdate

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;
        } NNRSubsUpdate;
```

Parameters

Name	Type	Description
SubsName[SUBS_ NAME_LEN]	char	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.

Name	Type	Description
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 Also

NNR_CLEAR

Subscription Management API Functions

NNRMgrAddSubscription

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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubs	const NNRSubs *	Input	Must be populated prior to this function call. Users need not specify the rule name.

Name	Type	Input/ Output	Description
pRSubs Data	const NNRSubsData *	Input	Must 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.

If a rule name is provided, the function checks to see 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 fails and sets 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 creates the subscription and automatically adds 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 is added successfully; zero (0) if an error occurs.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddRule

NNRMgrUpdateOwnerPerm

NNRMgrUpdatePublicPerm

NNRMgrReadSubscription

NNRMgrReadSubscription

NNRMgrReadSubscription() reads subscription maintenance information for one subscription.

When retrieving subscription information, user permission to read the subscription is 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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubs	const NNRSubs *	Input	Must be populated prior to this function call. The rule name does not have to be provided in the NNRSubs structure pointed to by pRSubs.
pRSubs Data	NNRSubsData* const	Output	NNRMgrReadSubscription() populates this structure. 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.

If a rule name is provided, pRSubs verifies whether the subscription exists for the rule name and checks rule permission. If the rule name is not provided, the function verifies 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddSubscription

NNRMgrGetFirstSubscription

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(), and then call NNRMgrGetNextSubscription().

When retrieving subscription information, user permission to read the subscription is 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 does not have a minimum of Read access, an error is returned, indicating the user does not have Read permission. If the rule name is not provided, all subscriptions are 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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubs	const NNRSubs *	Input	Must be completely populated except for the SubscriptionName field prior to this function call. User need not specify a rule name.
pRSubsRead Data	NNRSubsRead Data * const	Output	Populated by this function call.

Remarks

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

The rule name does not have to be provided in the NNRSubs structure pointed to by pRSubs. If provided, the function retrieves the first subscription associated with the rule. If not provided, the function retrieves 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

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() before NNRMgrGetNextSubscription().

When retrieving subscription information, user permission to read both the rule and the subscription is checked. If the user is the owner or another user has 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 returns indicating 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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubsRead Data	NNRSubsRead Data * const	Output	Populated by this function call.

Remarks

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

Return Value

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

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDuplicateSubscription() creates a new subscription based on the subscription name provided. The new subscription has the name provided in the pNewSubsName and inherits 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	Valid Rules Management object returned from call to NNRMgrInit().
pSub	const NNRSubs*	Input	Must be populated prior to this function call.
NewSubs Name	const char* const	Input	Names of duplicate specified subscription.

Return Value

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

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetNextRuleUsingSubs

NNRMgrUpdateSubscription

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 provides the new information in the pRSubsUpdate structure.

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

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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubs	const NNRSubs *	Input	Must be populated prior to this function call. The user does not have to specify a rule name; the name is ignored.

Name	Type	Input/ Output	Description
pRSubs Update	const NNRSubsUpdate *	Input	Must be populated prior to this function call.

Remarks

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

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

Return Value

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

Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteSubscriptionFromRule

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	Valid Rules Management object returned from call to 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, is deleting the subscription, and the subscription is successfully deleted. Returns zero (0) if an error occurs. An error will occur if the user does not have update permission.

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

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNRMgrDeleteEntireSubscription

NNRMgrDeleteEntireSubscription

NNRMgrDeleteEntireSubscription() 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 is deleted. If the user is not the owner but does have Update access, the subscription is set to inactive but not deleted. If the user does not have Update access, an error is 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	Valid Rules Management object returned from call to 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNRMgrDeleteSubscriptionFromRule

NNRMgrGetFirstRuleUsingSubs

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 is 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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRSubs	const NNRSubs *	Input	Must be completely populated except for the Subscription Name field prior to this function call. User must not specify a rule name.
pRuleName	char* const	Output	Populated by this function call.

Remarks

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

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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddSubscription

NNRMgrReadSubscription

NNRMgrGetFirstSubscription

NNRMgrUpdateSubscription

NNRMgrGetNextRuleUsingSubs

NNRMgrGetNextRuleUsingSubs

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 is 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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRuleName	char* const	Output	Populated by this function call.

Remarks

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

The rule name does not have 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddSubscription

NNRMgrReadSubscription

NNRMgrGetFirstSubscription

NNRMgrUpdateSubscription

NNRMgrGetFirstRuleUsingSubs

Action Management APIs

Action are commands 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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises when two components are named the same with only case differences. See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. See Appendix B*: Operator Types* on page 317.

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

Action Management API Structures

NNRAction

NNRAction is passed as a pointer as the second parameter of select Action Management APIs. The pointer cannot 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;
        } NNRAction;
```

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.

Name	Type	Description
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.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

See Also

NNR_CLEAR

NNRActionData

NNRActionData is passed as a pointer as the third parameter of the Action Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to Action Management API calls. Use of this structure is described in the Action Management API section.

Syntax

```
typedef struct NNRActionData{
        NNDate DateChange;
        int ChangeAction;
        char OptionValue[OPTION_VALUE_LEN];
        long InitFlag;
        ) NNRActionData;
```

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 first option.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

See Also

NNR_CLEAR

NNRActionReadData

NNRActionReadData is passed as a pointer as the third parameter of select Action Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to being populated by the user or by Action Management API calls. Use of this structure is described in each 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;
        } NNRActionReadData;
```

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.

Name	Type	Description
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

NNRActionUpdate

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;
        } NNRActionUpdate;
```

Parameters

Name	Type	Description
ActionName[ACTION_ NAME_LEN]	char	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 Also

NNR_CLEAR

Action Management API Functions

NNRMgrAddAction

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 is 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 is returned indicating that the user does not have Update permission, and no change occurs.

Syntax

```
const long NNRMgrAddAction(
      NNRMgr *pMgr,
      const NNRAction *pRAction,
      const NNRActionData *pRActionData,
      int *pActionId);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRAction	const NNRAction *	Input	Must be populated prior to this function call. The rule name is ignored.

Name	Type	Input/ Output	Description
pRActionData	const NNRAction Data *	Input	DateChange and ChangeAction should be populated with NULL since they are provided only for future enhancements.
pActionId	int *	Input	Value of the action identifier used to insert all but the first option for an action.

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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstAction

NNRMgrGetNextAction

NNRMgrDeleteAction

NNRMgrGetFirstAction

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.

When retrieving action information, user permission to read the subscription is 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 is 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
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRAction	const NNRAction *	Input	Must be populated prior to this function call. RuleName, ActionName, and OptionName do not have to be populated before this call.
pRAction Data	NNRActionRead Data * const	Output	NNRMgrGetFirstAction() populates this structure.

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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetNextAction

NNRMgrAddApp()

NNRMgrAddMsg()

NNRMgrAddRule()

NNRMgrAddSubscription()

NNRMgrAddAction()

NNRMgrAddOption()

NNRMgrGetNextAction

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 is 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 is 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
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRActionData	NNRActionRead Data * const	Output	NNRMgrGetNextAction() populates this structure.

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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstAction

NNRMgrResequenceAction

NNRMgrResequenceAction() enables the user to resequence actions within a subscription. Given the current numeric position of the action, NNRMgrResequenceAction() moves 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, the following actions exist in your code:

```
putqueue(TargetQ, MessageType)
reformat(inputformat, outputformat)
```

You want the reformat should occur before the putqueue, so 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 is 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	Valid Rules Management object returned from call to NNRMgrInit().
pRAction	const NNRAction *	Input	Must be populated prior to this function call. The rule name is ignored.
oldPosition	int	Input	Old numeric position of the action to be resequenced.
newPosition	int	Input	New numeric position 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 is 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteAction

NNRMgrGetFirstAction

NNRMgrGetNextAction

NNRMgrUpdateAction

NNRMgrUpdateAction

NNRMgrUpdateAction() enables the user to update an action for a previously defined subscription. NNRMgrUpdateAction() only changes the action name. To update options, use the Option Management APIs.

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 is 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 is returned indicating the user does not have Update permission and no changes occur.

Syntax

```
const long NNRMgrUpdateAction (
      NNRMgr *pMgr,
      const NNRAction *pRAction,
      const NNRActionUpdate *pRActionUpdate,
      int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRAction	const NNRAction *	Input	Should be populated prior to this function call. The rule name is ignored.

Name	Type	Input/ Output	Description
pRActionUpdate	const NNRAction Update *	Input	Should be populated prior to this function call.
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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteAction

NNRMgrGetFirstAction

NNRMgrGetNextAction

NNRMgrResequenceAction

NNRMgrDeleteAction

NNRMgrDeleteAction deletes the specified action from a subscription. After this function is performed, the action and all its options are deleted and subsequent actions are re-sequenced.

The user must have Update permission for the subscription. If the user is the owner, the user can delete the action from a subscription. If the user attempting to delete an action is not the owner, an error is returned indicating that the user does not have Update permission and no changes occur.

Syntax

```
const long NNRMgrDeleteAction(
      NNRMgr *pMgr,
      const NNRAction *pRAction,
      int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pRAction	const NNRAction *	Input	Must be populated prior to this function call. The rule name is ignored.
position	int *	Input	Numeric order of the action to be deleted.

Remarks

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

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 deleted.

Returns zero (0) if the input parameters are not initialized with NNR_CLEAR, the current user does not have Update permission for the subscription, the action does not exist, or a different error occurs. Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstAction

NNRMgrGetNextAction

NNRMgrAddAction

Option Management APIs

Options are name-value pairs that further define an action. The first option is added with the action, and others 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 into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNRie to fail during import if a conflict arises when two components are named the same with only case differences. See ***System Management*** for information on using NNRie.

Also, case-sensitive operators may not work correctly on case-insensitive databases. See Appendix B*: Operator Types* on page 317.

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

Option Management API Structures

NNROption

NNROption is passed as a pointer as the second parameter of select Option Management APIs. The pointer cannot 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 NNROption{
        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];
        int ActionId;
        char OptionName [OPTION_NAME_LEN];
        long InitFlag;
} NNROption;
```

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. The message type is the input format name if the user is using Formatter.
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.

Name	Type	Description
SubsName[SUBS_NAME_LEN]	char	Name of the subscription associated with a message name and application group.
ActionName[ACTION_NAME_LEN]	char	Name of action.
ActionId	int	Value of the action identifier used to insert all but the first option for an action.
OptionName[OPTION_NAME_LEN]	char	Name of the option associated with this action. If this field is empty, "default" is used as the OptionName.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

NNROptionData

NNROptionData is passed as a pointer as the third parameter of the Option Management APIs. The pointer cannot be NULL and must be cleared using 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;
        } NNROptionData;
```

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 option. If this field is empty, "default" is used as the OptionValue.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a Rules Management API.

See Also

NNR_CLEAR

NNROptionReadData

NNROptionReadData is passed as a pointer as a parameter of select Option Management APIs. The pointer cannot be NULL and must be cleared using NNR_CLEAR prior to being populated 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;
        } NNROptionReadData;
```

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 option.

Name	Type	Description
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 Also

NNR_CLEAR

NNROptionUpdate

NNROptionUpdate is passed as a pointer as a parameter of select functions in the Option Management API. The pointer cannot 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;
        } NNROptionUpdate;
```

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 Also

NNR_CLEAR

Option Management API Functions

NNRMgrAddOption

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 is returned indicating that the user does not have Update permission and no change occurs.

Syntax

```
const long NNRMgrAddOption(
      NNRMgr *pMGR,
      const NNROption *pROption,
      const NNROptionData *pROptionData);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
NNROption	const NNROption *	Input	Must be populated prior to this function call. The rule name is ignored.
NNROption Data	const NNROption Data *	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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrDeleteOption

NNRMgrGetFirstOption

NNRMgrGetNextOption

NNRMgrGetFirstOption

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.

When retrieving option information, user permission to read the subscription is 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 is 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
pMgr	NNRMgr *	Input	Name of a current Rules Management object.
pROption	const NNROption *	Input	Must be populated prior to this function call. The rule name is ignored.
pROption Data	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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

If the error returned is RERR_NO_MORE_OPTIONS, no options were found for the application group and message type specified in the pROption structure.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetNextOption

NNRMgrAddApp()

NNRMgrAddMsg()

NNRMgrAddRule()

NNRMgrAddSubscription()

NNRMgrAddOption()

NNRMgrGetNextOption

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 is 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 is 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
pMgr	NNRMgr *	Input	Name of a current Rules Management object.
pROption Data	NNROption ReadData * const	Output	NNRMgrGetNextOption() populates this structure.

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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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 *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstOption

NNRMgrResequenceOption

NNRMgrResequenceOption() enables the user to resequence options within an action. Given the current numeric position of the option, NNRMgrResequenceOption() moves 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)
```

A call to NNRMgrResequenceOption switches the option in position 4 (argument3) to the option in position 3. The option in position 3 (argument2) then resides 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 is returned indicating that the user does not have Update permission, and no change occurs.

Syntax

```
const long NNRMgrResequenceOption (
      NNRMgr *pMgr,
      const NNROption *pROption,
      int oldPosition,
      int newPosition);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pROption	const NNROption *	Input	Must be populated prior to this function call. The rule name is ignored.
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 is 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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteOption

NNRMgrGetFirstOption

NNRMgrGetNextOption

NNRMgrUpdateOption

NNRMgrUpdateOption

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 is checked. The user or owner has Update permission for the rule and can update the rule information. If the user does not have Update access, an error is returned indicating that the user does not have Update permission, and no change occurs.

Syntax

```
Const long NNRMgrUpdateOption (
      NNRMgr *pMgr,
      const NNROption *pROption,
      const NNROptionUpdate *pROptionUpdate,
      int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pROption	const NNROption *	Input	Must be populated prior to this function call.

Name	Type	Input/ Output	Description
pROption Update	const NNROptionUpdate *	Input	Must be populated prior to this function call. The rule name is ignored.
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 NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() 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

NNRMgrDeleteOption

NNRMgrDeleteOption() deletes the specified option from a subscription action. This call deletes the option and resequences subsequent options for the action. If the action contains only the one option, the entire action is deleted.

The user must have Update permission for the subscription to perform this action. If the user does not have Update permission, an error is returned and no changes occur.

Syntax

```
const long NNRMgrDeleteOption(
      NNRMgr *pMGR,
      const NNROption *pROption,
      int position);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid Rules Management object returned from call to NNRMgrInit().
pROption	const NNROption *	Input	The position parameter is the Option Sequence number (starting with 1) for the Action defined by the pROption Action Id. Does not need the RuleName or OptionName populated.
position	int	Input	Numeric order of the option to be deleted.

Remarks

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 deleted.

Returns zero (0) if the input parameters are not initialized with NNR_CLEAR, the current user does not have update permission, the action or option does not exist, or a different error occurred. Use NNRGetErrorNo() to retrieve the number for the error that occurred, or use NNRGetErrorMessage() to retrieve the error message.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddOption

NNRMgrGetFirstOption

NNRMgrGetNextOption

NNRMgrResequenceOption

Rules Management Error Handling

NNRGetErrorNo

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.

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 *Rules Management API Sample Program* on page 351.

See Also

NNRGetErrorMessage

NNRMgrInit

NNRGetErrorMessage

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.

Return Value

Returns the error message for an error occurring during any of the previous Rules Management calls.

Example

See *Rules Management API Sample Program* on page 351.

See Also

NNRGetErrorNo

NNRMgrInit

Chapter 5

Rules Error Messages

The following lists of errors are available for this release and are subject to change:

- Data processing related errors
- Client code errors
- Rules Management data errors

  If you receive one of these errors, verify that the DBMS is still running properly
- General Rules Management errors

  Component refers to any item with its own permissions, for example, Rules or Subscriptions.
- Permission data errors

  Component refers to any item with its own permissions, for example, Rules or Subscriptions.
- General permission errors

The listed errors are generic. When an error code is set, the error message is enhanced with contextual information. For example, when a rule does not exist, the given Application Group name, Message Type name, and Rule name are appended to the error message with a space and dash separating each name.

Note:
Error numbers -10000 to -10099 are Rules daemon specific and are not included in this list. For more information, see ***System Management***.

Data Processing Related Errors

Code	Name	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 database. The message on the queue does 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 database. The message on the queue does not have a valid OPT_MSG_ TYPE option.	Check the Application Group and Message Type set in the eval() call. Check the OPT_APP_GRP and OPT_MSG_TYPE options for the message in the input queue.
-1003	NO_ OPERATIONS	Rules not configured or Operations missing for message	Rule data in the database is incorrect.	Run Consistency Checker to check data.
-1004	NO_ ARGUMENTS	Rules configuration missing Arguments for message	Rule missing active arguments in the database.	Run Consistency Checker to check data.

Code	Name	Message	Explanation	Response
-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.	Run Consistency Checker to check data.
-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.	This error code is used internally only as a warning. It should never appear to the user. Call technical support if it does.
-1009	GET_APP_MSG_ SQL_ERROR	Major Database Error Retrieving Application Group / Message Type	Major database error.	Verify that database is up and schema is okay.
-1010	GET_ARG_SQL_ ERROR	Major Database Error Retrieving Arguments	Major database error.	Verify that database is up and schema is okay.

Code	Name	Message	Explanation	Response
-1011	GET_BOOLEAN_ OP_SQL_ERROR	Major Database Error Retrieving Boolean Operators	Major database error	Verify that database is up and schema is okay.
-1012	GET_OPERN_ SQL_ERROR	Major Database Error Retrieving Operations	Major database error	Verify that database is up and schema is okay.
-1013	GET_RULE_SQL_ ERROR	Major Database Error Retrieving Rules	Major database error	Verify that database is up and schema is okay.
-1014	GET_SUBACT_ SQL_ERROR	Major Database Error Retrieving Subscription Actions	Major database error	Verify that database is up and schema is okay.
-1015	GET_SUBS_SQL_ ERROR	Major Database Error Retrieving Subscriptions	Major database error	Verify that database is up and schema is okay.

Client Code Errors

Code	Name	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 daemon	The Session pointer was invalid.	Check your DBMS and run Open DbmsSession() again.
-2002	EMPTY_INPUT_ MESSAGE_TYPE	NULL or missing message type provided to Rules daemon	No message type name set in eval().	Send in a valid message type.
-2003	ERROR_LOAD_ ARGUMENTS_ ADDARG	Error adding an argument to Rules daemon	(Should never see) Memory may be low.	Shut down Rules daemon 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 daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.

Code	Name	Message	Explanation	Response
-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_ NULL	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 daemon	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.
-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 daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.

Code	Name	Message	Explanation	Response
-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 daemon	(Should never see) Memory may be low.	Shut down Rules daemon 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 daemon and restart.

Code	Name	Message	Explanation	Response
-2024	ERROR_ NEGATIVE_ RULE_COUNT	INTERNAL ERROR - failed to resize rules	(Should never see) Memory may be low.	Shut down Rules daemon 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 (use apitest).
-2026	IE_TOO_MANY_ OPERATIONS	INTERNAL ERROR - incorrect operation count	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-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 daemon 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 daemon.

Code	Name	Message	Explanation	Response
-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 daemon.	Check call to OpenDbmsSession ().
-2037	INVALID_RULE_ SUBSCRIPTION	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.
-2039	EMPTY_ APPLICATION_ GROUP_NAME	Empty Input Value for Application Group Name	No application group name passed into eval().	Check call to eval().

Code	Name	Message	Explanation	Response
-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 daemon 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 daemon 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.

Code	Name	Message	Explanation	Response
-2049	FIELD_OPER_ TYPE_ MISMATCH	Field value does not have valid Month and / or Day.	A Date or DateTime comparison is not valid against Time data - the month and day are then 00.	Verify a Time value is not used in a Date comparison and that the month and day have valid non-zero values.
-2050	ERROR_ADDING _SUB_ACTION_ OPTION	Error adding Subscription Action / Option to Rules daemon.	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2051	ERROR_ADDING _SUB_RULE_ LINK	Error adding Subscription Rule Link to Rules daemon.	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2052	INVALID_ COMPONENT_ TYPE	Invalid Component Type passed into Reload Call.	For NEONRules 4.1.1, the only valid components to reload are: NNRCOMP_MSG and NNRCOMP_ SUBS.	Verify that the Load RuleComponent API is not sent Component Type NNRCOMP_ APP or NNRCOMP_ RULE.
-2053	FAILED_REMOVE _SUBSCRIPTION	Error removing Rule Subscription Link to Rules Engine.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.
-2054	FAILED_COMP_ RULE_LIST_FOR_ SUB	Error comparing old and new Subscription Rule Links.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.

Code	Name	Message	Explanation	Response
-2055	FAILED_REMOVE _RELOAD_ COMPONENT	Error Removing Reload Component from Reload List in Rules daemon.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.
-2056	FAILED_MEM_ ALLOC_ENGINE	Error allocating memory for new Rules daemon object.	(Should never see) Severe error. Memory must be low.	Shut down Rules daemon and restart.

Rules Management Data Errors

Code	Name	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.
-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, and so on.	Run Consistency Checker to check data.

Code	Name	Message	Explanation	Response
-2505	RERR_COUNTER _INSTANCE_ UPDATE	DB error Counter Instance Update	Data may be incorrect to add new Rule, Subscription, and so on.	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.

Code	Name	Message	Explanation	Response
-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.
-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.

Code	Name	Message	Explanation	Response
-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.
-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.

Code	Name	Message	Explanation	Response
-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.

Code	Name	Message	Explanation	Response
-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.
-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_ FAILED	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.

Code	Name	Message	Explanation	Response
-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.

Code	Name	Message	Explanation	Response
-2554	RERR_UPDATE_ ACTION_FAILED	DB error update action	Could not update action. May have wrong parameters.	Run Consistency Checker to check data.
-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_ FAILED	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.

Code	Name	Message	Explanation	Response
-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 operator. 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.
-2565	RERR_APP_ GROUP_DELETE	DB error application group delete failed.	Could not delete application group.	Run Consistency Checker to check data.
-2566	RERR_MSG_TYPE _DELETE	DB error message type delete failed	Could not delete message type.	Run Consistency Checker to check data.

General Rules Management Errors

Code	Name	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.

Code	Name	Message	Explanation	Response
-2608	RERR_RULE_ NAME_ ALREADY_ EXISTS	Error rule name already exists	Application group/message type pairs cannot 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.
-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.

Code	Name	Message	Explanation	Response
-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 NNRMgrGetFirst Action.	Subscription must have at least one action.
-2620	RERR_NO_MORE_OPERATORS	No more operators	Not really an error.	
-2621	RERR_NO_MORE_ARGUMENTS	No more arguments	Not really error unless returned from NNRMgrGetFirst Argument.	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.	

Code	Name	Message	Explanation	Response
-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
-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 NNRMgrGetFirst Subscription.	Rule must have at least one subscription.
-2628	RERR_NO_ MORE_RULES	No more rules	Not really an 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

Code	Name	Message	Explanation	Response
-2630	RERR_OPTION_ DOES_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 NNRMgrGetFirst Option.	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.

Code	Name	Message	Explanation	Response
-2637	RERR_EXPORT_ ARG_FAILURE	Export argument failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-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 an error.	
-2642	RERR_NO_ MORE_ APPLICATIONS	No more applications	Not really an 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.

Code	Name	Message	Explanation	Response
-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
-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.

Code	Name	Message	Explanation	Response
-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 of database	Can only export and import to version 4.1 databases.	Check version of NEONRules.
-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.	

Code	Name	Message	Explanation	Response
-2660	RERR_EXPORT_ VERSION_ FAILURE	Error exporting version	Could not retrieve version for export. Can only export from version 4.0 and higher.	Check 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.
-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.

Code	Name	Message	Explanation	Response
-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_ OPERATOR	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 must have 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.

Code	Name	Message	Explanation	Response
-2680	RERR_ARG_ MUST_BE_ ACTIVE	Arguments must be active for NEONet 4.0+	Arguments can no longer be Inactive.	Change input expression parameter.
-2681	RERR_USE_ UPDATE_EXPR	Must Use NNR MgrUpdate Expression to perform desired update	Cannot use NNRMgrAddArg ument unless all arguments are just ANDed together.	Use NNRMgrUpdate Expression.
-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	Two Operands are required 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).

Code	Name	Message	Explanation	Response
-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.
-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 NNRMgrAdd Option or NNRMgr UpdateOption.

Code	Name	Message	Explanation	Response
-2695	RERR_OPT_PUT_ PROP_INVALID	Invalid option value for putqueue MQS _PROPAGATE option	Must be PROPAGATE or NO_ PROPAGATE.	Change the parameters sent into NNRMgrAdd Option or NNRMgr UpdateOption.
-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 NNRMgrAdd Option or NNRMgr UpdateOption.
-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 NNRMgrAdd Option or NNRMgr UpdateOption.
-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 NNRMgrAdd Option or NNRMgr UpdateOption or add the needed formats.
-2699	RERR_INVALID_ INT_ARG_VALUE	Invalid integer static comparison value.	For integer comparison values, no non-numeric characters are allowed except for a (+/-) sign as the first character (No decimal point is allowed).	Check input into Argument or Expression APIs.

Code	Name	Message	Explanation	Response
-2700	RERR_INT_ARG_VALUE_OUT_OF_RANGE	Integer static comparison value out of valid range.	The valid INT values are whole numbers in the integer range for the platform used (usually about -2.1 to about 2.1 billion).	Check input into Argument or Expression APIs.
-2701	RERR_INVALID_FLOAT_ARG_VALUE	Invalid float static comparison value.	For float comparison values, the only non-numeric characters allowed are (+/-) sign as the first character and a decimal point.	Check input into Argument or Expression APIs.
-2702	RERR_FLOAT_ARG_VALUE_MISSING_DECIMAL	Float static comparison value must have a decimal.	Valid float comparison values must contain a decimal point.	Check input into Argument or Expression APIs.
-2703	RERR_FLOAT_ARG_VALUE_OUT_OF_RANGE	Float static comparison value out of valid range.	The valid FLOAT values include a whole number in the integer range for the platform used (usually about -2.1billion to about 2.1 billion) and a decimal mantissa with a maximum of 31 digits.	Check input into Argument or Expression APIs.

Code	Name	Message	Explanation	Response
-2704	RERR_STATIC_ ARG_VALUE_ TOO_LONG	Static comparison value too long.	Static comparison values cannot exceed 64 characters (plus a terminating NULL).	Check input into Argument or Expression APIs.
-2705	RERR_NO_PERM _TO_DELETE_ ALL_APP	Could not delete all rules and / or subscriptions in application group.	The user deleting might not have permissions for all the rules and / or subscriptions in the application group.	Check into the permissions for the rules and / or subscriptions. Only the owner can delete them.
-2706	RERR_NO_PERM _TO_DELETE_ ALL_MSG	Could not delete all rules and / or subscriptions in message type.	The user deleting might not have permissions for all the rules and / or subscriptions in the message type.	Check into the permissions for the rules and / or subscriptions. Only the owner can delete them.
-2707	RERR_RULE_SUB _LINK_SUB_NOT _EXIST	Error linking subscription to rule. Subscription does not exist.	Subscription was not imported.	Look at the error message as to why the subscription was not imported.
-2708	RERR_IMPORT_ EXPRESSION	Error importing expression.	Malformed expression or problem in the database.	Review the expression and run consistency checker on the database.

Code	Name	Message	Explanation	Response
-2709	RERR_WRONG_ VER_FOR_ OVERWRITE	Error. -O flag is not supported in pre 4.10 versions. The -o flag is used instead.	Due to significant changes in the NNRie file formats, NEONRules does not support the –O in import files from pre 4.10 versions.	Remove the message types you want to completely overwrite using the NEONRules GUI or NEONRules Management APIs prior to importing.
-2710	RERR_IMPORT_ VERSION_ FAILURE_FILE	Unsupported version of import file.	The import file was created from a version of NNRie.exe that is no longer supported in NEONRules.	Check the version in the import file. This might require using the NNCrypt utility. Check the version of NNRie used to create the export file.
-2711	RERR_MISSING_ VERSION_IN_ FILE	Missing version information in export file.	The version of the export file is missing.	Check the file to see that the version line is present. This might require using the NNCrypt utility. Check the version of NNRie used to create the export file.

Code	Name	Message	Explanation	Response
-2712	RERR_NOT_ RULES_FILE	Missing key information to the NNRie export file.	Missing the "R" as the first non-comment line in the NNRie export file.	Check the file to see that the "R" line is present. This might require using the NNCrypt utility. Check the version of NNRie used to create the export file.
-2713	RERR_NOTHING _IMPORTED_ EXPORTED	Nothing was imported or exported.	There are no valid lines to import or no data to export.	Check the database or the import file to see if they contain the data required.

Permission Data Errors

Code	Name	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.
-5503	NN_NODE_DOES _NOT_EXIST	Node does not exist	Must run on valid 4.1 database with node data saved.	Check installation.
-5504	NN_HIERARCHY _DOES_NOT_ EXIST	Hierarchy does not exist	Must run on valid 4.0 database with hierarchy data saved.	Check install. Run Consistency Checker to check data.
-5505	NN_ COMPONENT_ ADD_FAILURE	Component add failure	Cannot add rule component to permission system; may be duplicate.	Run Consistency Checker to check data.
-5506	NN_ COMPONENT_ LOAD_FAILURE	Component load failure	Cannot retrieve rule component information from permission system; may not exist.	Run Consistency Checker to check data.
-5507	NN_DELETE_ COMPONENT_ FAILURE	Delete component failure	Cannot delete rule component information from permission system; may not exist.	Run Consistency Checker to check data.

Code	Name	Message	Explanation	Response
-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.	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.
-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.

Code	Name	Message	Explanation	Response
-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

Code	Name	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.
-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.

Code	Name	Message	Explanation	Response
-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_ INVALID_ 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.

Code	Name	Message	Explanation	Response
-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.
-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

Operator Types

The following operator types are available for use in Rules expressions. These operator types are described in the subsequent tables:

- Existence
- Integer
- String
- Field-to-field integer
- Field-to-field string
- Float
- Case-sensitive string
- Field-to-field case-sensitive
- Date
- Field-to-field date
- Time
- Field-to-field time
- DateTime
- Field-to-field DateTime

Note:
Case-sensitive operators do not work correctly on case-insensitive databases.

Existence Operators

Operator Symbol	Operator Handle	Description
NOT_EXIST	0	Required Field Is Not Present
NOT_EXIST_TRIM	104	Required Field Is Not Present (After Trimming)
EXIST	1	Required Field Is Present
EXIST_TRIM	105	Required Field Is Present (After Trimming)

Integer Operators

Operator Symbol	Operator Handle	Description
INT=	2	Integer Equals
INT>	3	Integer Greater Than
INT<	4	Integer Less Than
INT>=	5	Integer Greater Than Or Equal To
INT<=	6	Integer Less Than Or Equal To
INT<>	7	Integer Not Equal To

String Operators

Operator Symbol	Operator Handle	Description
STRING=	8	String Equal To
STRING_TRIM=	106	String Equal To (After Trimming)
STRING>	9	String Greater Than

Operator Symbol	Operator Handle	Description
STRING_TRIM>	107	String Greater Than (After Trimming)
STRING<	10	String Less Than
STRING_TRIM<		String Less Than (After Trimming)
STRING_TRIM>=	109	String Greater Than Or Equal To (After Trimming)
STRING>=	11	String Greater Than Or Equal To
STRING<=	12	String Less Than Or Equal To
STRING_TRIM<=	110	String Less Than Or Equal To (After Trimming)
STRING<>	13	String Not Equal To
STRING_TRIM<>	111	String Not Equal To (After Trimming)

Field To Field Integer Operators

Operator Symbol	Operator Handle	Description
F2FINT=	18	Field To Field Integer Equal To
F2FINT>	19	Field to Field Integer Greater Than
F2FINT<	20	Field to Field Integer Less Than
F2FINT>=	21	Field to Field Integer Greater Than Or Equal To
F2FINT<=	22	Field to Field Integer Less Than Or Equal To

Operator Symbol	Operator Handle	Description
F2FINT<>	23	Field To Field Integer Not Equal To

Field To Field String Operators

Operator Symbol	Operator Handle	Description
F2FSTRING=	24	Field To Field String Equal To
F2FSTRING_TRIM=	112	Field To Field String Equal To (After Trimming)
F2FSTRING>	25	Field To Field String Greater Than
F2FSTRING_TRIM>	113	Field To Field String Greater Than (After Trimming)
F2FSTRING<	26	Field To Field String Less Than
F2FSTRING_TRIM<	114	Field To Field String Less Than (After Trimming)
F2FSTRING>=	27	Field To Field String Greater Than Or Equal To
F2FSTRING_TRIM>=	115	Field To Field String Greater Than Or Equal To (After Trimming)
F2FSTRING<=	28	Field To Field String Less Than Or Equal To
F2FSTRING_TRIM<=	116	Field To Field String Less Than Or Equal To (After Trimming)
F2FSTRING<>	29	Field To Field String Not Equal To

Operator Symbol	Operator Handle	Description
F2FSTRING_TRIM<>	117	Field To Field String Not Equal To (After Trimming)

Float Operators

Operator Symbol	Operator Handle	Description
FLOAT=	34	Float Equals
FLOAT>	35	Float Greater Than
FLOAT<	36	Float Less Than
FLOAT>=	37	Float Greater Than Or Equal To
FLOAT<=	38	Float Less Than Or Equal To
FLOAT<>	39	Float Not Equal To

Case Sensitive String Operators

Operator Symbol	Operator Handle	Description
CSSTRING =	56	Case Sensitive String Equal To
CSSTRING_TRIM=	118	Case Sensitive String Equal To (After Trimming)
CSSTRING>	57	Case Sensitive String Greater Than
CSSTRING_TRIM>	119	Case Sensitive String Greater Than (After Trimming)
CSSTRING<	58	Case Sensitive String Less Than

Operator Symbol	Operator Handle	Description
CSSTRING_TRIM<	120	Case Sensitive String Less Than (After Trimming)
CSSTRING>=	59	Case Sensitive String Greater Than Or Equal To
CSSTRING_TRIM>=	121	Case Sensitive String Greater Than Or Equal To (After Trimming)
CSSTRING<=	60	Case Sensitive String Less Than Or Equal To
CSSTRING_TRIM<=	122	Case Sensitive String Less Than Or Equal To (After Trimming)
CSSTRING<>	61	Case Sensitive String Not Equal To
CSSTRING_TRIM<>	123	Case Sensitive String Not Equal To (After Trimming)

Field To Field Case Sensitive Operators

Operator Symbol	Operator Handle	Description
F2FCSSTRING=	62	Field To Field Case Sensitive String Equal To
F2FCSSTRING_TRIM=	124	Field To Field Case Sensitive String Equal To (After Trimming)
F2FCSSTRING>	63	Field To Field Case Sensitive String Greater Than
F2FCSSTRING_TRIM>	125	Field To Field Case Sensitive String Greater Than (After Trimming)

Operator Symbol	Operator Handle	Description
F2FCSSTRING<	64	Field To Field Case Sensitive String Less Than
F2FCSSTRING_TRIM<	126	Field To Field Case Sensitive String Less Than (After Trimming)
F2FCSSTRING>=	65	Field To Field Case Sensitive String Greater Than Or Equal To
F2FCSSTRING_TRIM>=	127	Field To Field Case Sensitive String Greater Than Or Equal To (After Trimming)
F2FCSSTRING<=	66	Field To Field Case Sensitive String Less Than Or Equal To
F2FCSSTRING_TRIM<=	128	Field To Field Case Sensitive String Less Than Or Equal To (After Trimming)
F2FCSSTRING<>	67	Field To Field Case Sensitive String Not Equal To
F2FCSSTRING_TRIM<>	129	Field To Field Case Sensitive String Not Equal To (After Trimming)

Date Operators

Operator Symbol	Operator Handle	Description
DATE=	68	Date Equal To
DATE>	69	Date Greater Than
DATE<	70	Date Less Than
DATE>=	71	Date Greater Than Or Equal To

Operator Symbol	Operator Handle	Description
DATE<=	72	Date Less Than Or Equal To
DATE<>	73	Date Not Equal To

Field To Field Date Operators

Operator Symbol	Operator Handle	Description
F2FDATE=	74	Field To Field Date Equal To
F2FDATE>	75	Field To Field Date Greater Than
F2FDATE<	76	Field To Field Date Less Than
F2FDATE>=	77	Field To Field Date Greater Than Or Equal To
F2FDATE<=	78	Field To Field Date Less Than Or Equal To
F2FDATE<>	79	Field To Field Date Not Equal To

Time Operators

Operator Symbol	Operator Handle	Description
TIME=	80	Time Equal To
TIME>	81	Time Greater Than
TIME<	82	Time Less Than
TIME>=	83	Time Greater Than Or Equal To
TIME<=	84	Time Less Than Or Equal To
TIME<>	85	Time Not Equal To

Field To Field Time Operators

Operator Symbol	Operator Handle	Description
F2FTIME=	86	Field To Field Time Equal To
F2FTIME>	87	Field To Field Time Greater Than
F2FTIME<	88	Field To Field Time Less Than
F2FTIME>=	89	Field To Field Time Greater Than Or Equal To
F2FTIME<=	90	Field To Field Time Less Than Or Equal To
F2FTIME<>	91	Field To Field Time Not Equal To

DateTime Operators

Operator Symbol	Operator Handle	Description
DATETIME=	92	DateTime Equal To
DATETIME>	93	DateTime Greater Than
DATETIME<	94	DateTime Less Than
DATETIME>=	95	DateTime Greater Than Or Equal To
DATETIME<=	96	DateTime Less Than Or Equal To
DATETIME<>	97	DateTime Not Equal To

Field To Field DateTime Operators

Operator Symbol	Operator Handle	Description
F2FDATETIME=	98	Field To Field DateTime Equal To
F2FDATETIME>	99	Field To Field DateTime Greater Than
F2FDATETIME<	100	Field To Field DateTime Less Than
F2FDATETIME>=	101	Field To Field DateTime Greater Than Or Equal To
F2FDATETIME<=	102	Field To Field DateTime Less Than Or Equal To
F2FDATETIME<>	103	Field To Field DateTime Not Equal To

Appendix B
Notices

Trademarks and Service Marks

Index

Symbols

& operator 12
| operator 12

A

Action Management API functions
    NNRMgrAddAction 234, 247
    NNRMgrGetFirstAction 236
    NNRMgrGetNextAction 238
    NNRMgrResequenceAction 240
    NNRMgrUpdateAction 244
Action Management APIs 227
    NNRAction 228
    NNRActionData 230
    NNRActionReadData 231
    NNRActionUpdate 233
actions 20, 227
AND operator 12
APIs
    action management 227
    application groups 92
    argument management 184
    expression management 174
    header files 27
    member functions 27
    message types 111, 115
    option management 249
    permissions 151
    Rules 39
    Rules error handling function 27
    Rules Management 127
    Rules Management APIs 85
    Rules Management functions 27
    Rules Management macros 27
    subscription management 194
    VRule member functions 27
Application Group Management API functions 97
    NNRMgrReadApp 99
    NNRMgrUpdateApp 107
Application Group Management APIs 92
    NNRApp 93
    NNRAppData 94
    NNRAppUpdate 96
application groups 11, 92
Argument Management API functions
    NNRMgrGetFirstArgument 190
    NNRMgrGetNextArgument 192
Argument Management APIs 184
    NNRArg 185
    NNRArgData 186
    NNRArgUpdate 188
arguments 12

B

Boolean operators 12

C

class/type definitions 39
client code errors 273
CreateRulesEngine 39, 47

D

data processing errors 273
date operators 15
dates
    standard notation 15
datetime operators 15
definitions 39
DeleteRuleEngine 39, 50
diskspace requirements 9
documents
    NEONet documentation set 6

E

environments
- multi-threaded 24
- non-threaded 23

error codes 273
- client code errors 273
- data processing errors 273
- permission errors 273
- Rules Management data errors 273

error handling 81
eval 21, 52
Expression Management API functions
- NNRmgrAddExpression 178
- NNRMgrReadExpression 180
- NNRmgrUpdateExpression 182

Expression Management APIs 174
- NNRExp 175
- NNRExpData 176

expressions 12

G

getaction 62
GetErrorMessage 82
GetErrorNo 81
getformatterobject 79
gethitrule 45, 55
getlog 66
getnohitrule 45, 57
getopt 64
GetRerror 83
getsubscription 59

H

header files 27

I

ISO-8601:1988 standard date notation 15

L

libraries 36
linking to libraries 36
LoadRuleSet 71

M

memory requirements 9
Message Type Management API functions 115
- NNRMgrAddMsg 115, 123, 125
- NNRMgrReadMsg 117, 119, 121
- NNRMsgData 113, 114

Message Type Management APIs 111
- NNRMsg 112

message types 11, 111, 115
MQSeries Integrator
- disk space requirements 9

multi-threaded environment 24

N

NN_CLEAR 154, 156
NNDate 87
NNPermissionData 153
NNR_CLEAR 91
NNRAction 228
NNRActionData 230
NNRActionReadData 231
NNRActionUpdate 233
NNRApp 93
NNRAppData 94
NNRAppUpdate 96
NNRArg 185
NNRArgData 186
NNRArgUpdate 188
NNRExp 175
NNRExpData 176
NNRGetErrorMessage 272
NNRMgrAddAction 234, 247
NNRmgrAddExpression 178
NNRMgrAddMsg 115, 123, 125
NNRMgrAddOption 256, 269
NNRMgrAddRule 135, 143
NNRMgrAddSubscription 203
NNRMgrChangeOwner 163
NNRMgrClose 90
NNRMgrDeleteEntireRule 148
NNRMgrDeleteEntireSubscription 109, 221
NNRMgrDeleteSubscriptionFromRule 219
NNRMgrDuplicateSubscription 105, 214
NNRMgrGetFirstAction 236
NNRMgrGetFirstArgument 190

NNRMgrGetFirstOperator 170
NNRMgrGetFirstOption 258
NNRMgrGetFirstPerm 157
NNRMgrGetFirstRule 139
NNRMgrGetFirstRuleUsingSubs 223
NNRMgrGetFirstSubscription 101, 103, 208
NNRMgrGetNextAction 238
NNRMgrGetNextArgument 192
NNRMgrGetNextOperator 172
NNRMgrGetNextOption 260
NNRMgrGetNextPerm 159
NNRMgrGetNextRule 141
NNRMgrGetNextRuleUsingSubs 225
NNRMgrGetNextSubscription 211
NNRMgrInit 89
NNRMgrReadApp 99
NNRMgrReadExpression 180
NNRMgrReadMsg 117, 119, 121
NNRMgrReadRule 137
NNRMgrReadSubscription 206
NNRMgrResequenceAction 240
NNRMgrResequenceOption 262
NNRMgrUpdateAction 244
NNRMgrUpdateApp 107
NNRmgrUpdateExpression 182
NNRMgrUpdateOption 266
NNRMgrUpdateOwnerPerm 161, 165
NNRMgrUpdatePublicPerm 167
NNRMgrUpdateRule 145
NNRMgrUpdateSubscription 216
NNRMSG 112
NNRMsgData 113, 114
NNROperator 169
NNROption 250
NNROptionData 252
NNROptionReadData 253
NNROptionUpdate 255
NNRRule 95, 127
NNRRuleData 129
NNRRuleReadData 131
NNRRuleUpdate 133
NNRSubs 195
NNRSubsData 197
NNRSubsReadData 199
NNRSubsUpdate 201
NNUserPermissionData 151
non-threaded environment 23

O

Operator Management API functions
- NNRMgrGetFirstOperator 170
- NNRMgrGetNextOperator 172

Operator Management APIs
- NNROperator 169

operators
- & 12
- | 12
- AND 12
- Boolean 12
- date 15
- datetime 15
- OR 12
- Rules 12
- time 15

Option Management API functions
- NNRMgrAddOption 256, 269
- NNRMgrGetFirstOption 258
- NNRMgrGetNextOption 260
- NNRMgrResequenceOption 262
- NNRMgrUpdateOption 266

Option Management APIs 249
- NNROption 250
- NNROptionData 252
- NNROptionReadData 253
- NNROptionUpdate 255

option name-value pairs 43
OPTIONPAIR structures 43
options 20
OR operator 12
Overall Permission Macro
- NN_CLEAR 156

Overview 11

P

Permission API functions 157
- NNRMgrChangeOwner 163
- NNRMgrGetFirstPerm 157
- NNRMgrGetNextPerm 159
- NNRMgrUpdateOwnerPerm 161, 165
- NNRMgrUpdatePublicPerm 167

permission errors 273
permissions
- Rules 19

- Subscription 19

Permissions APIs 151

Permissions Management API functions
- NNPermissionData 153
- NNUserPermissionData 151

Permissions Management API structures 151

R

requirements
- diskspace 9
- memory 9
- MQSeries Integrator disk space 9

Rule Management API functions
- NNRMgrAddRule 135, 143
- NNRMgrDeleteEntireRule 148
- NNRMgrGetFirstRule 139
- NNRMgrGetNextRule 141
- NNRMgrReadRule 137
- NNRMgrUpdateRule 145

Rule Management APIs
- NNRRule 95, 127
- NNRRuleData 129
- NNRRuleReadData 131
- NNRRuleUpdate 133

RULE structure
- gethitrule 45
- getnohitrule 45

Rules 12
- application groups 11
- CreateRulesEngine 47
- DeleteRuleEngine 50
- libraries 36
- linking to libraries 36
- message types 11
- NN_CLEAR 154
- OPTIONPAIR 43
- Overview 11
- RULE structure 45
- SUBSCRIPTION 41
- thread-safe functions 23
- VRule 39
- VRule member functions
  - CreateRulesEngine 47
  - DeleteRuleEngine 50
- VRule supporting functions 46

Rules APIs 39

Rules error codes 273
- client code errors 273
- data processing errors 273
- permission errors 273
- Rules Management data errors 273

Rules error handling 81
- GetErrorMessage 82
- GetErrorNo 81
- GetRerror 83

Rules Management
- NN_CLEAR 154

Rules Management APIs 85, 127
- NNDate 87
- NNRMgrClose 90
- NNRMgrInit 89

Rules Management data errors 273

Rules Management error handling
- NNRGetErrorMessage 272

Rules Management functions 27

Rules Management macros 27
- NNR_CLEAR 91

Rules operators 12

Rules permissions 19

S

standard date notation 15

Subscription Management API functions
- NNRMgrAddSubscription 203
- NNRMgrDeleteEntireSubscription 109, 221
- NNRMgrDeleteSubscriptionFromRule 219
- NNRMgrDuplicateSubscription 105, 214
- NNRMgrGetFirstRuleUsingSubs 223
- NNRMgrGetFirstSubscription 101, 103, 208
- NNRMgrGetNextRuleUsingSubs 225
- NNRMgrGetNextSubscription 211
- NNRMgrReadSubscription 206
- NNRMgrUpdateSubscription 216

Subscription Management APIs 194
- NNRSubs 195
- NNRSubsData 197
- NNRSubsReadData 199
- NNRSubsUpdate 201

Subscription permissions 19

SUBSCRIPTION structures 41

subscriptions 20

T

ThreadCleanup 67
thread-safe functions 23
time operators 15

V

Virtual Rules Engine 39
VRule member functions 27
    CreateRulesEngine 47
    DeleteRuleEngine 50
    eval 52
    getaction 62
    getformatterobject 79
    gethitrule 55
    getlog 66
    getnohitrule 57
    getopt 64
    getsubscription 59
    LoadRuleSet 71
    ThreadCleanup 67
VRule object 39
VRule supporting functions 46

Sending your comments to IBM
MQSeries Integrator
Programming Reference for NEONRules
SC34-5506-01

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 mail, use the Readers' Comment Form
- 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

Readers' Comments
MQSeries Integrator
Programming Reference for NEONRules
SC34-5506-01

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or organization	
Telephone	Email

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States of America
Belgium	France	Israel	New Zealand	Spain	
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP 095)
Hursley Park
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name ______________________
Company or Organization ______________________
Address ______________________
____
EMAIL ______________________
Telephone ______________________

1 Cut along this line

4 Fasten here with adhesive tape

Printed in U.S.A

SC34-5506-01