Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms

New Era of Networks Rules Programming Reference

Version 6.0

Note: **Before using this information, and the product it supports, be sure to read the general information under** *notices* on page 391.

First edition (August 2005)
This edition applies to Rules and Formatter Extension for WebSphere [tm] Message Broker for Multiplatforms, Version 6.0, for IBM ® WebSphere Message Broker 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

This chapter includes the following information:

- *About this Document*
- *Documentation Set*

About this Document

This programming reference provides descriptions and examples for each function in NEONRules and NEONRules Management APIs. This document is divided into two main sections: NEONRules APIs and NEONRules Management APIs.

- Chapter 1, *Introduction*, provides a brief description of NEONRules and the documentation set and documentation conventions.
- Chapter 2, *Overview*, describes NEONRules components, rules naming conventions, APIs, header files, and libraries.
- Chapter 3, NEONRules *APIs*, provides class and type definitions and contains the NEONRules APIs.
- Chapter 4, NEONRules *Management APIs*, provides rules management API structures, rules management APIs and macros.
- Chapter 5, *Rules Error Messages*, contains a list of rules error messages.
- Appendix A, *Operator Types*, describes the available operator types for use in rules expressions.

Documentation Set

The Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms documentation set includes:

- ***System Management Guide***
- NEONFormatter ***Programming Reference***
- NEONRules ***Programming Reference***
- ***Application Development Guide***
- Rules, Formatter, and Visual Tester online help
- Installation Readme

Document Conventions

The following document conventions are used in this guide.

Text	Convention	Example
code	courier	<user ID> <password>
command line display	courier	`The message successfully parsed.`
command line entry	courier bold	**`NNFAD-t`**
command line prompt	courier	`Enter the input file name:`
path	regular	ora/bin (UNIX) ora\bin (NT)
book names	bold, italic	***Installation Guide***
chapter and section names	italic	*NT Installation*

Chapter 1
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

The NEONRules components are:

- Application groups
- Message types
- Rules

An application group is a logical grouping used to organize rules. For example, a company can divide rules into groups by projects or split projects into logical sub-groups.

A message type defines the layout of a string of data. Each application group can contain several message types and a message type can be in more than one application group. When using NEONFormatter, the message type is the same as either the input format name or the user-defined NEONRules message type. Message types are defined either in NEONFormatter or in NEONRules.

A rule contains specific actions to be processed by the application if the rule evaluates to true against a message. These actions can be thought of as computer commands and the associated parameters required to execute the rule.

Rules Naming Conventions

When you are creating names for rule components, use the following conventions:

- Create unique, descriptive component names that are easy to distinguish from one another.
- Do not use case differences to distinguish component names. Some databases do not distinguish case and would interpret both components ITEM1 and Item1 as having the same name. In this case, each matching component would conflict during importing.
- Do not use the component name, NONE. It is reserved for another use.
- Do not use single quotes, double quotes, or spaces in component names. These may cause database problems.
- Do not exceed the maximum of 120 characters when creating component names. If you exceed 120 characters, a message box appears requiring a change.

  The maximum number of characters for double byte is 60.

APIs and Header Files

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

Use NEONRules 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 NEONRules 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	(NNSesDBBase *Session)
VRule *	CreateRulesEngine	(DbmsSession* Session, int alert=1, char *logfile=NULL)
VRule *	CreateRulesEngine	(NNSesDBBase* 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)
int	eval	(char *AppName, char *MsgName, char NNFieldValueContainer*, pFVList)
Formatter	getformatterobject	None
RULE*	gethitrule	None
RULE*	gethitrule	None
char*	getlog	None
SUBSCRIPTION*	getsubscription	None
OPTIONPAIR*	getopt	None
int	LoadRuleSet	(char *AppGrp, char*MsgType, int LoadNow=0)
int	LoadRuleComponent	(char *AppGrp, char*MsgType, NNRComponentTypes ComponentType, char* ComponentType, int LoadNow=0)
int	populatesubsriptionlist	(RulesSubscriptionList& subsContainer)
void	ThreadCleanup	None

SubscriptionList, ActionList, OptionList Functions

Return Type	Function	Arguments
ThisType	&operator=	const ThisType& right
NNSY_NAMESPACE_SF	append_back	(RulesSubscription* pSubscription)
NNSY_NAMESPACE_SF	append_front	(RulesSubscription* pSubscription)
NNSY_NAMESPACE_SF	clear	None
void	createOwnCopyOfData	None
NNSY_NAMESPACE_SF	deleteSubscription	(int subscriptionId)
RulesSubscription	getFirst	None
RulesSubscriptionList	getNewSubscription	None
RulesSubscription	getNext	None
NNSY_NAMESPACE_SF	insert	(RulesSubscription* pSubscription)
NNSY_NAMESPACE_SF	insert	(RulesSubscriptionList* pSubscription)
RulesSubscriptionList	newCopy	None
NNSY_NAMESPACE_SF	push_front	(RulesSubscription* pSubscription)
NNSY_NAMESPACE_SF	push_back	(RulesSubscription* pSubscription)
int	size	None

Subscription, Action, Option Functions

Return Type	Function	Arguments
ThisType	&operator=	const ThisType& right
NNSY_NAMESPACE_SF	compareById	(int subscriptionId)
void	createOwnCopyOfData	None
RulesSubscription	geActionList	None
NNSY_NAMESPACE_SF	getId	(int& subscriptionId)
NNSY_NAMESPACE_SF	getName	(const STL_STRING& subscriptionName)
RulesSubscriptionList	newCopy	None
NNSY_NAMESPACE_SF	set_Id	(int& subscriptionId)
NNSY_NAMESPACE_SF	set_Name	(const STL_STRING& subscriptionName)

NNFieldValueContainer Functions

Return Type	Function	Arguments
char*	GetFieldString	(char* name, int instance=-1)
int	GetFieldCount	(char* name)

NNNameValueList Functions

Return Type	Function	Arguments
int	Add	(const NNName *pName, const NNValue *pValue)
int	Read	(const NNName *pName, const NNValue *pValue, int instance)
int	Update	(const NNName *pName, const NNValue *pValue, int instance)
int	ClearAll	None
int	getFirst	(const NNName *pName, const NNValue *pValue)
int	getNext	(const NNName *pName, const NNValue *pValue)

NNName Functions

Return Type	Function	Arguments
int	set	(char* name)
int	set	(char* name, int length)
bool	operator<	(const NNName& name1, const NNName& name2)
bool	operator==	(const NNName& name1, const NNName& name2)
void	operator=	(const NNName& name1)
bool	isEmpty	None
char*	GetString	None
int	GetLength	None

NNValue Functions

Return Type	Function	Arguments
int	set	(char* value)
int	set	(char* value, int length)
bool	operator<	(const NNValue& value1, const NNValue& value1)
bool	operator==	(const NNValue& value1, const NNValue& value1)
void	operator=	(const NNValue& value1)
bool	isEmpty	None
char*	GetString	None
int	GetLength	None

Rules Error Handling Functions

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

Application Group Management Functions

Return Type	Function	Arguments
const long	NNRMgrAddApp	(NNRMgr *pMgr, const NNRApp *pRApp, const NNRAppData *pRAppData)
const long	NNRMgrReadApp	(NNRMgr *pMgr, const NNRApp *pRApp, NNRAppData *const pRAppData)
const long	NNRMgrGetFirstApp	(NNRMgr *pMgr, const NNRAppReadData *const pRAppData)
const long	NNRMgrGetNextApp	(NNRMgr *pMgr, const NNRAppReadData *const pRAppData)
const long	NNRMgrDuplicateApp	(NNRMgr *pMgr, const NNRApp* pRApp, *const char* NewAppName)
const long	NNRMgrUpdateApp	(NNRMgr *pMgr, const NNRApp* pRApp, const NNRAppUpdate *pRAppUpdate)
const long	NNRMgrDeleteEntireApp	(NNRMgr *pMgr, const NNRApp* pRApp)

Message Type Management Functions

Return Type	Function	Arguments
const long	NNRMgrAddMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg, const NNRMstgData *pRMsgData)
const long	NNRMgrDeleteEntire Msg	(NNRMgr *pMgr, const NNRMsg* pRMsg)
const long	NNRMgrDuplicateMsg	(NNRMgr *pMgr, const NNRMsg* pRMsg, const char *NewAppName)
const long	NNRMgrGetFirstMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg, NNRMsgReadData *const pRMsgData)
const long	NNRMgrGetNextMsg	(NNRMgr *pMgr, const NNRMsgReadData *const pRMsgData)
const long	NNRMgrReadMsg	(NNRMgr *pMgr, const NNRMsg *pRMsg, NNRMsgData *const pRMsgData)
const long	NNRMgrUpdateMsgN ame	(NNRMgr *pMgr, const char *OldMsgName, const char *NewMsgName)

Rules Management Functions

Return Type	Function	Arguments
NNRMgr *	NNRMgrInit	(DbmsSession *session)
void	*NNRMgrClose* on page 164	(NNRMgr *pMgr)
N/A	*NNR_CLEAR* on page 165	(_p)
N/A	NN_CLEAR	(_p)
const long	NNRMgrAddRule	(NNRMgr *pMgr, const NNRRule *pRRule, const NNRRuleData *pRRuleData)
const long	*NNRMgrReadRule* on page 212	(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)

Permissions Functions

Return Type	Function	Arguments
const long	NNRMgrGetFirst Perm	(NNRMgr *pRMgr, const NNRComponent * pRComponent, NNUserPermissionData const * pPermissionData)
const long	NNRMgrGetNext Perm	(NNRMgr *pRMgr, NNUserPermissionData const * pPermissionData)
const long	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)

Operator Management Functions

Return Type	Function	Arguments
const long	NNRMgrGetFirst Operator	(NNRMgr *pMgr, NNROperator * const pOperator)
const long	NNRMgrGetNext Operator	(NNRMgr *pMgr, NNROperator * const pOperator)

Expression Management Functions

Return Type	Function	Arguments
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)

Argument Management Functions

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)

Subscription, Action, Option Management Functions

Return Type	Function	Arguments
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)
const long	NNRMgrGetFirst RuleUsingSubs	(NNRMgr *pMgr, const NNRSubs *pRSubs, char* const pRuleName)

Return Type	Function	Arguments
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	NNRMgrResequenceAction	(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)
const long	NNRMgrGetFirst Option	(NNRMgr *pMgr, const NNROption * pROption, NNROptionReadData * const pROptionData)

Return Type	Function	Arguments
const long	NNRMgrGetNext Option	(NNRMgr *pMgr, NNROptionReadData * const pROptionData)
const long	NNRMgrResequenceO ption	(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

Shared libraries are archived collections of object files. The following is the path to the libraries that must be linked with the application object files:

In UNIX, the libraries are in {installroot}/bin.

In Windows, the shared libraries and DLLs are in {installroot}\bin. The libraries needed to compile custom code are in {installroot}\lib.

Refer to the example makefiles for more library information.

Note:
Library file extensions are .sl for HP-UX, .dll for Windows, and .so for AIX.

WARNING!
Do not move the libraries. The executables search for them in a specific directory or folder. If you move or delete the libraries, the executables are rendered useless.

Chapter 2
NEONRules APIs

This chapter includes the following information:

- *Class/Type Definitions*
- *VRule Engine APIs*
- *Subscription, Action, Option APIs*
- *Evaluation Field Value Containers*

Class/Type Definitions

VRule Class

This class provides a standard interface for handling NEONRules API calls and allows the user to perform all rule evaluation and subscription retrieval.

See vrule.h in the /include directory.

RulesSubscriptionList, RulesActionList, & RulesOptionList Classes

The **RulesSubscriptionList** class allows the user to create a RulesSubscriptionList object. This object can then be passed in the VRule::populatesubscriptionlist member function to pull the subscriptions that hit for the active message. The RulesSubscriptionList contains instances of RulesSubscriptions.

The **RulesActionList** class allows the user to pull the actions that are valid for a given subscription. An instance of the RulesSubscription class contains a RulesActionList object which contains many instances of RulesActions.

The **RulesOptionList** class allows the user to pull the options that are valid for a given subscription. An instance of the RulesSubscription class contains a RulesOptionList object which contains many instances of RulesOptions.

RulesSubscription, RulesAction, and RulesOption Classes

The **RulesSubscription** class allows the user to create a RulesSubscription object. These objects are generally found inside the RulesSubscriptionLists. The RulesSubscription is used to traverse the list of subscriptions retrieved from the VRule::populatesubscriptionlist method.

The **RulesAction** class allows the user to create a RulesAction object. These objects are generally found inside the RulesActionLists. The RulesAction is used to traverse the list of actions retrieved from the RulesSubscription::getActionList method.

The **RulesOption** class allows the user to create a RulesOption object. These objects are generally found inside the RulesOptionLists. The RulesOption is used to traverse the list of options retrieved from the RulesAction::getOptionList method.

NNFieldValueContainer Class

The NNFieldValueContainer class is the base class for any class that contains field values that can be retrieved by name. Formatter and NNNameValueList classes inherit from this class. Users can input their own object containing field values into the VRule::eval() API as long as the object inherits from this NNFieldValueContainer base class and has the correct member functions.

NNValueValueList Classes

The **NNNameValueList** class is used to identify field values that can be retrieved by name. The NNNameValueList contains a list of field name and value pairs from the NNName and NNValue classes where the name is up to 120 characters and the value can be of any length for rules evaluation.

The **NNName** class is used for some of the NNNameValueList methods to identify the object from which field name information is retrieved. This class enables retrieval of field or object name information without using NEONFormatter to parse the information.

The **NNValue** class is used for some of the NNNameValueList methods to identify the value information to retrieve. This class enables retrieval of field or object value information without using NEONFormatter to parse the information.

VRule Engine APIs

To use NEONRules APIs, you must include the following header files:

- dbtypes.h & ses.h OR NNOT.h & NNSesDBBase.h
- rerror.h
- ruleuser.h
- vrule.h
- RulesSubscriptionList.h
- RulesSubscription.h
- RulesActionList.h
- RulesAction.h
- RulesOptionList.h
- RulesOption.h

Note:
THREAD SAFETY: For multithreading, you must also link with the appropriate thread library matching the NEONRules release. For example, link with the thread library for UI threads and pthread for POSIX threads.

A VRule object is a Virtual Rules Engine instance. This class provides a standard interface for handling NEONRules 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().

VRule.h is defined as follows:

```
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 * AppName, char * MsgName,
                       NNFieldValueContainer *pFBContainer,
                       int log=0) = 0;
    virtual SUBSCRIPTION * getsubscription() = 0;
    virtual int populatesubscriptionlist(RulesSubscriptionList&
                           subsContainer)
    virtual OPTIONPAIR * getopt() = 0;
    virtual RULE * gethitrule() = 0;
    virtual RULE * getnohitrule() = 0;
    virtual char * GetErrorMessage() = 0;
    virtual void ThreadCleanup() = 0;
    virtual int LoadRuleSet(char* AppName, char * MsgName,
                           int LoadNow = 0) = 0;
   virtual int LoadRuleComponent(char* AppGrp, char * MsgType,
                           NNRComponentTypes ComponentType,
                           char * ComponentName,
                           int LoadNow = 0) = 0;
    virtual Formatter *getFormatterobject() = 0;
};
```

VRule Structures

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

Remarks

The action and SubName members point to memory inside the VRule object. Do not modify their values.

It is recommended that programmers use the new RulesSubscription classes instead of the SUBSCRIPTION and OPTIONPAIR structures.

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

Remarks

The Name and Value members point to memory inside the VRule object. Do not modify their values.

Example

The following code segment illustrates walking through a list of options. 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

Remarks

The RuleName member points to memory inside the VRule object. Do not modify their values.

Example

The following code fragment describes how to walk through both 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

CreateRulesEngine

Syntax 1

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

Description

CreateRulesEngine() creates a VRule object for the application session provided in the session parameter.

Parameters

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

Syntax 2

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

Description

CreateRulesEngine() creates a VRule object for the session provided in the session parameter.

Parameters

Name	Type	Input/ Output	Description
Session	NNSesDBBase*	Input	Name of the open session.

Syntax 3

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

Description

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

Parameters

Name	Type	Input/ Output	Description
Session	DbmsSession *	Input	Name of the open Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms session. See OpenDbmsSession() in the ***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. Only valid if alert is True (1). Defaults to no file (NULL).

Syntax 4

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

Description

CreateRulesEngine creates a VRule object for the NEONRules and NEONFormatter session provided in the session parameter and enables the user to specify whether alerts should be sent to the log file.

Parameters

Name	Type	Input/ Output	Description
Session	NNSesDBBase*	Input	Name of the open Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms session. See OpenNNSesDBBase() in the ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms 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. 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("MySesName", 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("MySesName", 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("MySesName", 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

There are two uses of the VRule::eval method. One is for use when the evaluation is based on information received from NEONFormatter and the other is for use when evaluating data derived from a NNNameValueContainer.

Syntax 1

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

Description

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.

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.

Name	Type	Input/ Output	Description
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.
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.

Syntax 2

```
int VRule::eval(char* AppName,
                char* MsgName,
                char* NNFieldValueContainer* pFVList);
```

Description

This version of eval takes in a NNFieldValueContainer pointer that is used to retrieve values based on names. New Era of Networks provides the NNNameValueList and Formatter classes which are NNFieldValueContainers. Programmers can create their own class derived from the NNNameValueContainer.

NEONFormatter is not used if a NNNameValueList is input. In that case, rules are defined in the same way, but message type's EvalType is

RulesMessageType and the field names are not defined in NEONFormatter, but supplied in a separate list of names.

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, the 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.
pFVList	NNFieldValueContainer*	Input	A pointer to the NNFieldValueContainer object to be used to retrieve values.

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.

Return Value

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

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 are read into cache. Subsequent calls to eval() do not reload the data unless LoadRuleSet() or LoadRuleComponent() were called previously with LoadNow set to FALSE. Modifications to the data are only reflected if one of the Load APIs is called prior to the eval() API. See*LoadRuleSet* on page 59 and*LoadRuleComponent* on page 54 for more information.

Example

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

See Also

CreateRulesEngine

DeleteRuleEngine

getsubscription

gethitrule

getnohitrule

GetErrorNo

GetRerror

GetErrorMessage

LoadRuleSet

LoadRuleComponent

NNFieldValueContainer

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.

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;
           }
```

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.

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 both 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

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 follows a call to CreateRulesEngine() but precedes 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.

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 both 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.

Note:
By using populatesubscriptionlist method instead of getsubscription and getopt, all eval data results are retrieved at one time, releasing VRule so that you can apply a re-evaluation.

Syntax

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

Parameters

None

Remarks

getsubscription() should be called after the eval() function, which follows 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.

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.

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

getopt

populatescriptionlist

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.

Note:
By using populatesubscriptionlist method instead of getsubscription and getopt, all eval data results are retrieved at one time, releasing VRule so that you can apply a re-evaluation.

Syntax

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

Parameters

None

Remarks

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

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

getsubscription

populatescriptionlist

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 NEONRules 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.

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 group. 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 type. 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 226 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 flagged 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 use the new Rules data
```

Note:
The LoadRuleComponent API returns a value of 2 if the NEONRules 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

LoadRuleSet

Using the application group and message type, LoadRuleSet() sets a flag indicating that the system should clear any current rule set information and load the rule set indicated by the AppName and MsgName parameters.

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, 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.

Note:
When LoadRuleSet is run, pointers to rule, subscription, and option information are overwritten. To maintain the pointers and their associated information, make a copy of the rule, subscription, and option information before LoadRuleSet is run.

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 use the new Rules data
```

Note:
The LoadRuleSet API returns a value of 2 if the NEONRules Engine instance has never evaluated a message using the specified application group / message name pair and the LoadNow is zero. 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

populatesubscriptionlist

The populatesubscriptionlist function allows a user to retrieve a subscription list from the NEONRules engine. The method pulls the subscriptions, actions, and options that hit for the active message. The method first pulls a subscription from the rules object. If a subscription exists, it retrieves the first action that applies. If the action exists, it loads all options pertaining to the action and stores them in an option list. When the list is full, options for the next action are added to the list. After the list of actions for the subscription is full, the method retrieves the actions and options for the next subscription. The populatesubscriptionlist method proceeds to load subscriptions, actions, and options until there are no more in the rules object. Upon completion of searching all subscriptions that hit and retrieving the applicable actions and options, the full subscription list is returned to the user via the parameter passed in the method call.

The populatesubscriptionlist should be used instead of the getsubscription and getopt methods since it retrieves all eval results once, thereby releasing VRule so that you can call eval again to apply a re-evaluation on the retrieved data.

Syntax

```
int VRule::populatescriptionlist(RulesSubscriptionList&
subsContainer)
```

Parameters

Name	Type	Input/ Output	Description
subsContainer	RulesSubscriptionList&	input output	Subscription list from the Rules Object.

Remarks

The user must create a RulesSubscriptionList before this call. This parameter is passed by reference and the method populates the data.

Return Value

Returns 1 if list is successfully populated and 0 if it is not.

See Also

getsubscription

getopt

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. 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.

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

Subscription, Action, Option APIs

The subscription classes enables the VRule::populatesubscriptionlist method to retrieve subscriptions, actions, and options for rules. Members of the classes enable creation and manipulation of the data within each class.

There are six classes covered under this API section. Three are list classes and three are object classes. The methods for each of the list classes are the same and the methods for each object classes are the same. A description of each class is provided below. Detailed information on the methods are only provided for the RulesSubscriptionList class and RulesSubscription class. Substitute "Subscription" for "Action" or "Option" to use the method to pull the appropriate action or option information.

List Classes

The **RulesSubscriptionList** class allows the user to create a RulesSubscriptionList object. This object can then be passed in the VRule::populatesubscriptionlist member function to pull the subscriptions that hit for the active message. The RulesSubscriptionList contains instances of RulesSubscriptions.

The **RulesActionList** class allows the user to pull the actions that are valid for a given subscription. An instance of the RulesSubscription class contains a RulesActionList object which contains many instances of RulesActions.

The **RulesOptionList** class allows the user to pull the options that are valid for a given subscription. An instance of the RulesSubscription class contains a RulesOptionList object which contains many instances of RulesOptions.

Object Classes

The **RulesSubscription** class allows the user to create a RulesSubscription object. These objects are generally found inside the RulesSubscriptionLists. The RulesSubscription is used to traverse the list of subscriptions retrieved from the VRule::populatesubscriptionlist method.

The **RulesAction** class allows the user to create a RulesAction object. These objects are generally found inside the RulesActionLists. The RulesAction is

used to traverse the list of actions retrieved from the RulesSubscription::getActionList method.

The **RulesOption** class allows the user to create a RulesOption object. These objects are generally found inside the RulesOptionLists. The RulesOption is used to traverse the list of options retrieved from the RulesAction::getOptionList method.

RulesSubscriptionList Member Functions

The RulesSubscriptionList class allows the user to create a RulesSubscriptionList object. This object can then be passed in the VRule::populatesubscriptionlist member function to pull the subscriptions, actions, and options that hit for an active message.

RulesSubscriptionList Constructor

This constructor allows the user to create an instance of the RulesSubscriptionList object.

Syntax

```
RulesSubscriptionList::RulesSubscriptionList()
```

Parameters

N/A

Return Value

N/A

RulesSubscriptionList Destructor

This destructor deallocates the memory for the internal data object if it is not being shared.

Syntax

```
RulesSubscriptionList::~RulesSubscriptionList()
```

Parameters

N/A

Return Value

N/A

RulesSubscriptionList Copy Constructor

This copy constructor allows the user to get a shared copy of the RulesSubscriptionList being passed. This method makes a shallow reference-counted copy of RulesSubscriptionList data.With the pointer to the internal data, the calling object references the data rather than making a separate copy of it. This results in saving memory.

The newCopy method should be used to get an unshared copy.

Syntax

```
RulesSubscriptionList::RulesSubscriptionList(const ThisType&
orig)
```

Parameters

Name	Type	Input/ Output	Description
orig	const ThisType&	Input	Pointer for the object to be copied.

Return Value

None. If the calling object is passed a NULL object, the new object remains NULL.

&operator= Assignment Operator

This member function makes a shallow reference-counted copy of RulesSubscriptionList data.With the pointer to the internal data, the calling object references the data rather making having a separate copy of it. This results in saving memory. The call returns an object of type RulesSubscriptionList containing the new pointer to ThisType's data. This function does not allow passing a copy of itself as a parameter.

Syntax

```
ThisType RulesSubscriptionList::&operator=(const ThisType&
right)
```

Parameters

Name	Type	Input/ Output	Description
right	const ThisType&	Input	Pointer for the calling object to be copied.

Return Value

RulesSubscriptionList object pointing to the shared data.

append_back

The append_back method enables adding a RulesSubscription object at the back of the RulesSubscriptionList object. It inserts a user's own RulesSubscriptionList onto the back of the calling RulesSubscriptionList.

The parameter must be a non-NULL RulesSubscription object. The return value is of type e_SF representing SF_Success when insert completes successfully or SF_Failure when the insert fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::append_back
(RulesSubscription* pSubscription)
```

Parameters

Name	Type	Input/ Output	Description
pSubscription	RulesSubscription*	Input	non-NULL RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

append_front

The append_front method enables adding a RulesSubscription object to the front of the RulesSubscriptionList object. It inserts a user's RulesSubscriptionList into the front of the calling RulesSubscriptionList.

The parameter must be a non-NULL RulesSubscription object. The return value is of type e_SF representing SF_Success when insert completes successfully or SF_Failure when the insert fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::append_front
(RulesSubscription* pSubscription)
```

Parameters

Name	Type	Input/ Output	Description
pSubscription	RulesSubscription*	Input	non-NULL RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

Clear

The Clear method removes all objects from the RulesSubscriptionList. The function clears the current list of RulesSubscriptions leaving the calling RulesSubscriptionList empty of all objects. The RulesSubscription objects in the list are deleted. Their internal data objects are deleted if they are not shared by other RulesSubscriptions.

The return value is of type e_SF representing SF_Success when clear completes successfully or SF_Failure when the clear fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::clear()
```

Parameters

N/A

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

createOwnCopyOfData

This method is typically used internally to create a new RulesSubscriptionList for a user. If the internal data is shared, it creates a new copy of internal data for the calling object.

Syntax

```
void RulesSubscriptionList::createOwnCopyOfData()
```

Parameters

N/A

Return Value

None.

DeleteSubscription

The DeleteSubscripiton method deletes the object in the list with the ID number provided in the parameter list. This function deletes the subscription from the calling RulesSubscriptionList object. It uses the integer that gets passed as a parameter to find the appropriate RulesSubscription. The RulesSubscription's internal data object is deallocated if it is not being shared by another RulesSubscription.

The return value is of type e_SF representing SF_Success when the deletion completes successfully or SF_Failure when the deletion fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF
RulesSubscriptionList::deleteSubscription (int subscriptionId)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionId	int	Input	RulesSubscription object Id.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful deletions or SF_Failure to indicate failed deletion attempts. Returns SF_Failure if the Subscription Id is not found.

getFirst

The getFirst method returns the first item in the RulesSubscriptionList.

Syntax

```
virtual RulesSubscription* RulesSubscriptionList::getFirst()
```

Parameters

N/A

Return Value

RulesSubscription pointer or NULL pointer for an empty list.

getNewSubscription

The getNewSubscription method is used to get a new object in the RulesSubscriptionList. This member function creates a subscription in the calling RulesSubscriptionList, sets the Id of the new Subscription, and gives a pointer to the new Subscription to the user. The method fails when the new subscription is NULL or the Id cannot be set.

Syntax

```
virtual RulesSubscriptionList*
RulesSubscriptionList::getNewSubscription()
```

Parameters

N/A

Return Value

RulesSubscriptionList subscription pointer

getNext

The getNext method retrieves the current object pointed to by the iterator and moves the iterator to the next object in the RulesSubscriptionList.

Syntax

```
virtual RulesSubscription* RulesSubscriptionList::getNext()
```

Parameters

N/A

Return Value

RulesSubscription pointer or NULL pointer for an empty list and when it reaches the end of the list.

insert (subscription)

This insert method enables adding an object before the current iterator position in the RulesSubscriptionList object. When traversing the list with the getFirst or getNext iterators, this insert method can be used to place a subscription before the object retrieved in either "get" call. The parameter must be a non-NULL RulesSubscription object. The return value is of type e_SF representing SF_Success when the insert completes successfully or SF_Failure when the insert fails.

If the current iterator position is equal to the beginning, use the push_front method to insert a RulesSubscription before the first object.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::insert
(RulesSubscription* pSubscription)
```

Parameters

Name	Type	Input/ Output	Description
pSubscription	RulesSubscription*	Input	non-NULL RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

insert (list)

This insert method inserts a user's RulesSubscriptionList into the calling RulesSubscriptionList based on the current iterator position. When traversing the list with the getFirst or getNext iterators, this insert method can be used to place the list before the current RulesSubscription.

The parameter must be a non-NULL RulesSubscriptionList object. The return value is of type e_SF representing SF_Success when insert completes successfully or SF_Failure when the insert fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::insert
(RulesSubscriptionList* pSubscriptionList)
```

Parameters

Name	Type	Input/ Output	Description
pSubscriptionList	RulesSubscriptionList*	Input	non-NULL RulesSubscriptionList object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

newCopy

The newCopy constructor allows the user to get an unshared deep copy of the RulesSubscriptionList being passed. Use this method to create a personally owned object of type RulesSubscriptionList. The calling object creates its own copy of internal data for itself.

Syntax

```
RulesSubscriptionList* RulesSubscriptionList::newCopy()
```

Parameters

N / A

Return Value

A pointer to the new copy of RulesSubscriptionList object with its own date.

push_front

The push_front method enables adding an object to the front of the RulesSubscriptionList object. It inserts a subscription into the front of the subscription list before any existing objects.

The parameter must be a non-NULL RulesSubscription object. The return value is of type e_SF representing SF_Success when insert completes successfully or SF_Failure when the insert fails

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::push_front
(RulesSubscription* pSubscription)
```

Parameters

Name	Type	Input/ Output	Description
pSubscription	RulesSubscription*	Input	non-NULL RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

push_back

The push_back method enables adding an object to the end of the RulesSubscriptionList object. The subscription is inserted after the "end" iterator pointer position after any existing objects.

The parameter must be a non-NULL RulesSubscription object. The return value is of type e_SF representing SF_Success when insert completes successfully or SF_Failure when the insert fails.

Syntax

```
virtual NNSY_NAMESPACE e_SF RulesSubscriptionList::push_back
(RulesSubscription* pSubscription)
```

Parameters

Name	Type	Input/ Output	Description
pSubscription	RulesSubscription*	Input	non-NULL RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

size

The size method returns the number of objects in the RulesSubscriptionList.

Syntax

```
virtual int RulesSubscriptionList::size()
```

Parameters

N/A

Return Value

Number of objects contained in the calling RulesSubscriptionList.

RulesSubscription Member Functions

The RulesSubscription class allows the user to create a RulesSubscription object. These objects are generally found inside the RulesSubscriptionLists. The RulesSubscription is used to traverse the list retrieved from the VRule::populatesubscriptionlist function.

See RulesSubscription.h

RulesSubscription Constructor

This constructor allows the user to create an instance of the RulesSubscription object.

Syntax

```
RulesSubscription::RulesSubscription()
```

Parameters

N / A

Return Value

N / A

RulesSubscription Destructor

This destructor deallocates the memory for the internal data object if it is not being shared.

Syntax

```
RulesSubscription::~RulesSubscription()
```

Parameters

N/A

Return Value

N/A

RulesSubscription Copy Constructor

This copy constructor allows the user to get a shared copy of the RulesSubscription being passed. The newCopy method should be used to get an unshared copy.

Syntax

```
RulesSubscription::RulesSubscription(const ThisType& orig)
```

Parameters

Name	Type	Input/ Output	Description
orig	const ThisType&	Input	Pointer for the object to be copied.

Return Value

None. If the calling object is passed a NULL object, the object remains NULL.

&operator= Assignment Operator

This member function makes a shallow reference-counted copy of RulesSubscription data.With the pointer to the internal data, the calling object references the data rather than making a separate copy of it. This results in saving memory.

The call returns an object of type RulesSubscription containing the new pointer to ThisType's data. This function does not allow passing a copy of itself as a parameter.

Syntax

```
ThisType RulesSubscription::&operator=(const ThisType& right)
```

Parameters

Name	Type	Input/ Output	Description
right	const ThisType&	Input	Pointer for the calling object to the internal data.

Return Value

RulesSubscription object pointing to the shared data.

compareById

The compareById method compares this subscription id with the id in the int parameter.

The parameter must be a non-negative integer. The return value is of type e_SF representing SF_Success when the Ids are equal or SF_Failure when the the Ids are not the same.

Syntax

```
NNSY_NAMESPACE e_SF RulesSubscription::compareById(int
subscriptionId)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionId	int	Input	Id of the RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate equality or SF_Failure to indicate inequality. Returns SF_Failure if NULL.

createOwnCopyOfData

This method is typically used internally to create a new RulesSubscription for a user. If the internal data is shared, it creates a new copy of internal data for the calling object.

Syntax

```
void RulesSubscription::createOwnCopyOfData()
```

Parameters

N/A

Return Value

None.

getActionList

The getActionList method retrieves the current action list to the user.

Syntax

```
RulesActionList* RulesSubscription::getActionList()
```

Parameters

N/A

Return Value

```
A pointer to the RulesActionList for this RulesSubsctiption.
```

getId

The getId method retrieves the subscriptionId via the contents of the int parameter. It fails when the object's Id is invalid (empty).

The parameter is set by this method. The return value is of type e_SF representing SF_Success when the retrieval completes successfully or SF_Failure when the retrieval fails.

Syntax

```
NNSY_NAMESPACE e_SF RulesSubscription::getId(int&
subscriptionId)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionId	int&	Output	Id of the RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

getName

The getName method retrieves the subscription name via the contents of the STL_STRING parameter. It fails when the object's value is invalid (empty).

The parameter is set by this method. The return value is of type e_SF representing SF_Success when the retrieval completes successfully or SF_Failure when the retrieval fails.

Syntax

```
NNSY_NAMESPACE e_SF RulesSubscription::getName(const
STL_STRING& subscriptionName)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionName	const STL_STRING&	Output	name of the RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

newCopy

The newCopy constructor allows the user to get an unshared deep copy of the RulesSubscription being passed. Use this method to create a personally owned object of type RulesSubscription. The calling object creates its own copy of internal data for itself.

Syntax

```
RulesSubscriptionList* RulesSubscription::newCopy()
```

Parameters

N/A

Return Value

A pointer to the new copy of the RulesSubscription object with its own data.

Example

```
RulesSubscriptionList yourRulesSubscriptionList;
//populate this list...
Rules->populatesubscriptionlist(yourRulesSubscriptionList);
RulesSubscription *pYourRulesSubscription =
          yourSubscriptionList.getFirst();
RulesSubscription *pMyRulesSubscription;
pMyRulesSubscription = pYourRulesSubscriptionList->newCopy();
```

setId

The setId method sets the subscriptionId to the contents of the int parameter. It fails when the parameter is invalid (empty).

The parameter must be a non-negative integer. The return value is of type e_SF representing SF_Success when the update completes successfully or SF_Failure when the update fails.

Syntax

```
NNSY_NAMESPACE e_SF RulesSubscription::setId(int&
subscriptionId)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionId	int&	Input	Id of the RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

setName

The setName method sets the subscription name to the contents of the STL_STRING parameter. It fails when the parameter is invalid (empty).

The parameter must be a non-empty string object. The return value is of type e_SF representing SF_Success when the update completes successfully or SF_Failure when the update fails.

Syntax

```
NNSY_NAMESPACE e_SF RulesSubscription::setName(const
STL_STRING& subscriptionName)
```

Parameters

Name	Type	Input/ Output	Description
subscriptionName	const STL_STRING	Input	name of the RulesSubscription object.

Return Value

Type e_SF return type, using the NNSY_NAMESPACE, to return SF_Success to indicate successful inserts or SF_Failure to indicate failed insert attempts. Returns SF_Failure if NULL.

Subscription, Action, Option Class Usage

The following information provides an example of using these classes. To view a full code example of the previous description, please see lines 406 to 506 of ruletest.cpp.

Populate Subscription List

After performing a Rules Evaluation where some Rules evaluate to true and some subscriptions need to be performed, use the VRule method populatesubscriptionlist to get a complete list of all of the subscriptions that have hit for the active message.

The method needs a copy of a RulesSubscriptionList with its own memory. The object should then be passed to the method as a parameter:

```
RulesSubscriptionList *pSubscriptionList =
newRulesSubscriptionList;
rules->populatesubscriptionlist( *pSubscriptionList )
```

Traverse the Subscription List

To begin traversing the list, use the RulesSubscription::getFirst method to obtain the first RulesSubscription from the pSubscriptionList:

```
RulesSubscription *pSubscription = NULL;
pSubscription = pSubscriptionList->getFirst( );
```

The RulesSubscription ID number and the name can now be retrieved for use using the getId and getName member functions of the RulesSubsciption class. Create an integer for the subscription ID and pass the variable to the method as a parameter. The function writes the result to your integer. For the name, create an STL_STRING and pass the variable to the method as a parameter. The function writes the result to your string:

```
int subscriptionId;
STL_STRING subscriptionName;
pSubscription->getId(subscriptionId);
pSubscription->getName(subscriptionName);
```

If a RulesSubscription exists in the list, enter a loop to retrieve all of the actions that may be in the subscription.

Get and Traverse Subscription's Actions

For each subscription in the list, ask each subscription for its RulesActionList:

```
RulesActionList = *pActionList = NULL;
pActionList = pSubscription->getActionList( );
```

To begin traversing the list, create a RulesAction with its own memory and use the RulesActionList::getFirst method to obtain the first RulesAction from the pActionList;

```
RulesAction = *pAction = NULL;
pAction = pActionList->getFirst( );
```

The RulesAction ID number and the name can now be retrieved for use using the getId and getName member functions of the RulesAction class. Create an integer for the subscription ID and pass the variable to the method as a parameter. The function writes the result to your integer. For the name, create an STL_STRING and pass the variable to the method as a parameter. The function writes the result to your string.

```
int actionId;
pAction->getId(actionId);
STL_STRING actionName;
pAction->getName(actionName);
```

If a RulesAction exists in the list, enter a loop to retrieve all of the options in the action.

Get and Traverse Subscription's Options

Then for each action, ask the action for its RulesOptionList:

```
RulesOptionList *pOptionList == NULL;
pOptionList = pAction->getOptionList( );
```

To begin traversing the list, use the RulesOptionList::getFirst method to obtain the first RulesOption from the pOptionList;

```
RulesOption *pOption == NULL;
pOption = pOptionList->getFirst( );
```

The RulesOption name and value can now be retrieved for use using the getName and getValue member functions of the RulesOption class. For the name, create an STL_STRING and pass the variable to the method as a parameter. For the value, create an STL_STRING and pass the variable to the method as a parameter. The function writes the result to your object.

```
STL)STRING optionName, optionValue;
pOption->getName(optionName);
pOption->getValue (optionValue);
```

If a RulesOption exists in the list, enter a loop to retrieve all of the options in the list. Use the RulesOptionList::getNext method to retrieve all of the options in for this action by calling it from inside of a loop. The method returns NULL when no more options exist in the list.

```
pOption = pOptionList->getNext( );
```

After retrieving all of the options from this particular action, ask you action list for another action and repeat the process of retrieving this action's options.

```
pAction = pActionList->getNext( );
```

If no more actions exist in this list, ask the subscription list for another subscription and repeat the process of retrieving this subscription's actions.

```
pSubscription = pSubscriptionList->getNext( );
```

Evaluation Field Value Containers

The NNFieldValueContainer class is used as the base class for any class that contains field values that can be retrieved by name. The Formatter and NNNameValueList classes inherit from this class. Users can input their own object containing field values into the eval() API as long as the object inherits from this NNFieldValueContainer base class and has the correct methods.

```
class NNFieldValueContainer
{
  public:
    NNFieldValueContainer();
    virtual ~NNFieldValueContainer();
    virtual char* GetFieldString(char* name,int instance=-1)=0;
    virtual int GetFieldCount(char *name) = 0;
};
```

NNFieldValueContainer Member Functions

GetField

Gets the field represented by the name in the form of an NDO. The first instance = 0. All classes that need to pass data to a rules evaluation must inherit from the NNFieldValueContainer and implement GetField.

Syntax

```
const NNDOData * GetField(char * name, int instance)
```

Parameters

Name	Type	Description
name	char*	The name identifies the field being retrieved.
instance	int	The instance identifies the instance of the field in the corresponding container when repeating names exist.

Return Value

A pointer to an NDO Data object. Type of unset when the object is empty and the field does exist in the container.

GetFieldString

This GetFieldString method is used to return values for a specific instance in a message. This method is used for expressions containing <fieldname>[<instance>] where the first instance is represented as zero (0). All classes that need to pass a rules evaluation must inherit from NNFieldValueContainer and implement GetFieldString.

Syntax

```
char* NNFieldValueContainer::GetFieldString (char* name, int
instance = -1)
```

Description of Instance Syntax

For NEONFormatter messages, the -1 provides the instruction to retrieve the current instance. For NNNameValueList data evaluations, the -1 gets converted to zero (0) to retrieve the first instance.

Parameters

Name	Type	Description
name	char*	The name of the field to be evaluated.
instance	int	The instance of the field that determines the return value.

Return Value

This returns a null-terminated string representation of the last specific instance of this field in the evaluation data. NULL or an empty string is returned if there is no instance of the field.

GetFieldCount

This pure virtual member function retrieves an integer for the number of repeating instances of this field in the active message. A field can have a NULL or empty value in it; therefore, NULL values get included in count. All classes that need to pass a rules evaluation must inherit from NNFieldValueContainer and implement GetFieldCount.

Syntax

```
int GetFieldCount(char *name)
```

Parameters

Name	Type	Description
name	char*	The name of the field to be evaluated.

Return Value

Returns the number of repeating fields in the active message.

GetInputCodeSet

Gets the code set attribute for the active container and returns it to the user via the codeSetStr parameter.

Syntax

```
const char * GetInputCodeSet()
```

Parameters

N/A

Return Value

Character string representation for the code set name.

GetInputLocale

Gets the locale attribute for the active container and returns it to the user via the localeStr parameter.

Syntax

```
public abstract const char * GetInputLocale()
```

Parameters

N/A

Return Value

Character string representation of the locale name.

SetInputCodeSet

Sets the code set for the active container as an attribute of the object.

Syntax

```
public abstract int SetInputCodeSet(const char * codeset)
```

Parameters

Name	Type	Description
codeset	const char*	An STL string representation of the codeSetStr.

Return Value

An integer representing 0 for failure and 1 for success.

SetInputLocale

Sets the locale for the active container as an attribute of the object.

Syntax

```
int SetInputLocale(const char * locale)
```

Parameters

Name	Type	Description
locale	const char*	An STL string representation of the locale name.

Return Value

An integer representing 0 for failure and 1 for success.

NNNameValueList Member Functions

The NNNameValueList class is used to identify field values that can be retrieved by name. The NNNameValueList contains a list of field name and value pairs from the NNName and NNValue classes where the name is up to 120 characters and the value can be of any length for rules evaluation. Multiple instances of each field name can be stored. Access to these instances is performed with an index starting at 0.

Users are able to input their own object containing field values in the eval() API as long as the object inherits from the NNFieldValueContainer base class and has the correct functions. Users creating their own messages to be passed to a rules eval have to perform their own name length validation before passing them into a NNNameValueList.

Any call to GetFieldString must be accompanied by an instance for the NNNameValueList to retrieve the correct value. The instance defaults to -1; however, the NNNameValueList converts this to a zero and retrieves the first instance.

```
class NNNameValueList: public NNFieldValueContainer {
public:
  NNNameValueList();
   ~NNNameValueList();
   int Add(const NNName *pName, const NNValue *pValue);
   int Read(const NNName *pName, NNValue *pValue);
   int Update(const NNName *pName, NNValue *pValue);
   int Delete(const NNName *pName);
   int ClearAll();
   int GetFirst( NNName *&pName, NNValue *&pValue);
   int GetNext( NNName *&pName, NNValue *&pValue);
   char* GetFieldString(char *fieldname, int instance);
   const NNDOData * GetField(char * fieldname, int instance)
   int GetFieldCount(char * fieldname)
   const char * GetInputCodeSet()
   const char * GetInputLocale()
   int SetInputCodeSet(const char * codeset)
   int SetInputLocale(const char * locale)
};
```

Name	Type	Description
*pName	NNName	Object name
*pValue	NNValue	Object value

The NNNameValueList member functions use the NNName and NNValue classes to add name and value information. The names must be unique to retrieve the appropriate value. See*NNName Member Functions* on page 128 and *NNValue Member Functions* on page 141 for more information.

The NNNameValueList contains a list of field name and value pairs where the name is at most 120 characters and the value can be of any length for rules evaluation. Validation of the name length is only performed for NEONFormatter data. Anyone creating their own messages to be passed to a rules eval has to perform their own length validation before passing them into a NNNameValueList.

NameValueList Constructor

The NameValueList constructor creates an instance of this object to allow applications to call the methods for this class. The contents of this object is a list of field name and value pairs where the name is any length up to 120 characters and the value is any length. Currently, the name and value must be NULL-terminated.

Syntax

```
void NNNameValueList::NNNameValueList()
```

Parameters

N/A

Return Value

N/A

See Also

NNName

NNValue

~NNNameValueList Destructor

The NameValueList destructor allows applications to remove an instance of this object. The space for the name and value strings are deallocated by this destructor.

Syntax

```
void NNNameValueList::~NNNameValueList()
```

Return Value

N / A.

Add

The Add member function uses the NNName and NNValue classes to add a name and value pair to the list of items. The name may be up to 120 characters in length and the value can be of any length. If the name already exists, the pair is added to the list after the previous pair with the same field name.

Syntax

```
int NNNameValueList::Add(const NNName *pName, const NNValue
*pValue)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
NNValue	*pValue	Object value.

Return Value

Returns a 1 if the pair was added successfully.

See Also

NNName

NNValue

Read

The Read member function allows the user to retrieve a value from the list of items based on the item name. If the name and instance of that name exist, the value is returned in the pValue parameter.

Syntax

```
int NNNameValueList::Read(const NNName *pName, const NNValue
*pValue, int instance)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
NNValue	*pValue	Object value.
instance	int	Instance value when a group of items exist with the same name.

Return Value

Returns a 1 if the value was found successfully. Otherwise, returns zero (0).

See Also

NNName

NNValue

Update

The Update member function allows the user to update the value in a name/value pair inside the list of items. If the name and instance of that name already exist, the value is updated. If the name and instance of that name do not exist, it is not added to the list. The Add method is needed to add the name and value to the list.

Syntax

```
int NNNameValueList::Update(const NNName *pName, const NNValue
*pValue, int instance)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
NNValue	*pValue	Object value.
instance	int	Instance value when a group of items exist with the same name.

Return Value

Returns a 1 if the update was successful.

Returns 0 if the name and instance were not found or the update could not be performed.

See Also

NNName

NNValue

Delete

The Delete member function allows the user to delete a name and value pair inside the list of items based on the name. If the name and instance of that name exist, the item is deleted. If the name and instance of that name does not exist, no changes are made.

Syntax

```
int NNNameValueList::Delete(const NNName *pName, int instance)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
instance	int	Instance value when a group of items exist with the same name.

Return Value

Returns a 1 if the item was deleted successfully. A return value of 0 means that no changes were made.

See Also

NNName

NNValue

ClearAll

The ClearAll member function allows the user to delete the list of items.

Syntax

```
int NNNameValueList::ClearAll()
```

Parameters

N/A

Return Value

Returns a 1 if the items were deleted successfully.

See Also

NNName

NNValue

GetFirst

The GetFirst member function allows the user to retrieve the first name value pair in the NameValueList. If the name or value does not exist, the function returns

Syntax

```
int NNNameValueList::GetFirst(const NNName *pName, const
NNValue *pValue)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
NNValue	*pValue	Object value.

Return Value

Returns a 1 if the NameValue pair was found successfully. Returns a 0 if the function failed to find a valid pair.

See Also

NNName

NNValue

GetNext

The GetNext member functions allows the user to retrieve the next NameValue pair in the NameValue list.

Syntax

```
int NNNameValueList::GetNext(const NNName *pName, const NNValue
*pValue)
```

Parameters

Name	Type	Description
NNName	*pName	Object name.
NNValue	*pValue	Object value.

Remarks

GetFirst() should be called prior to calling GetNext().

Return Value

Returns a 1 if the NameValue pair was found successfully.

Returns a 0 if the function failed to find a valid pair or if the GetFirst function has not been applied.

See Also

NNName

NNValue

GetField

Retrieves an NDO representation of the field value for the given instance of the name provided.

Syntax

```
const NNDOData * GetField(char * fieldname, int instance)
```

Parameters

Name	Type	Description
fieldname	char*	The fieldname identifies the field in the corresponding container.
instance	int	The instance identifies the instance of the field in the corresponding container.

Return Value

NNDOData* is a pointer to the container's NDO representation of the value.

GetFieldCount

Retrieves an integer representing the number of elements with the given fieldname in the active container.

Syntax

```
public int GetFieldCount(char * fieldname)
```

Parameters

Name	Type	Description
fieldname	char*	The fieldname identifies the field in the corresponding container.

Return Value

An integer for the number of elements with the given fieldname.

GetInputCodeSet

Gets the code set attribute for the active container and returns it to the user via the codeSetStr parameter.

Syntax

```
public const char * GetInputCodeSet()
```

Parameters

N/A

Return Value

Character string representation for the code set name.

GetInputLocale

Gets the locale attribute for the active container and returns it to the user via the localeStr parameter.

Syntax

```
public const char * GetInputLocale()
```

Parameters

N/A

Return Value

Character string representation of the locale name.

GetInputCodeSet

Sets the code set for the active container as an attribute of the object.

Syntax

```
public int SetInputCodeSet(const char * codeset)
```

Parameters

Name	Type	Description
codeset	char*	An STL string representation of the codeSetStr.

Return Value

An integer representing 0 for failure and 1 for success.

SetInputLocale

Sets the locale for the active container as an attribute of the object.

Syntax

```
public int SetInputLocale(const char * locale)
```

Parameters

Name	Type	Description
locale	char*	An STL string representation of the locale name.

Return Value

An integer representing 0 for failure and 1 for success.

NNName Member Functions

The NNName class is used for some of the NNNameValueList methods to identify the object from which field name information is retrieved. The names within the object may be up to 120 characters for use within rules. This class enables retrieval of field or object name information without using the NEONFormatter to parse the information.

```
class NNName {
public:
    NNName();
    NNName(char* name);
    NNName(char* name, int length);
    NNName(const NNName& Original);
    ~NNName();
    int set(char* name);
    int set(char* name, int length);
    friend bool operator<(const NNName& name1,
                          const NNName& name2);
    friend bool operator==(const NNName& name1,
                          const NNName& name2);
    void operator=(const NNName& name1);
    bool IsEmpty();
    char* GetString();
    int GetLength();
};
```

NNName Constructor

The default constructor creates an empty NNName object. Use one of the set() methods to set the name. IsEmpty() returns true, GetLength() returns 0, and GetString() returns an empty string after NNName is created using this constructor.

Syntax

```
NNName::NNName()
```

Parameters

N/A

Return Value

None

NNName Constructor

This constructor creates a NNName object and sets the name to the NULL-terminated value given. The character array is copied and the length of the NNName object is set to strlen(name).

Syntax

```
NNName::NNName(char* name)
```

Parameters

Name	Type	Description
name	char*	NULL-terminated Field name

Return Value

None

NNName Constructor

This constructor creates a NNName object and sets the name to the value given. The character array is copied and the length of the NNName object is set to the length given. The character array has a NULL (\0) placed at the end when using this method.

Syntax

```
NNName::NNName(char* name, int length)
```

Parameters

Name	Type	Description
name	char*	Field name
length	int	Data length, in bytes, of the name to be evaluated

Return Value

None

NNName Copy Constructor

This copy constructor creates a NNName object and sets the name to the name in the passed NNName parameter. The character array is copied and the length of the NNName object is set to the length given.

Syntax

```
NNName::NNName(const NNName &Original)
```

Parameters

Name	Type	Description
&Original	NNName	NNName object to copy

Return Value

None

NNName Destructor

This NNName destructor deallocates the memory used by the character array.

Syntax

```
NNName::~NNName()
```

Parameters

N/A

Return Value

None

set

This set member function sets the name to the NULL-terminated value given. The character array is copied and the length of the NNName object is set to strlen(name). If the NNName was not previously empty, the old name is deallocated before the new name is copied.

Syntax

```
int NNName::set(char* name)
```

Parameters

Name	Type	Description
name	char*	NULL-terminated Field name

Return Value

Return value is always 1.

set

This set member function sets the name to the value given. The character array is copied and the length given. If the NNName was not previously empty, the old name is deallocated before the new name is copied. The character array has a NULL (\0) placed at the end when using this method.

Syntax

```
int NNName::set(char* name, int length)
```

Parameters

Name	Type	Description
name	char*	Field name
length	int	Data length, in bytes, of the name to be evaluated

Return Value

Return value is always 1.

operator<

This operator compares two NNName instances. Each character in the name is compared (case-sensitive comparison). If the characters are all the same and objects are the same length, the NNName objects are said to be equal. If one NNName is longer but all the characters up to that point are the same, the longer NNName is said to be greater.

This function returns different values on systems with different character sets, such as ASCII and EBCDIC systems, which sort characters in different orders. This function uses the sort order for the local system.

Syntax

```
bool operator< (const NNName& name1, const NNName& name2)
```

Parameters

Name	Type	Description
name1	NNName&	First object instance against which the second instance is evaluated
name2	NNName&	Second object instance

Return Value

Return value true if the first object is less than the second object Otherwise false is returned.

operator==

This operator function compares two NNName instances. Each character in the name is compared (case-sensitive comparison). If the characters are all the same and objects are the same length, the NNName objects are equal. If one NNName is longer but all the characters up to that point are the same, the longer NNName is greater.

This function returns different values on systems with different character sets, such as ASCII and EBCDIC systems, which sort characters in different orders. This function uses the sort order for the local system.

Syntax

```
bool operator== (const NNName& name1, const NNName& name2)
```

Parameters

Name	Type	Description
name1	NNName&	First object instance against which the second instance is evaluated
name2	NNName&	Second object instance

Return Value

Return value true if the object values are equal. Otherwise false is returned.

operator=

This assignment operator sets the current NNName value to be the same as the one passed into the method (right-hand side of the equal sign). If the current NNName has data, that memory is cleared and the character array and length are copied from the NNName parameter.

Syntax

```
void NNName::operator= (const NNName& name1)
```

Parameters

Name	Type	Description
name1	`NNName&`	Object instance that specifies the setting for the current NNName value

Return Value

None.

IsEmpty

The IsEmpty method returns true if the NNName is empty (empty string, length is 0).

Syntax

```
bool NNName::IsEmpty()
```

Parameters

N/A

Return Value

Returns true if the object is empty. Otherwise false is returned.

GetString

The GetString method returns a NULL-terminated string value. If the original character array has embedded NULL characters, these characters look like the end of the string. An empty string is returned ("") if the NNName is empty.

Syntax

```
char* NNName::GetString()
```

Parameters

N/A

Remarks

Do not modify the string returned. If modifications are required, copy the value into a locally-allocated memory location.

Return Value

Returns a character pointer to the memory inside the object.

GetLength

The GetLength method returns the length of the character array (up to, but not including, the final NULL-character). This should match strlen(SetString()).

Syntax

```
int NNName::GetLength()
```

Parameters

N/A

Return Value

Returns a non-negative whole number for the length of the NNName; returns zero (0) if the NNName is empty.

NNValue Member Functions

The NNValue class is used for some of the NNNameValueList methods to identify the value information to retrieve. The values within the object may be up of any length. This class enables retrieval of field or object value information without using the NEONFormatter to parse the information.

```
class NNValue {
public:
  NNValue();
  NNValue(char * pValue, char * pEncoding = 0, char *
                            pLocale = 0);
  NNValue(char * pValue, unsigned int & length, char *
                            pEncoding = 0, char * pLocale = 0);
  NNValue(const NNValue& Original);
  ~NNValue();
  int set(char * pValue, char * pEncoding = 0, char *
                            pLocale = 0);
  int set(char* value, int length);
  friend bool operator<(const NNValue& value1,
                          const NNValue& value2);
  friend bool operator==(const NNValue& value1,
                          const NNValue& value2);
  void operator=(const NNValue& value1);
  bool IsEmpty();
  char* GetString();
  int GetLength();
  NNValue( NNDOData * pNdoData)
  const const I18NEncodingContext * getCodeSet()
  const NNDOData * GetField()
  const const I18NLocaleContext * getLocale()
  int set( NNDOData * pNdoData)
};
```

NNValue Constructor

The default constructor creates an empty NNValue object. Use one of the set() methods to then set the value. IsEmpty() returns true, GetLength() returns 0, and GetString() returns an empty string after NNValue is created using this constructor.

Syntax

```
public NNValue( NNDOData * pNdoData)
```

Parameters

Name	Type	Description
pNdoData	NNDOData	The NDO object that used to create a new NNValue.

Return Value

None

NNValue Constructor

This constructor creates a NNValue object and sets the value to the NULL-terminated value given. The character array is copied and the length of the NNValue object is set to strlen(value).

Syntax

```
public NNValue(char * pValue, unsigned int & length, char *
pEncoding, char * pLocale)
```

Parameters

Name	Type	Description
pValue	char*	NULL-terminated Field value.
length	int	Data length, in bytes, of the value to be evaluated.
pEncoding	char*	The encoding character string to be used for this object's encoding.
pLocale	char*	The locale character string to be used for this object's locale.

Return Value

None

NNValue Constructor

This constructor creates a NNValue object and sets the value to the value given. The character array is copied and the length of the NNValue object is set to the length given. The character array has a NULL (\0) placed at the end when using this method.

Syntax

```
public NNValue(char * pValue, char * pEncoding, char * pLocale)
```

Parameters

Name	Type	Description
pValue	char *	Field value.
pEncoding	char *	The encoding character string to be used for this object's encoding.
pLocale	char *	The locale character string to be used for this object's locale.

Return Value

None

NNValue Copy Constructor

This copy constructor creates a NNValue object and sets the value to the value in the passed NNValue parameter. The character array is copied and the length of the NNValue object is set to the length given.

Syntax

```
NNValue::NNValue(const NNValue &Original)
```

Parameters

Name	Type	Description
&Original	NNvalue	NNValue object to copy.

Return Value

None

NNValue Destructor

This NNValue destructor deallocates the memory used by the character array.

Syntax

```
NNValue::~NNValue()
```

Parameters

N/A

Return Value

None

getCodeSet

Retrieves the I18NEncodingContext object for this object's encoding.

Syntax

```
public const I18NEncodingContext * getCodeSet()
```

Parameters

N/A

Return Value

18NEncodingContext* is a pointer to a globally allocated I18NEncodingContext object.

getLocale

Retrieves the I18NLocaleContext object for this object's locale.

Syntax

```
public const I18NLocaleContext * getLocale()
```

Parameters

N / A

Return Value

I18NLocaleContext is a pointer to a globally allocated I18NLocaleContext object.

GetField

Retrieves an NNOData object representation of this object's data contents.

Syntax

```
public const :: NNDOData * GetField()
```

Parameters

N/A

Return Value

A pointer to this object's NNDOData attribute.

set

Sets this object's attributes with the NDO parameter.

Syntax

```
public int set( NNDOData * pNdoData)
```

Parameters

Name	Type	Description
pNdoData	NNDOData	The NDO representation of this object's new data attributes.

Return Value

An integer representing 0 for failure and 1 for success.

set

This set member function sets the value to the NULL-terminated value given. The character array is copied and the length of the NNValue object is set to strlen(value). If the NNValue was not previously empty, the old value is deallocated before the new value is copied.

Syntax

```
public int set(char * pValue, char * pEncoding, char * pLocale)
```

Parameters

Name	Type	Description
pValue	char *	NULL-terminated Field value.
pEncoding	char *	The character string representation of this object's new encoding.
pLocale	char *	The character string representation of this object's new locale.

Return Value

Return value is always 1.

set

This set member function sets the value to the value given. The character array is copied and the length given. If the NNValue was not previously empty, the old value is deallocated before the new value is copied. The character array has a NULL (\0) placed at the end when using this method.

Syntax

```
public int set(char * pValue, unsigned int length, char *
pEncoding, char * pLocale)
```

Parameters

Name	Type	Description
pValue	char*	Field value
length	int	Data length, in bytes, of the value to be evaluated
pEncoding	char *	The character string representation of this object's new encoding.
pLocale	char *	The character string representation of this object's new locale.

Return Value

Return value is always 1.

operator<

This operator compares two NNValue instances. Each character in the value is compared (case-sensitive comparison). If the characters are all the same and objects are the same length, the NNValue objects are said to be equal. If one NNValue is longer but all the characters up to that point are the same, the longer NNValue is said to be greater.

This function returns different values on systems with different character sets, such as ASCII and EBCDIC systems, which sort characters in different orders. This function uses the sort order for the local system.

Syntax

```
bool operator< (const NNValue& value1, const NNValue& value2)
```

Parameters

Name	Type	Description
value1	`NNValue&`	First object instance against which the second instance is evaluated
value2	`NNValue&`	Second object instance

Return Value

Return value true if the first object is less than the second object. Otherwise, false is returned.

operator==

This operator function compares two NNValue instances. Each character in the value is compared (case-sensitive comparison). If the characters are all the same and objects are the same length, the NNValue objects are said to be equal. If one NNValue is longer but all the characters up to that point are the same, the longer NNValue is said to be greater.

This function returns different values on systems with different character sets, such as ASCII and EBCDIC systems, which sort characters in different orders. This function uses the sort order for the local system.

Syntax

```
bool operator== (const NNValue& value1, const NNValue& value2)
```

Parameters

Name	Type	Description
value1	`NNValue&`	First object instance against which the second instance is evaluated
value2	`NNValue&`	Second object instance

Return Value

Return value true if the object values are equal. Otherwise false is returned.

operator=

This assignment operator sets the current NNValue value to be the same as the one passed into the method (right-hand side of the equal sign). If the current NNValue has data, that memory is cleared and the character array and length are copied from the NNValue parameter.

Syntax

```
void NNValue::operator= (const NNValue& value1)
```

Parameters

Name	Type	Description
value1	`NNValue&`	Object instance that specifies the setting for the current NNValue value

Return Value

None.

IsEmpty

The IsEmpty method returns true if the NNValue is empty (empty string, length is 0).

Syntax

```
bool NNValue::IsEmpty()
```

Parameters

N/A

Return Value

Returns true if the object is empty. Otherwise false is returned.

GetString

The GetString method returns a NULL-terminated string value. If the original character array has embedded NULL characters, these characters look like the end of the string. An empty string is returned ("") if the NNValue is empty.

Syntax

```
char* NNValue::GetString()
```

Parameters

N/A

Remarks

Do not modify the string returned. If modifications are required, copy the value into a locally-allocated memory location.

Return Value

Returns a character pointer to the memory inside the object.

GetLength

The GetLength method returns the length of the character array (up to, but not including, the final NULL-character). This should match strlen(SetString()).

Syntax

```
int NNValue::GetLength()
```

Parameters

N/A

Return Value

Returns a non-negative whole number for the length of the NNValue; returns zero (0) if the NNValue is empty.

Chapter 3

NEONRules Management APIs

This chapter includes the following information:

- *NEONRules Management API Structures*
- *Overall NEONRules Management APIs and Macros*
- *Application Group Management APIs*
- *Message Type Management APIs*
- *Rule Management APIs*
- *Permissions APIs*
- *Operator Management APIs*
- *Expression Management APIs*
- *Argument Management APIs*
- *Subscription Management APIs*
- *Action Management APIs*
- *Option Management APIs*
- *NEONRules Management Error Handling*

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

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

NEONRules 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

The names of formats and fields should not be changed if they are used by a rule. The following occurs if 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.
- 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 fails.

See the NEONFormatter ***Programming Reference*** for information on changing formats and field names.

NEONRules Management API Structures

NNDate

NNDate is passed as part of an argument in several NEONRules 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 NEONRules Management APIs and Macros

NNRMgrInit

When using NEONRules 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

NEONRules

NNRMgrInit() should be called prior to any NEONRules Management API calls. For information about the DbmsSession Type to use, see OpenDbmsSession() in ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms*** *Application Development Guide*.

Return Value

Returns a pointer to an instance of a NNRMgr object.

See Also

NNRMgrClose

NNRMgrClose

When using NEONRules 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 NEONRules 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 cannot manage rules without calling NNRMgrInit() again.

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

Return Value

None

See Also

NNRMgrInit

NNR_CLEAR

When using NEONRules 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 NEONRules Management APIs can alert the user to a non-initialized structure.

Syntax

```
NNR_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any rules management structure	Input	Any structure used in NEONRules 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 used to create applications and associate the applications with rules, subscriptions, and user permissions.

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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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 NEONRules 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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules Management API.

See Also

NNR_CLEAR

NNRAppUpdate

NNRAppUpdate is a structure designed to pass update information within the NEONRules Management APIs. It must be cleared using NNR_CLEAR prior to being populated, and must be populated prior to any NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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 NEONRules. The user creates a name and provides it to NNRMgrAddApp(), which then saves it in NEONRules. Only after an application group has been defined can the application name be used in other NEONRules 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 NEONRules 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.

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 NEONRules 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.

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 NEONRules Management object returned from call to NNRMgrInit().
RAppData	NNRAppReadData *const	Output	NNRMgrGetFirstApp populates this structure.

Remarks

NNRMgrInit() should be called prior to any NEONRules 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.

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 NEONRules Management object returned from call to NNRMgrInit().
RAppData	NNRAppReadData *const	Output	NNRMgrGetNextApp populates this structure.

Remarks

NNRMgrInit() should be called prior to any NEONRules 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.

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 NEONRules 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.

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 NEONRules 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 NEONRules 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 are not deleted. However, the components that the user does have permission for are 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 only deletes if there are not rules and subscriptions left. The application group only deletes 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 NEONRules 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 NEONRules 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.

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.

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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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. If the EvalType is empty, NNSYRF_FORMATTER is assumed.

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	Valid entries are NNSYRF_FORMATTER and NAME_VALUE.
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 NEONRules 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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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().

If the message type is new (not used in an existing application group), this function creates a new identifier for the message type.

If no APP_NAME is given in the pRMsg parameter, the message type is added to the database but not to any specific application group. If the message type already exists, it is added to the application group if it does not already belong to that application group. The EvalType is ignored if the message type already exists in the database and is just added to the application group.

Syntax

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

Parameters

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

Name	Type	Input/ Output	Description
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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrReadMsg

NNRMgrReadMsg

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

If no APP_NAME is given in the pRMsg parameter, the message type is read from the database but not associated with a specific application group.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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.

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

Return Value

Returns 1 if the message is read successfully; zero (0) if an error occurs.

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 NEONRules 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.

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 NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrDeleteEntireMsg

NNRMgrDuplicateMsg

NNRMgrGetFirstMsg

NNRMgrReadMsg

NNRMgrUpdateMsgName

NNRMgrUpdateMsgName modifies all NNNameValueList message types with the name given in the pRMsg parameter with the data given in the pRMsgData parameter.

The user can change the Message Type name for Name-Value Message Types.

- For those Message types that refer to Format Name, the Message Type must be updated if the Format Name is changed.
- If the EvalType is NNSYRF_FORMATTER, the MsgName must refer to a valid Input Format Name.

Syntax

```
const long NNRMgrUpdateMsgName(
      NNRMgr *pMgr,
      const char *OldMsgName,
      const char *NewMsgName);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr*	Input	Valid NEONRules Management object returned from call to NNRMgrInit().
OldMsgName	const char	Input	
NewMsgName	const char	Output	

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 NEONRules 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.

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 only deletes if there are no rules and subscriptions left.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddMsg

NNRMgrDuplicateMsg

NNRMgrReadMsg

Rule Management APIs

Use Rule Management APIs to create rules that contain expressions and associate rules with subscriptions and user permissions.

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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
RuleName [RULE_NAME_LEN]	char	Name of the rule defined within an application group and message name pair. This rule name is defined by the user. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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 NEONRules 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 NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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 NEONRules 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 NEONRules 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.

Of the data in the structures passed to NNRMgrAddRule(), not all variables used in release 4.11 or later need to be populated in the AddRule method.

The following are the variables that are used:

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

typedef struct NNRRuleData {
        NNDate DateChange;
        int ChangeAction;
        int ArgumentCount;
        int OrCondition;
        int SubscriberIndex;
        int RuleActive;
// 1 => rule is active, 0 =>rule is inactive
        NNDate
```

```
RuleEnableDate;
        NNDate
RuleDisableDate;
        long InitFlag;
}  NNRRuleData;
```

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.

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 NEONRules 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 NEONRules 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.

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 NEONRules 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 NEONRules 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.

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 are 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 NEONRules 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 NEONRules 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.

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 NEONRules 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.

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 are 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 changes occur.

Syntax

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

Parameters

Name	Type	Input / Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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 NEONRules 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 changes occur.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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 NEONRules 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 NEONRules 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 NEONRules Management API.

See Also

NNR_CLEAR

NNPermissionData

NNPermissionData is passed as an argument in several NEONRules 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 NEONRules 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 NEONRules 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 NEONRules 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 NEONRules Management APIs can alert the user to a non-initialized structure.

Syntax

```
NN_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any NEONRules management permissions structure	Input	Any structure used in NEONRules 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 NEONRules 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.

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 NEONRules 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.

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 NEONRules 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.

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 NEONRules 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.

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 NEONRules 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.

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 NEONRules 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.

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 NEONRules 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 NEONRules 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.

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 NEONRules 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 NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstOperator()

NNRMgrGetErrorNo()

NNRMgrGetError()

Expression Management APIs

The following addressing is accepted in the NEONRules Expression Management APIs:

```
FIELD_NAME[instance]

FIELD_NAME[*]

MAX( field_instance_definition )

MIN( field_instance_definition )

COUNT( field_instance_definition )

LAST( field_instance_definition )

AVG ( field_instance_definition )

SUM( field_instance_definition )

FIELD_NAME[rules_max_operator( field_instance_definition )]

FIELD_NAME[rules_min_operator ( field_instance_definition )]

FIELD_NAME[ field_instance_definition  rules_equal_operator
comparison_value]

Rule ::= rule_expression [ boolean_operator rule_expression ]

rule_expression ::= field_expression rules_operator [
field_expression | constant ]

field_expression ::= [ field_name | field_instance_expression ]

field_instance_expression ::= instance_function
(field_instance_definition)
```

```
instance_function ::= [ MIN | MAX | COUNT |  SUM | LAST | AVG ]

field_instance_defintion ::= [ field_name[instance] |
field_name[*] ]

boolean_operator ::= [ | | & ]

rules_operator ::= [ STRING= | INT< | EXISTS | etc. ]

rules_equal_operator ::= [ STRING= | INT= | FLOAT=| etc. ]

rules_max_operator ::= [ STRING_MAX | INT_MAX | FLOAT_MAX |
etc. ]

rules_min_operator ::= [ STRING_MAX | INT_MAX | FLOAT_MAX |
etc. ]
```

Rules for Creating Expressions

- Functions must have a field within parens ( ).
- The instance for a field within a function (other than SUBSTRING) must be an asterisk.
- SUBSTRING does not require an instance to be specified. SUBSTRING(F3,1,4) is valid.
- Left and Right (if right is a Field) operand needs to be quoted.
- Field to Field comparisons cannot compare field instances.
- Quotes are not allowed in field names if you are using field functions. We run out of quotes.
- If a field name needs to be quoted or the operand contains any functions which have parens ( ), the operand must be enclosed in double quotes.
- Sibling arguments require the following:

  The only allowable field functions in a sibling argument are MIN and MAX.

The only comparisons allowed in sibling argument are Equal ones (STRING=, INT=, FLOAT= ....)

(An sibling argument is what is between the [].)

Example:

```
F1[F2[*] STRING= '1'] INT> 10
         F2[*] STRING= '1' is the sibling argument.
F1[MAX_INT(F2[*])] INT> 10
        MAX_INT(F2[*]) is the sibling argument.
```

- You can only have 1 function per operand. (Operands are what is on the left and right (if F2F) of the operator.)

  Example:

```
MAX_INT(F3[*]) F2FINT= MIN_INT(F4[*]) is valid
SUBSTRING(MAX_STRING(F3[*]),3,4) F2FSTRING=
 MIN_STRING(F4[*]) is invalid.
```

Expression Management API Structures

NNRExp

NNRExp is passed as an argument in several NEONRules 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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules, the message type is the input format name. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules Management API.

NNRExpData

NNRExpData is passed as an argument in several NEONRules 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];
} 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 NEONRules Management API.
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 NEONRules 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.

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 NEONRules 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.

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 changes occur.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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.

See Also

NNRMgrDeleteEntireRule

NNRMgrAddExpression

NNRMgrReadExpression

Argument Management APIs

These APIs are used only for backwards compatibility. The Expression APIs should be used instead.

Argument Management API Structures

NNRArg

NNRArg structure is passed as a pointer as the second parameter of selected Argument Management APIs. The pointer cannot 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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. Using NEONFormatter, the message type is the input format name. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
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.NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules Management API.

NNRArgData

NNRArgData structure 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;
        int OperatorType;
        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 is applied. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
OperatorId	int	ID retrieved by NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().
OperatorType	int	Type retrieved by NNRMgrGetFirstOperator() or NNRMgrGetNextOperator().
SecondFieldName [SECOND_FIELD_ NAME_LEN]	char	Value to which the field is compared for a field to field operator. NULL-terminated string of length 1 to 120 inclusive.
ArgValue [ARG_VALUE_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 NEONRules 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 NEONRules 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.

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.

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.

Note:
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 NEONRules Management object returned by 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.

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.

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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
MsgName [MSG_NAME_LEN]	char	Name of the message for which the user is defining rules for message evaluation. Using NEONFormatter, the message type is the input format name. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive. RuleName is required only when adding a subscription to a specific rule. It is ignored for action, option, update, and delete functions.
SubsName [SUBS_NAME_LEN]	char	Name of the subscription associated with a message name and application group. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules Management API.

NNRSubsData

NNRSubsData is passed as a pointer as the third parameter of select Subscription Management APIs. The pointer cannot be NULL and must be cleared prior to being populated 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 NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
MsgName [MSG_NAME_LEN]	char	Name of the message type to identify the subscription. NULL-terminated string of length 1 to 120 inclusive.
RuleName [RULE_NAME_LEN]	char	Name of the rule to link to the subscription, if provided. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
SubsName [SUBS_NAME_LEN]	char	Name of the subscription to be read. NULL-terminated string of length 1 to 120 inclusive.
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	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 NEONRules 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 NEONRules 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 is 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 changes occur.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules Management object returned from call to NNRMgrInit().

Name	Type	Input/ Output	Description
pRSubs	const NNRSubs *	Input	Must be populated prior to this function call. Users need not specify the rule name.
pRSubsData	const NNRSubsData *	Input	Must be populated prior to calling this function. Set DateChange, ChangeAction, SubsEnableDate and SubsDisableDate 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() checks to see if the subscription exists in the rule set. If the subscription already exists, the function is set to the RERR_SUBS_ALREADY_EXISTS_IN_RULESET error code. If not, the function creates 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.

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 NEONRules 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.

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 NEONRules 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 NEONRules 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 NEONRules Management object returned from call to NNRMgrInit().
pRSubsReadData	NNRSubsRead Data * const	Output	Populated by this function call.

Remarks

NNRMgrInit() should be called prior to any NEONRules 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 NEONRules Management object returned from call to NNRMgrInit().
pSub	const NNRSubs*	Input	Must be populated prior to this function call.
NewSubsName	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.

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.

Subscription Update permission is also checked when an action or option is 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 NEONRules 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.
pRSubs Update	const NNRSubsUpdate *	Input	Must be populated prior to this function call.

Remarks

NNRMgrInit() should be called prior to any NEONRules 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 NEONRules 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 occurs 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.

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 is returned.

When deleting subscription information, user permission to update the subscription is 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 changes occur.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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 NEONRules 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.

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 NEONRules 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 NEONRules 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.

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

Remarks

NNRMgrInit() should be called prior to any NEONRules 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.

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.

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. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
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 NEONFormatter, the message type is the input format name. NULL-terminated string of length 1 to 120 inclusive.
RuleName [RULE_NAME_LEN]	char	The rule name is ignored for actions and options. NULL-terminated string of length 1 to 120 inclusive.
SubsName [SUBS_NAME_LEN]	char	Name of the subscription associated with a rule name, message name, and application group. NULL-terminated string of length 1 to 120 inclusive.
ActionName [ACTION_NAME_ LEN]	char	Name of the action associated with this subscription. NULL-terminated string of length 1 to 120 inclusive.
OptionName [OPTION_NAME_ LEN]	char	Name of the first option associated with this action. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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 NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
OptionName [OPTION_NAME_LEN]	char	Name of the first option associated with the action. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
OptionValue [OPTION_VALUE_LEN]	char	Static value of the first option if there are no actions.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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 NEONRules 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	Set DateChange and ChangeAction to NULL; 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.

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 NEONRules 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.

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 NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstAction

NNRMgrResequenceAction

NNRMgrResequenceAction() enables the user to resequence actions within a subscription. NNRMgrResequenceAction() moves the action to the specified new position. The user provides the unique application group, message type, subscription name, current position, and the position to move the action to.

For example, the following actions exist in your code:

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

You want reformat to occur before putqueue. 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 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 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 does not have Update access, an error is returned indicating that the user does not have Update permission, and no changes occur.

Syntax

```
const long NNRMgrResequenceAction (
      NNRMgr *pMgr,
      const NNRAction *pRAction,
      int oldPosition,
      int newPosition);
```

Parameters

Name	Type	Input/ Output	Description
pMgr	NNRMgr *	Input	Valid NEONRules 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 NEONRules Management API calls.

NEONRules 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 NEONRules 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 update.

Remarks

NNRMgrInit() should be called prior to any NEONRules 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 NEONRules 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 delete.

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.

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().

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 in which the user is defining rules for evaluation. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
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. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.
InitFlag	long	Flag used to determine if variables have been initialized prior to calling a NEONRules 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 NEONRules 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. NULL-terminated string of length 1 to 120 inclusive.
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. NULL-terminated string of length 1 to 120 inclusive.

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 NEONRules 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 update. NULL-terminated string of length 1 to 120 inclusive.
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 NEONRules 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 is 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 NEONRules 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	Set DateChange and ChangeAction to NULL; 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.

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 NEONRules 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.

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 NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrGetFirstOption

NNRMgrResequenceOption

NNRMgrResequenceOption() enables the user to resequence options within an action. NNRMgrResequenceOption() moves the option to the specified new position. The user provides the unique application group, message type, rule name, subscription name, current position, 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 is 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 NEONRules 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 NEONRules Management API calls.

NEONRules 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 NEONRules 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 NEONRules 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 NEONRules 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.

See Also

NNRMgrInit

NNR_CLEAR

NNRMgrAddOption

NNRMgrGetFirstOption

NNRMgrGetNextOption

NNRMgrResequenceOption

NEONRules Management Error Handling

NNRGetErrorNo

NNRGetErrorNo() retrieves the error number from previous NEONRules 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 NEONRules Management calls; returns zero (0) if no NEONRules Management functions were called prior to this call or NNR_NO_ERR if no error exists. Use NNRGetErrorMessage() to get the associated error message.

See Also

NNRGetErrorMessage

NNRMgrInit

NNRGetErrorMessage

NNRGetErrorMessage() retrieves the error message from previous NEONRules Management calls.

Syntax

```
const char * NNRGetErrorMessage(NNRMgr *pRMgr);
```

Parameters

Name	Type	Input/ Output	Description
pRMgr	NNRMgr *	Input	Name of a current NEONRules Management object.

Return Value

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

See Also

NNRGetErrorNo

NNRMgrInit

Chapter 4

Error Messages

The following lists of errors 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 NEONRules Broker specific and are not included in this list. For more information, see the ***System Management Guide***.

Data Processing Related Errors

Code	Message	Explanation	Response
-1000	Unknown error code or no error	No matching error code.	
-1001	Rules configuration missing Application Group	Application group passed into eval() does not exist for Rules database. 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	Rules configuration missing Message Type	Application group message type pair passed into eval() does not exist for the Rules database. 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	Rules not configured or Operations missing for message	Rule data in the database is incorrect.	Run Consistency Checker to check data.
-1004	Rules configuration missing Arguments for message	Rule missing active arguments in database.	Run Consistency Checker to check data.
-1005	Rules configuration missing Rules	No active rules defined for the application group / message type pair.	Review the data in the database.
-1006	Rules configuration missing Subscriptions	No active subscriptions for the rules in the application group / message type pair.	Run Consistency Checker to check data.
-1007	Rules configuration missing Subscription Actions	At least one subscription does not have any actions.	Make sure all rules have subscription actions.

Code	Message	Explanation	Response
-1008	Rules configuration missing Boolean Operators	All rules have a single argument.	Error code is used internally as a warning. It should never appear to the user. Call New Era of Networks Technical Support.
-1009	Major Database Error Retrieving Application Group / Message Type	Major database error.	Verify that database is up and schema is okay. This error may occur if an old version of the Rules schema is being used.
-1010	Major Database Error Retrieving Arguments	Major database error.	Verify that database is up and schema is okay.
-1011	Major Database Error Retrieving Boolean Operators	Major database error	Verify that database is up and schema is okay.
-1012	Major Database Error Retrieving Operations	Major database error	Verify that database is up and schema is okay.
-1013	Major Database Error Retrieving Rules	Major database error	Verify that database is up and schema is okay.
-1014	Major Database Error Retrieving Subscription Actions	Major database error	Verify that database is up and schema is okay.
-1015	Major Database Error Retrieving Subscriptions	Major database error	Verify that database is up and schema is okay.

Client Code Errors

Code	Message	Explanation	Response
-2000	Unknown error code or no error	No error.	
-2001	NULL or dead dbms connection provided to Rules daemon	The session pointer is invalid.	Check DBMS and run OpenDbmsSession() again.
-2002	NULL or missing message type provided to Rules daemon	No message type name set in eval().	Send in a valid message type.
-2003	Error adding argument to Rules daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2004	Wrong number of argument columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2005	Unexpected argument column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2006	NULL argument column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2007	Error adding operation to Rules daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2008	Wrong number of operation columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2009	Unexpected operation column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2010	NULL operation column during load	Data in the database is incorrect.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2011	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	Wrong number of rule columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2013	Unexpected rule column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2014	NULL rule column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2015	Error adding a Subscription to Rules daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2016	Wrong number of subscription columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2017	Unexpected subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2018	NULL subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2019	Error adding a Rule Subscription to Rules daemon	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2020	Wrong number of Rule Subscription columns during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2021	Unexpected Rule Subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2022	NULL Rule Subscription column during load	Data in the database is incorrect.	Run Consistency Checker to check data.
-2023	INTERNAL ERROR - failed to resize operations	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2024	INTERNAL ERROR - failed to resize rules	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2025	Formatter failed to parse input message	The message type may not match the format of the input message.	Check Input Format Name (MsgType) and message (use apitest).
-2026	INTERNAL ERROR - incorrect operation count	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2027	Invalid Argument loaded - operation id too high	Data in the database is incorrect.	Run Consistency Checker to check data.
-2028	Input message had an invalid length	Call to eval() had an invalid msglen parameter.	Check the parameters sent to eval().
-2029	Rule argument count is invalid - check table	Data in the database is incorrect.	Run Consistency Checker to check data.
-2030	Formatter instance is NULL	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2031	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.
-2032	Internal Error - Evaluation failure #1	Problem evaluating part of a rule – operator may be invalid.	Run Consistency Checker to check data.
-2033	Internal Error - Load failure #1	Problem loading arguments.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2034	Internal Error - Load failure #2	Problem loading operator.	Run Consistency Checker to check data.
-2035	Internal Error - Evaluation failure #2	Problem evaluating part of a rule; operator may be invalid.	Run Consistency Checker to check data.
-2036	Database type not supported	Invalid DbmsType in the Session variable used to create Rules daemon.	Check call to OpenDbmsSession().
-2037	Internal Error - Load failure #3	Problem loading subscriptions.	Run Consistency Checker to check data.
-2038	Internal Error - Load failure #4	Problem loading subscriptions.	Run Consistency Checker to check data.
-2039	Empty Input Value for Application Group Name	No application group name passed into eval().	Check call to eval().
-2040	Empty Input Value for Message Name	No message type name passed into eval().	Check call to eval().
-2041	Internal Error - Lookup failure #1	Problem loading message type.	Run Consistency Checker to check data.
-2042	Internal Error - Lookup failure #2	Problem loading application group.	Run Consistency Checker to check data.
-2043	Internal Error - NULL Engine Instance	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2044	Error setting HitList	gethitrule() had problems retrieving hit rules.	Run Consistency Checker to check data.
-2045	Error setting NoHitList	getnohitrule() had problems retrieving no hit rules.	Run Consistency Checker to check data.
-2046	Internal Error - No error handler	(Should never see) Memory may be low.	Shut down Rules daemon and restart.

Code	Message	Explanation	Response
-2047	Internal Error - Error Setting Thread Specific Data	Problem with threading - maybe too many threads.	Shut down process immediately, check system, and restart.
-2048	Internal Error - Error Loading Boolean Operators	Problem loading Boolean operators.	Run Consistency Checker to check data.
-2049	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 Subscription Action / Option to Rules daemon.	(Should never see) Memory may be low.	Shut down Rules daemon and restart.
-2051	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 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	Error removing Rule Subscription Link to Rules daemon.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.
-2054	Error comparing old and new Subscription Rule Links.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.
-2055	Error Removing Reload Component from Reload List in Rules daemon.	(Should never see) Memory may be corrupted.	Shut down Rules daemon and restart.

Code	Message	Explanation	Response
-2056	Error allocating memory for new Rules daemon object.	(Should never see) Severe error. Memory must be low.	Shut down Rules daemon and restart.
-2057	Invalid operation in expression.	Operator type in a rule for this Rule Set is invalid.	Run Consistency Checker to check data.
-2058	Duplicate found for Sibling; cannot determine correct field.	Message contain two fields that met criteria for Sibling relationship; no way to determine which field to use.	Check message sent to eval() and make sure the Sibling has a unique value.
-2059	SubString function failed; possible invalid parameters.	Substring function may have been called with a negative number for the start or length of the field value.	Check the expression for a SubString call with invalid parameters.
-2060	Internal Error - Evaluation failure #3	Problem evaluating part of a rule.	Check system resources. Run Consistency Checker to check data.
-2061	Field Name 2 is missing for a Field-to-Field comparison	A Rule in the Rule Set is attempting to perform a field-to-field comparison without the second field name.	Run Consistency Checker to check data.
-2062	Field Name 1 is missing for a Field-to-Field comparison	A Rule in the Rule Set is attempting to perform a field-to-field comparison without the first field name.	Run Consistency Checker to check data.

Rules Management Data Errors

Code	Message	Explanation	Response
-2500	No rules management error	No error.	
-2501	DB error	Not in use.	(Should never see)
-2502	DB error Counter Insert	Data may be incorrect to add new Application Group.	Run Consistency Checker to check data.
-2503	DB error Counter Update	Data may be incorrect to add new Application Group.	Run Consistency Checker to check data.
-2504	DB error Counter Instance Insert	Data may be incorrect to add new Rule, Subscription, and so on.	Run Consistency Checker to check data.
-2505	DB error Counter Instance Update	Data may be incorrect to add new Rule, Subscription, and so on.	Run Consistency Checker to check data.
-2506	DB error Application Group Insert	Problem inserting Application Group. May be duplicate.	Run Consistency Checker to check data.
-2507	DB error message type insert (format)	Problem inserting Message Type. May not be valid format.	Run Consistency Checker to check data.
-2508	DB error message type insert	Problem inserting Message Type. May be duplicate.	Run Consistency Checker to check data.
-2509	DB error rule insert	Problem inserting Rule. May be duplicate.	Run Consistency Checker to check data.
-2510	DB error rule update	Problem updating Rule. Rule may not exist.	Run Consistency Checker to check data.
-2511	DB error argument op insert	Problem inserting operator for rule.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2512	DB error argument insert (Arg)	Problem inserting argument for rule.	Run Consistency Checker to check data.
-2513	DB error argument op update	Problem updating argument for rule.	Run Consistency Checker to check data.
-2514	DB error subscription list insert	Problem inserting subscription. May be duplicate.	Run Consistency Checker to check data.
-2515	DB error subscription master insert	Problem inserting subscription. May be duplicate.	Run Consistency Checker to check data.
-2516	DB error action insert	Problem inserting action.	Run Consistency Checker to check data.
-2517	DB error application group read	Problem retrieving application group. May have wrong name.	Run Consistency Checker to check data.
-2518	DB error message type read	Problem retrieving message type. May have wrong parameters.	Run Consistency Checker to check data.
-2519	DB error rule read	Problem retrieving rule. May have wrong parameters.	Run Consistency Checker to check data.
-2520	DB error subscription list read	Problem retrieving subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2521	DB error subscription master read	Problem retrieving subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2522	DB error subscription action read	Problem retrieving subscription action. May have wrong parameters.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2523	DB error message type read (message id)	Problem retrieving message type/format. May have wrong parameters.	Run Consistency Checker to check data.
-2524	DB error operator read	Problem retrieving operator. May have wrong parameters.	Run Consistency Checker to check data.
-2525	DB error operator type read	Problem retrieving operator type. May have invalid operator.	Run Consistency Checker to check data.
-2526	DB error argument read	Problem retrieving rule action. May have wrong parameters.	Run Consistency Checker to check data.
-2527	DB error counter read	Problem retrieving new application id. May have wrong parameters.	Run Consistency Checker to check data.
-2528	DB error counter instance read	Problem retrieving new ids for rule, subscription, and so on. May have wrong parameters.	Run Consistency Checker to check data.
-2529	DB error operation read	Problem retrieving argument info. May have wrong parameters.	Run Consistency Checker to check data.
-2530	DB error unreferenced operations	Arguments exist that are not used in a rule.	Run Consistency Checker to check data.
-2531	DB error argument update	Cannot update argument.	Run Consistency Checker to check data.
-2532	DB error subscription multi-read	Problem retrieving subscription info. May have wrong parameters.	Run Consistency Checker to check data.
-2533	DB error options not found	No options found for subscription action.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2534	DB error option delete	Cannot delete option.	Run Consistency Checker to check data.
-2535	DB error action resequence	Cannot resequence actions. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2536	DB error option resequence	Cannot resequence options. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2537	DB error delete all arguments failed	Cannot delete all arguments for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2538	DB error delete all list subscriptions failed	Cannot delete all subscriptions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2539	DB error delete all subscription masters failed	Cannot delete all subscriptions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2540	DB error delete all actions failed	Cannot delete all actions for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2541	DB error operation decrement	Cannot reduce the number of arguments using a specific operator.	Run Consistency Checker to check data.
-2542	DB error delete rule	Cannot delete rule. May have wrong parameters.	Run Consistency Checker to check data.
-2543	DB error delete arguments	Cannot delete argument. May have wrong parameters.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2544	DB error delete operation	Cannot delete argument information for a rule. May have wrong parameters.	Run Consistency Checker to check data.
-2545	DB error delete actions	Cannot delete action. May have wrong parameters.	Run Consistency Checker to check data.
-2546	DB error delete subscriptions	Cannot delete subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2547	DB error resequence multiple options	Cannot resequence options. May have invalid sequence parameters.	Run Consistency Checker to check data.
-2548	DB error option insert	Cannot insert option. May have wrong parameters.	Run Consistency Checker to check data.
-2549	DB error get max action	Cannot retrieve the maximum number of actions. May not have any actions.	Run Consistency Checker to check data.
-2550	DB error get max option	Cannot retrieve the maximum number of options. May not have any options.	Run Consistency Checker to check data.
-2551	DB error move action	Cannot resequence action. May have invalid sequence parameter.	Run Consistency Checker to check data.
-2552	DB error move option	Cannot resequence option. May have invalid sequence parameter.	Run Consistency Checker to check data.
-2553	DB error resequence multiple actions	Cannot resequence actions. May have invalid sequence parameters.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2554	DB error update action	Cannot update action. May have wrong parameters.	Run Consistency Checker to check data.
-2555	DB error update option	Cannot update option. May have wrong parameters.	Run Consistency Checker to check data.
-2556	DB error update subscription	Cannot update subscription. May have wrong parameters.	Run Consistency Checker to check data.
-2557	DB error option read	Cannot retrieve option. May have wrong parameters	Run Consistency Checker to check data.
-2558	DB error get max argument	Cannot retrieve the maximum number of arguments. May not have any arguments.	Run Consistency Checker to check data.
-2559	DB error application group update	Cannot update application name. May have wrong old name.	Run Consistency Checker to check data.
-2560	DB error get version failed	Cannot retrieve version information for import / export.	Run Consistency Checker to check data.
-2561	DB error update field name failed	Cannot update the old name to the new field name.	Run Consistency Checker to check data.
-2562	DB error get max boolean operator	Cannot retrieve the maximum number of Boolean operators. May have wrong parameters.	Run Consistency Checker to check data.
-2563	DB error boolean operator add failed	Cannot insert Boolean operator. May have wrong parameters.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2564	DB error boolean operator update failed	Cannot update Boolean operator. May have wrong parameters.	Run Consistency Checker to check data.
-2565	DB error application group delete failed.	Cannot delete application group.	Run Consistency Checker to check data.
-2566	DB error message type delete failed	Cannot delete message type.	Run Consistency Checker to check data.
-2567	DB error field function add / update	Error modifying field functions in the database.	Run Consistency Checker to check data.
-2568	DB error field instance add / update	Error modifying field instance in the database.	Run Consistency Checker to check data.
-2569	DB error field add / update	Error modifying field in the database.	Run Consistency Checker to check data.

General Rules Management Errors

Code	Message	Explanation	Response
-2600	Invalid application group parameters	Invalid application group name.	Check passed-in application group name.
-2601	Error application group already exists	Cannot add application with duplicate name.	Check passed-in application group name.
-2602	Error application group does not exist	Invalid application group name.	Check passed-in application group name.
-2603	Invalid message type parameters	Invalid application group / message type pair.	Check passed-in application group / message type name.
-2604	Error message type already exists	Application group already has the message type.	Check passed-in application group / message type name.
-2605	Error message type does not exist	Invalid application group / message type pair.	Check passed-in application group / message type name.
-2606	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	Invalid rule parameters	Invalid application group / message type / rule name.	Check passed-in parameters.
-2608	Error rule name already exists	Application group / message type pairs cannot have duplicate rule names.	Check passed-in parameters.
-2609	Error rule name does not exist	Invalid application group / message type / rule name.	Check passed-in parameters.

Code	Message	Explanation	Response
-2610	Invalid operator parameters	Invalid operator ID.	Check passed-in parameter.
-2611	Invalid argument parameters	Invalid parameters to create / update / retrieve argument.	Check passed-in parameters.
-2612	Invalid subscription parameters	Invalid parameters to create / update / retrieve subscription.	Check passed-in parameters.
-2613	Error subscription name already exists	Subscription names cannot be duplicated within a rule.	Check passed-in parameters.
-2614	Error subscription name does not exist	Application group / message type / rule name / subscription name not found.	Check passed-in parameters.
-2615	Invalid action parameters	Invalid parameters to create / update / retrieve action.	Check passed-in parameters.
-2616	Error action does not exist	Application group / message type / rule name / subscription name / action name not found.	Check passed-in parameters.
-2617	Invalid option parameters	Invalid parameters to create / update / retrieve action	Check passed-in parameters.
-2618	Error during conversion	Conversion of static argument value failed.	Check passed-in parameters. Run Consistency Checker.
-2619	No more actions	Not error unless returned from NNRMgrGetFirst Action.	Subscription must have at least one action.
-2620	No more operators	Not an error.	

Code	Message	Explanation	Response
-2621	No more arguments	Not error unless returned from NNRMgrGetFirst Argument.	Rule must have at least one argument.
-2622	Invalid rules management object passed in	Must call NNRMgrInit() before calling any other functions.	Call NNRMgrInit() prior to calling any other functions.
-2623	Feature not implemented	Feature is not implemented at this time.	
-2624	Argument does not exist	Invalid parameters to update/retrieve argument.	Check passed-in parameters: AppGrp MsgType RuleName ArgSeq Fields Operator
-2625	Operation does not exist	Invalid parameters to update/retrieve argument information.	Check passed-in parameters: AppGrp MsgType RuleName ArgSeq Fields Operator
-2626	Unknown operator type	Operator may be invalid.	Check passed-in parameters.
-2627	No more subscriptions	Not really error unless returned from NNRMgrGetFirst Subscription.	Rule must have at least one subscription.
-2628	No more rules	Not an error.	

Code	Message	Explanation	Response
-2629	Action does not exist	Invalid parameters to update/retrieve action.	Check passed-in parameters: AppGrp MsgType RuleName SubName ActSeq
-2630	Option does not exist	Invalid parameters to update/retrieve option.	Check passed-in parameters: AppGrp MsgType RuleName SubName ActSeq OptSeq
-2631	App id corrupted	Data for Application Group may be incorrect.	Run Consistency Checker to check data.
-2632	Msg id corrupted	Data for Message Type may be incorrect.	Run Consistency Checker to check data.
-2633	No more options	Not really error unless returned from NNRMgrGetFirst Option.	Action must currently have at least one option.
-2634	Export app name failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2635	Export message name failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2636	Export rule failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-2637	Export argument failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2638	Export subscription failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2639	Export action failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2640	Export option failed	Export failed during retrieval, encoding, or writing to file.	Run Consistency Checker to check data.
-2641	No more messages	Not really an error.	
-2642	No more applications	Not really an error.	
-2643	Error reading import file	Import failed to read from file.	Check file. Recreate file by exporting again.
-2644	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	Invalid import/export type	Can only import/export Rules components.	Should never see this error.
-2646	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.

Code	Message	Explanation	Response
-2647	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	Memory allocation failure	Cannot allocate memory.	Shut down excess items. Restart import/export.
-2649	Error importing argument	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2650	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	Error importing action	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2652	Error importing option	Import failed during reading of file, decoding, or writing to database.	Check file. Run Consistency Checker to check data.
-2653	Unsupported version of database	Can only export and import to version 4.1 databases.	Check version of NEONRules.
-2654	Decoding failure	Cannot decode line in file.	Export File may be corrupt. Recreate file by exporting again.
-2655	Cannot add permission if not owner	Rule old owner may not be a valid user of the current database.	Check database users.
-2656	No permission to read	Cannot read permission. Read permission not granted.	Assign permissions to rules.

Code	Message	Explanation	Response
-2657	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	Permission list read failure	Cannot read permission list.	Run Consistency Checker to check data.
-2659	No more permissions	Not really an error.	
-2660	Error exporting version	Cannot retrieve version for export. Can only export from version 4.0 and higher.	Check install.
-2661	Error exporting permissions	Cannot export rule permissions.	Run Consistency Checker to check data.
-2662	Invalid field name parameter	The field name provided is invalid.	Check parameters to function call.
-2666	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	Invalid non-numeric date/time value in argument	Bad format of static date/time value.	Check input parameter.
-2668	Invalid year in argument	Bad format of static date/time value.	Check input parameter.
-2669	Invalid month in argument	Bad format of static date/time value.	Check input parameter.
-2670	Invalid day in argument	Bad format of static date/time value.	Check input parameter.
-2671	Invalid hour in argument	Bad format of static date/time value.	Check input parameter.

Code	Message	Explanation	Response
-2672	Invalid minute in argument	Bad format of static date / time value.	Check input parameter.
-2673	Invalid second in argument	Bad format of static date / time value.	Check input parameter.
-2674	Unbalanced quotes in expression after	Invalid Boolean expression; quotes must be balanced.	Check input expression parameter.
-2675	Invalid Rules	Operator in expression in Invalid Rules operator.	Check the Operator list for spelling / case.
-2676	Expression missing Rules Operator	Rules expression must have a Rules Operator.	Check input expression parameter.
-2677	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	Unbalanced parentheses in expression	Parentheses must be balanced in Rules expression.	Check input expression parameter.
-2679	Expected terminal in expression	Expression ended incorrectly.	Check input parameter.
-2680	Arguments must be active for NEONet 4.0+	Arguments can no longer be Inactive.	Change input expression parameter.
-2681	Must Use NNR MgrUpdateExpression to perform update	Cannot use NNRMgrAddArgument unless all arguments are ANDed together.	Use NNRMgrUpdate Expression.
-2682	Trailing characters found in expression	Extra characters in the expression.	Make sure you are using '&' and '|' for Boolean operators.
-2683	Missing operand in boolean expression before / after	Two Operands are required around a Boolean operator.	Check input expression parameter.

Code	Message	Explanation	Response
-2684	Cannot delete item if not owner.	User not the owner of the sub/rule Cannot delete.	Delete as owner.
-2685	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	Invalid component type as parameter	Invalid component type parameter.	Check component type - input parameter.
-2687	Invalid or missing parameter	May have invalid parameter.	Check passed in parameters, for example, NULL values.
-2688	Invalid or missing change owner parameter	May have invalid parameter.	Check passed in parameter.
-2689	Invalid or missing component owner parameter	May have invalid parameter.	Check passed in parameter for NULL value.
-2690	Subscription list read failure	Failure reading subscription list.	Run Consistency Checker to check data.
-2691	Rule list read failure	Failure reading rule list.	Run Consistency Checker to check data.
-2692	Error importing permission	Error importing permission.	Check file. Run Consistency Checker to check data.
-2693	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	Invalid option value for putqueue MQS_FORMAT option	Option can be only 8 characters long.	Change the parameters sent to NNRMgrAdd Option or NNRMgr UpdateOption.

Code	Message	Explanation	Response
-2695	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	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	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	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 required formats.
-2699	Invalid integer static comparison value.	For integer comparison values, no non-numeric characters are allowed except for a (+/-) sign as the first character. Decimal point is not allowed.	Check input to Argument or Expression APIs.
-2700	Integer static comparison value out of valid range.	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.

Code	Message	Explanation	Response
-2701	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	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	Float static comparison value out of valid range.	Valid FLOAT values include a whole number in the integer range for the platform used, -2.1billion to about 2.1 billion, and a decimal mantissa with a maximum of 31 digits.	Check input into Argument or Expression APIs.
-2704	Static comparison value too long.	Static comparison values cannot exceed 64 characters plus a terminating NULL.	Check input into Argument or Expression APIs.
-2705	Cannot delete all rules and subscriptions in application group.	The user might not have permissions for all the rules and subscriptions in the application group.	Check permissions for rules and subscriptions. Only the owner can delete them.
-2706	Cannot delete all rules and subscriptions in message type.	The user might not have permissions for all the rules and subscriptions in the message type.	Check permissions for rules and subscriptions. Only the owner can delete them.
-2707	Error linking subscription to rule. Subscription does not exist.	Subscription was not imported.	See error message as to why the subscription was not imported.

Code	Message	Explanation	Response
-2708	Error importing expression.	Malformed expression or problem in the database.	Review the expression and run Consistency Checker on the database.
-2709	Error. -O flag is not supported in pre 4.10 versions. The -o flag is used instead.	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 GUI or Management APIs prior to importing.
-2710	Unsupported version of import file.	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 MQSeries Integrator V1.1 NNCrypt utility. Check the version of NNRie used to create the export file.
-2711	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 MQSeries Integrator V1.1 NNCrypt utility. Check the version of NNRie used to create the export file.
-2712	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 MQSeries Integrator V1.1 NNCrypt utility. Check the version of NNRie used to create the export file.

Code	Message	Explanation	Response
-2713	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 it contains the data required.
-2714	Argument failed to parse	Syntax problem with one of the arguments in the expression.	Check the syntax.
-2715	Invalid argument syntax.	Syntax or logic problem with one of the arguments in the expression.	Check the syntax.
-2716	Error adding field function	Problem adding information to the database.	Run Consistency Checker to check data and check the NNSYrfStatusLog.
-2717	Invalid field instance	Field instance invalid in context it was used.	Check instance references in the expression. Possibly the field expression does not have the required field instance of [*} contained in it.
-2718	Field function missing the field	Field function missing required field name.	Change expression to add the field name.
-2719	Sibling function invalid in this context.	Lookup ability for one field to determine the instance for another field must use MIN or MAX field functions.	Change the Sibling argument to use the MIN or MAX field functions.
-2720	Invalid group for the operation or function	Operators or function metadata is incorrect.	Run Consistency Checker to check data.
-2721	Invalid data types; must match each other	Data types of the field functions must match the data types of the operators.	Change the data types of the operators or field functions so they match.

Code	Message	Explanation	Response
-2722	Sibling operation invalid in this context	Lookup ability for one field to determine the instance for another field must use operators with the = operator.	Change sibling argument to use operators with the = operator.
-2723	Invalid field function	Field function in the expression invalid.	Check function name against list of valid field functions.
-2724	Comparing all instances of two field is invalid.	Ability to see is any instance of a field is equal to another is not a feature in this version.	Rearrange your expression to compare against a static value instead.

Permission Data Errors

Code	Message	Explanation	Response
-5500	No NEONRules database error	No error.	
-5501	Get next id insert error	Error getting new ids for user/permission.	Run Consistency Checker to check data.
-5502	Get next id update error	Error getting new ids for user/permission.	Run Consistency Checker to check data.
-5503	Node does not exist	Must run on valid 4.1 database with node data saved.	Check installation.
-5504	Hierarchy does not exist	Must run on valid 4.0 database with hierarchy data saved.	Check install. Run Consistency Checker to check data.
-5505	Component add failure	Cannot add rule component to permission system; may be duplicate.	Run Consistency Checker to check data.
-5506	Component load failure	Cannot retrieve rule component information from permission system; may not exist.	Run Consistency Checker to check data.
-5507	Delete component failure	Cannot delete rule component information from permission system; may not exist.	Run Consistency Checker to check data.
-5508	Unable to determine user	Permission user not a valid database user.	Run Consistency Checker to check data.
-5509	Unable to find user in database	Permission user not a valid database user.	Run Consistency Checker to check data.
-5510	Unable to find user in NEONRules	Permission user not a valid permission user.	Run Consistency Checker to check data.

Code	Message	Explanation	Response
-5511	Unable to add user to NEONRules	Cannot add permission user. May not be a valid database user.	Run Consistency Checker to check data.
-5512	Unable to add permission	Cannot add permission - may be a duplicate.	Run Consistency Checker to check data.
-5513	Unable to find permission	Cannot find permission. May have invalid parameters.	Run Consistency Checker to check data.
-5514	Unable to read permission	Cannot retrieve permission. May have invalid parameters.	Run Consistency Checker to check data.
-5515	Unable to update permission	Cannot update permission. May have invalid parameters.	Run Consistency Checker to check data.
-5516	User is not a valid user of the database instance	Permission user not a valid database user.	Run Consistency Checker to check data.
-5517	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	Unable to delete the permission set	Invalid parameters to delete permission set for a user/rule pair.	Run Consistency Checker to check data.
-5519	No permissions were found	Indicates no more permissions to read for rule or subscription.	Rule or subscription must have at least two permissions.
5520	Component update failure	Cannot update permission.May have invalid parameter.	Run Consistency Checker to run data.

General Permission Errors

Code	Message	Explanation	Response
-5000	No Errors	No error.	
-5001	Next id invalid parameters	Invalid parameters to get new user/component id for permission system.	Check passed-in parameters.
-5002	Update permission invalid parameters	Invalid parameters to update permission.	Check passed-in parameters.
-5003	Get node invalid parameters	Invalid parameters to retrieve node information.	Check passed-in parameters.
-5004	Get hierarchy level invalid parameters	Invalid parameters to retrieve hierarchy level information.	Check passed-in parameters.
-5005	Get hierarchy invalid parameters	Invalid parameters to retrieve hierarchy information.	Check passed-in parameters.
-5006	Add component invalid parameters	Invalid parameters to add component to permission system.	Check passed-in parameters.
-5007	Load component invalid parameters	Invalid parameters to retrieve component from permission system.	Check passed-in parameters.
-5008	Delete component invalid parameters	Invalid parameters to delete component from permission system.	Check passed-in parameters.
-5009	Load user invalid parameters	Invalid parameters to retrieve user from permission system.	Check passed-in parameters.
-5010	Add user invalid parameters	Invalid parameters to add user to permission system.	Check passed-in parameters.

Code	Message	Explanation	Response
-5011	Add permission invalid parameters	Invalid parameters to add permission to permission system.	Check passed-in parameters.
-5012	Load permission invalid parameters	Invalid parameters to retrieve permission from permission system.	Check passed-in parameters.
-5013	Adding permission that already exists	Duplicate permissions not allowed for user/component/permission.	Check passed-in parameters.
-5014	Changing user invalid parameters	Invalid parameters to change the owner for a certain component.	Check passed-in parameters.
-5015	Deleting permission set invalid parameters	Invalid parameters to delete all permissions for a user/component.	Check passed-in parameters.
-5016	Cannot add permission if not owner	User is not the owner of the component. Cannot add/update permission.	Add as owner of component.
-5017	No permission to read	Read permission not granted to PUBLIC or User.	Grant read permission for component.
-5018	Permission list read failure	Cannot read permission list.	Run Consistency Checker to check data.
-5019	No more permissions	Indicates no more permissions to read for rule or subscription.	Rules and Subscriptions must have at least two permissions.
-5020	No more components.	Not really an error.	
-5021	No permission to update	Update permission not granted to PUBLIC or User.	Grant update permission for component.

Code	Message	Explanation	Response
-5022	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 rule 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

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and / or changes in the product(s) and / or the program(s) described in this publication at any time without notice.

Any references in this document to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England,
SO21 2JN.
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. therefore, the results obtained in other operating environments

may vary significantly. some measurements may have been made on develoment-level systems and there is no guarantee that these measurements will be the same on generally available systems. furthermore, some mesurements may hve been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information includes examples of data and reports used in daily business operations. To illustrate them as completley as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

this information contains sample application programs in source language, which illustrate programming techniques on varius operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX	CICS	DB2
DB2 Universal Database	developerWorks	
Everyplace	FFST	First Failure Support Technology
IBM	IMS	IMS/ESA
iSeries	Language Environment	MXSeries
MVS	NetView	OS/400
OS/390	pSeries	RACF
RETAIN	RS/6000	SupportPac
Tivoli	VisualAge	WebSphere
xSeries	z/OS	zSeries

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.

Pentium is a registered trademark of Intel.

UNIX is a registered trademark in the United States andother countries licensed exclusively through The Open Group.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Index

Symbols

A

C

D

E

G

H

I

L

M

N

O

P

R

S

V

Sending your comments to IBM
Rules and Formatter Extension for WebSphere Message Broker for Multi-platforms
System Management Guide

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:

  IBM United Kingdom Laboratories
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN

- By fax:
  - From outside the U.K., use your international access code followed by 44 1962 870229
  - From within the U.K., use 01962 816151

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