MQSeries® Integrator

Programming Reference for NEONFormatter™

Version 1.0

SC34-5507-00

Note: Before using this information and the product it supports, be sure to read the general information under "Notices" on page 359.

.

.

First edition (January 1999)

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

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

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

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

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

Contents

Chapter 1

Introduction

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

This document contains two main sections: NEONFormatter APIs and NEONFormatter Management APIs.

Product Documentation Set

The MQSeries Integrator documentation set includes:

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

Note

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

Documentation Conventions

Tip

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

Note

Notes point out useful extra information.

WARNING!

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

Supported Platforms and Compilers

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

Chapter 2

Formatter Overview

What is Formatter?

NEONFormatter is one of two components of the MQSeries Integrator middleware product. The other component is NEONRules. NEONFormatter has two main functions: parsing and reformatting.

- Parse means to separate an input message into individual fields.
- Reformat means to transform an input message into an output message with a different format.

NEONFormatter is packaged as a library of C++ objects that have public functions that constitute the API (Application Programming Interface) or SDK (Software Development Kit). Application developers develop applications that invoke public Formatter functions to parse and reformat messages.

NEONFormatter uses format definitions that describe how to parse an input message and how to format an output message. Format definition data resides in a relational database. Users build and modify format definitions using one of two methods: the Formatter graphical user interface (GUI) tool or the Formatter management API functions.

The Formatter GUI allows users to populate screens with format definition data and store the information in a relational database.

Formatter management API functions are a set of C functions that create format definition data in a relational database. Users can write their own applications that call the management API functions to build format definitions.

Two executables, apitest and msgtest, are delivered with NEONFormatter. These executables show how to invoke the public functions and serve as tools for validating format definitions. apitest parses an input message and displays a hierarchical representation of the parse tree. msgtest reformats an input message into an output message.

The NEONFormatter Consistency Checker program checks the correctness of the format definition data in the relational database. As users build and maintain format definition data, they should run the Consistency Checker periodically to insure the integrity of their data.

NNFie is a command line tool that allows users to export format definitions from a database to an export file, and to import from the export file into a database. NNFie can import data from a MQSeries Integrator 1.0 export file into a MQSeries Integrator 1.0 database. NNFie exports data from a 1.0 database only.

The NEONFormatter GUI tool has its own import/export function as well. This function uses an export file with a format different from the one used by NNFie.

Format Structure

To format an input message into an output message using NEONFormatter, you need to create an input format that describes how to parse the input message and an output format that describes how to format the output message. When you maintain format definition data through the NEONFormatter GUI tool or the NEONFormatter management API functions, you will see that formats are built from components.

Formatter Components

Component	Description
Literals	See *Literals* on page 274.
Input Controls	See *Input Controls* on page 7..
Output Controls	See *Output Controls* on page 15.
Output Operations and Output Operation Collections	See *Output Operations* on page 20. See *Output Operation Collections* on page 23.
Fields	See *Input Format Structure* on page 6.
Flat Input Formats, Compound Input Formats, Flat Output Formats, and Compound Output Formats	See *Input Format Structure* on page 6.
User-Defined Types	See *User-Defined Data Type Management APIs* on page 203.

Input Format Structure

The simplest kind of input format is a flat input format. A flat input format contains a set of fields. Fields are defined by name and description. Within an input format, fields are associated with input controls. (Within an output format, fields are associated with output controls.)

A flat input format represents a message with a sequence of fields that occurs once without repetition, and the sequence may be ordinal, where fields appear in a specified order in the input message or random, where fields can appear in any order in the message. If you have a message which has a portion that repeats, you must build a compound input format to represent the message. Formatter can handle multiple levels of nesting (repetitions within repetitions of components).

Flat Input Formats

A flat input format is composed only of fields: named divisions of data each with a data type and a value. A flat input format cannot include another format as a component. Flat formats contain a set of defined fields and the associated information needed to parse or format them (input or output controls, depending on whether the format is for input or output). A flat input format has two properties: ordinal or random field order and format termination.

Properties of Component Fields of an Input Format

When you insert a field into a flat input format, you also associates an input control with the field. The input control is what describes the characteristics of the field data in detail. The same field can be inserted into different input formats with different input controls. For example, field F1 in input format IFF1 is variable length string data delimited by a comma. And field F1 in input format If F2 is a fixed length, 6-byte IBM packed decimal field.

Input Controls

An input control defines how to find the beginning and end of the field data, and how to interpret its value.

Input Control Types

Fields always contain data. The data may be of fixed or variable length. The data can have either a constant or variable value.

The data may be accompanied by a tag, which is an alternate identifier for a field. Formatter can retrieve the data for a field using either its name or tag.

Field data can be accompanied by a length value, which specifies the length of the field data in bytes.

The input control type specifies the form in which the data appears in an input message.

Type	Description
Data Only	Field data has a variable value and has no corresponding tag value or length in bytes value.
Tag and Data	Field data has a variable value and an associated tag value. The information appears in the input message in the order: tag followed by data.
Tag, Length and Data	Field data has a variable value, an associated tag value, and an associated length in bytes value. The information appears in the input message in the order: tag, followed by length, followed by data.

Type	Description
Length and Data	Field data has a variable value, and a length in bytes indicated by a length in bytes value. The information appears in the input message in the order: length followed by data.
Repetition Count	Field data contains a number that should be used as the count of the repetitions of the next repeating component that appears in the message. (This implies that the flat format that this field appears in is embedded in a compound format with a repeating component.)
Literal	Field data has a constant value.
Length, Tag and Data	Field data has a variable value, an associated tag value, and an associated length in bytes value. The information appears in the input message in the order: length, followed by tag, followed by data.
Regular Expression	Field data in the input message will conform to a string pattern defined by a regular expression. (The syntax of regular expressions is described in detail in the section, “Regular Expression Syntax”.).

Regular Expression Syntax

Regular Expression (RE) input controls express rules for string pattern matching. Instead of direct character-by-character matches, the input control value is interpreted as a regular expression to match the input. String-matching capabilities for this feature comply with the POSIX 1003.2 standard for regular expressions.

You can only build REs for the String data type. Within REs, only printable string characters are valid.

Note

RE matches are significantly slower than direct string matches. Be sure to build efficient REs.

Several rules apply for REs:

Ordinary characters (not specially defined for REs) act as one-character REs to match themselves. For example, if your RE is “X” in a parse control for a repeating field and your message is “X,Y,Z”, you will get a match on the first repeating field.

Backslashes (\) followed by special RE characters act as one-character REs that match the special character. Periods (.), asterisks (*), left square brackets ([), and backslashes (\) are special unless they are within square brackets (see below).

Carets (^) and dollar signs ($) are not supported. Do not use them in REs without preceding them with a backslash character (\).

Periods (.) act as one-character REs that match any character except a new line character. For example, if your RE is “. “in a parse control for a field and the

contents of the field is “This is a sample”, you won’t get a match for the field (no space following the sentence).

Non-empty strings of characters enclosed in square brackets ([]) act as one-character REs that match any one character in the enclosed string. “[a]” searches for the letter “a” in a fields contents.

The minus (-) character can be used to indicate a range of consecutive characters. For example, “[0-9]” is equivalent to “[0123456789]”.

Note that the right square bracket (]) does not terminate such a string when it is the first character within it (after an initial caret, if any). For example “[]a-f]” matches a right square bracket or one of the letters “a” through “f” inclusive. Also note that the four special characters (“.”, “*”, “\”, and “[“) represent themselves within the square brackets, so “[*]” searches for an asterisk within field contents.

A one-character RE followed by an asterisk (*) matches zero (0) or more occurrences of the RE. A one-character RE followed by a plus sign (+) matches zero (0) or one occurrence of the RE. If there are multiple strings matching the RE, the longest leftmost string permitting a match is chosen.

A one-character RE followed by {m}, {m,}, or {m,n} matches a range of occurrences of the one-character. m and n must be non-negative integers less than 256. {m} matches exactly m occurrences. {m,} matches at least m occurrences. And {m,n} matches any number of occurrences between m and n inclusive. If a choice exists, the RE matches as many occurrences as possible.

For example, “a{3,}” matches 3 or more concatenated “a” characters. This could also be done by REs of “aaa+” or “aaaa*”. 7. If REs are concatenated, the merged RE matches the concatenation of the strings matched by each component of the RE. For example, “XY” matches strings containing those two letters side-by-side.

A RE enclosed between parentheses (“(“,”)”) is a RE that matches whatever the non-parenthesized RE matches. There is no difference between parenthesized and non-parenthesized REs. This is useful when using a complex RE with the + or * operator as in (AB)+ which matches AB, or ABAB, or ABABAB...

A pipe (|) character indicates an “or”, so “(RE1|RE2)” matches either RE1 or RE2.

Optional and Mandatory Input Control Properties

An input control can be specified as either optional or not optional. When an input control is defined as not optional (mandatory), Formatter must be able to parse the field data according to its input control information. The data can be empty (zero length), but it must be able to be parsed successfully. If the data is not valid for the data type, or a delimiter, tag or length value cannot be found, then the parse will fail.

If the input control is defined as optional, Formatter continues parsing without error if it cannot successfully parse the corresponding field.

Building an Input Control that Specifies How to Parse a Field

The Formatter input control screen has three definitions: DATA, LENGTH and TAG. You must specify information required to parse the data portion of a field, since all fields have data. Depending on whether the field has a tag or length portion (based on the input control type), you will also need to specify information required to parse the tag and length portions of the field.

Data Portion of an Input Control

In the data section of the input control are eight definitions that describe how to parse and interpret the value in the data portion of a field.

Type	Description
Type	Specifies the data type of the field. Formatter supports a set of native data types, for which it knows how to interpret data values and convert data values from one native type to another. The set of data types are described in the section, “Supported data types”. Some of the supported data types imply fixed length data: (Endian types and date/time types). The other data types imply that the data can have variable length. If the length of the data is variable, Formatter must determine the length of the data. There are two ways to specify this: 1) Choose an input control type of “Length and data”, “Tag, length and data”, or “Length, tag and data”, indicating that the field contains an associated length in bytes value. 2) Choose a termination for the data (see “Termination” entry in this table).
Base Type	This applies only to data types of “date/time”. The user needs to specify the underlying data type (Base Type) of the data as “String”, “Numeric”, or “EBCDIC”, specifying a base data type and format properties of an input control. (See “Supported data types”.)
Format	This applies only to data types of “date/time”. The user needs to specify the date/time format string to which the data must conform. (See “Supported data types”.)
Year cutoff	Year cutoff value to indicate how to convert Two-digit years to four- digit. (See section, “Specifying a year cutoff value”.)
Termination	This specifies how to find the end of the data. The choices are: Delimiter (data is terminated by a literal delimiter); Exact length (data has a fixed length in bytes); Minimum Length + Delimiter (data is of at least a minimum length in bytes followed by a literal delimiter); Minimum Length + White Space (data is of at least a minimum length in bytes followed by a single white space); Not Applicable (data ends at the end of the message, or just before the format terminator of the flat format that contains the field); White Space Delimited (data is terminated by a single white space).

Type	Description
Length	Specifies the length in bytes of a fixed length piece of data, or the minimum number of bytes to parse if the termination of the variable length data is “Minimum Length + Delimiter” or “Minimum Length + White Space”.
Delimiter	Specifies the delimiter if the termination of the data is “Delimiter” or “Minimum Length + Delimiter”.
Decimal Loc	If the type of the data is one of the IBM types (packed, signed packed, zoned, or signed zoned), specifies the implied decimal location within the data. Only a positive value is allowed and it must not be more than the number of digits the field can hold. The value determines how many digits are to the right of the decimal location when Formatter interprets the message data. For example, an input value of 12345 with a decimal location of two is interpreted as 123.45. (See “Supported data types”.)
Literal	Only applies if the input control type is “Literal” or “Regular Expression” (the title in the GUI screen changes to “Regular Expression” in this case). The user should choose an existing literal.

Specifying a year cutoff value

The internal application functions of MQSeries Integrator use date-time information for archiving, time stamping, logging, etc. These functions use four-digit year notation or use Universal Time Coordinated (UTC) for time stamps and, therefore, are Year 2000 compliant.

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

For an input control that specifies a data type of custom date / time with a two-digit year format string, specify a “year cutoff” value, which tells Formatter how to convert the two-digit year date value to a four-digit year date value. To perform this conversion, Formatter compares the year value of the input data to the specified Year Cutoff value and assigns the century designation as required. That is, based on the comparison, Formatter converts the year value “XX” to “20XX” (21st century year) or “19XX” (20th century year) as appropriate.

The year cutoff algorithm is as follows:

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

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

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

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

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

Tag Portion of an Input Control

In the tag section of the input control, there are five fields. These five fields define how to parse and interpret the value in the tag portion of a field. This applies only if the input control type is Tag and Data, Tag, Length and Data or Length, Tag and Data.

Type	Description
Type	Specifies the data type of the tag portion of the field. Formatter supports a set of native data types, for which it knows how to interpret data values and convert data values from one native type to another. A tag may have any of the supported native data types except for the date/time data types. The set of data types are described in the section, *Supported Data Types* on page 26.
Tag Value	The user may specify an existing literal, or NONE. If the user specifies an existing literal, then Formatter will attempt to parse the exact literal value specified. If the user specifies NONE, then this indicates that the tag can take on different values, and Formatter will need to determine the value by parsing it.
Termination	This applies only to a Tag Value = NONE, where Formatter may have a variable tag value to parse. You can specify the following terminations: Delimiter (tag is terminated by a literal delimiter); Exact length (tag has a fixed length in bytes); Minimum Length + Delimiter (tag is of at least a minimum length in bytes followed by a literal delimiter); Minimum Length + White Space (tag is of at least a minimum length in bytes followed by a single white space;); Not Applicable (no termination specified because data type of tag is of fixed length); White Space Delimited (tag is terminated by a single white space).
Length	Specifies the length in bytes of a fixed length tag, or the minimum number of bytes to parse if the termination of the variable length tag is Minimum Length + Delimiter or Minimum Length + White Space.

Type	Description
Delimiter	Specifies the delimiter if the termination of the tag is Delimiter or Minimum Length + Delimiter. Delimiters in cross-platform formats should be composed of printable characters.

Length Portion of an Input Control

The length section of an input control contains four fields. These four fields define how to parse and interpret the value in the length portion of a field. This applies only if the input control type is Length and Data, Tag, Length and Data, or Length, Tag and Data.

Type	Description
Type	Specifies the data type of the length portion of the field. Formatter supports a set of native data types, for which it interprets and converts data values from one native type to another. A length may have only a data type that can contain an integer number: Endian types, IBM types, EBCDIC, Numeric and String. The set of data types are described in the section, *Supported Data Types* on page 26.
Termination	If the data type of the length value indicates a variable length value, then the user can specify the following terminations: Delimiter (tag is terminated by a literal delimiter); Exact length (length value has a fixed length in bytes); Minimum Length + Delimiter (length value is of at least a minimum length in bytes followed by a literal delimiter); Minimum Length + White Space (length value is of at least a minimum length in bytes followed by a single white space); Not Applicable (no termination specified because data type of length value is of fixed length); White Space Delimited (length value is terminated by a single white space).
Length	Specifies the length in bytes of a fixed length value, or the minimum number of bytes to parse if the termination of the length value is Minimum Length + Delimiter or Minimum Length + White Space.
Delimiter	Specifies the delimiter if the termination of the length value is Delimiter or Minimum Length + Delimiter.

Compound Input Formats

A compound input format is composed of other compound or flat format components. The components may or may not be repeating. Types of compound input formats are: Ordinal, Tagged Ordinal, and Alternative.

Compound Input Format Types

Type	Description
Ordinal	The component formats of the compound input format always appear in the same order.
Tagged Ordinal	The component formats of the compound input format always appear in the same order, and each component format is a tagged format. A tagged format is a format where the first field in the format is a literal.
Alternative	Alternative formats are special compound formats where only one format in a set of alternatives will apply to a message. For example, if an alternative format is named A, it may contain component formats B, C, and D. A message of format A may actually be of variation B, C, or D. Exactly one of the alternatives must apply or the entire alternative format does not apply. Formatter attempts to parse each alternative until it finds a successful match. The first successful match it finds is the one it chooses, so you have to be careful to construct your alternative format so that a more general alternative does not supersede a more specific alternative. If none of the alternatives apply, and the format is a mandatory component of the message, the parse fails.

Inserting a Component Format into a Compound Input Format

You may insert either a flat or a compound input format component into a compound input format. You can decide on two properties of the component:

Type	Description
Optional / not optional	The component format does not need to appear in the input message. The parse succeeds if the component is not successfully parsed; otherwise, the parse fails.
Repeating / not repeating	The component format repeats in the message. If the component repeats, the user chooses a “Repeat termination” that specifies a fixed number of repetitions, or a variable number of repetitions (The repetitions are terminated by a literal delimiter, or the repetition count is contained in another field in the message).

Output Format Structure

The simplest kind of output format is a flat output format. A flat output format contains a set of fields. The flat format represents a message with a sequence of fields that occurs once without repetition. If you have a message which has a portion that repeats, you must build a compound output format to represent the message. Formatter can handle multiple levels of nesting (repetitions within repetitions of components).

Flat Output Formats

A flat output format is composed only of fields. It cannot include another format as a component. Unlike flat input formats, it has no ordinal/random field order. Fields are always output in the order in which they are defined in the format. There is also no format termination property. A format termination may be constructed by inserting a literal field at the end of the format.

Properties of Component Fields of an Output Format

Property	Description
Output Control Name	The name of the output control that specifies how the field data is to be formatted.
Access Mode	Specifies how to access a field in the input message to acquire a field value for the output field. (This is important for fields that repeat. The access mode tells which instance of the repeating input field to access when getting the value for the next repeating instance of the output field.) See *Access Modes* on page 17..
Subscript	Applies only to the access mode “Access nth instance of field”. The subscript indicates which instance of the input field to access to acquire a field value for the output field. (0 indicates the first instance.) See *Access Modes* on page 17.
Input Field Name	This is used to map an input field to an output field with a different name. By default, input fields are mapped to output fields by matching names.

The same field can be inserted into different output formats with different output controls, access modes, and field mapping.

Output Controls

Output controls describe how to format the data in the output field. They specify how to get the starting value for an output field (search by name, tag, use a literal…), what data type transformation to perform, what formatting operations to perform on the data (formatting operations are defined by subordinate output operations associated with the output control), and whether or not to output a tag and/or a length in bytes value along with the data for the field.

Output formatting is performed in the following order:

1. A starting field value is generated (by accessing a field value from the input message, by using a literal, or by using the result of a calculation).
2. The field value is converted to its final output data type.
3. The formatting operations defined by the subordinate output operations are executed in the order in which they are defined. (Formatting operations include justifying and trimming data, converting the case of string data to upper or lower case, substituting values, performing substring operations, executing user exit functions or mathematical expressions, adding prefixes or suffixes, adjusting the width of the output field, and specifying the default value of the field.)

Output Control Types

There are two categories of output control types. The first category specifies how to retrieve the initial value of the output field, which can then be formatted.

Type	Description
Data Field Name Search	Map the output field to an input field by field name. (This makes use of access modes to determine which instance of the input field to access, and uses input field mapping if the user wants a different named input field to map to the output field.) See *Access Modes* on page 17.
Data Field Tag Search	Map the output field to an input field by tag value. (There is no use of access modes or input field mapping for this kind of search. Formatter looks for the first field in the input message that it can find with the specified tag value.)
Literal	Field value is a literal.
Calculated Field	Performs a calculation using the left and right operand fields to generate a value for the output field. Available calculations are Add, Subtract, Multiply and Divide. **Note**: It is recommended that you use the Mathematical expression operation rather than Calculated Field, which will become obsolete in a future release of Formatter.
Conditional Field	Mark field as to be output only if the Existence check field exists.

Type	Description
Rules Field	Rules Field enables you to create several different output controls for a single output field by integrating Rules with Formatter. Formatter can then use the boolean logic capabilities of the Rules Engine to express and evaluate the conditions for formatting a field. Using the Rules Field capability, you build different output controls for the same output field. This eliminates the need to create several output formats for a single input format. If no rule hits, output data is formatted according to the other settings in the Rules Field output control. Otherwise, the data is formatted according to the output control specified by the rule that hits. If multiple conflicting rules hit (multiple output controls are returned by Rules for the same output field), Formatter generates an error. See *Conditional Branching* in the ***MQSeries Integrator User Guide*** for details and how to use the Rules GUI. **Notes**: When you define rules for the output control, you cannot delete the fields used in the rules without first deleting the rules that use the fields. The Output Control type determines which fields require information. Fields containing Not Applicable or NONE do not require a value.

The second category of output control types mark the associated output fields as control fields. No data is output for fields with these control types. The fields control the behavior of other fields, or of the format as a whole.

Type	Description
Left Operand Field	Mark field as left operand. (Used for Calculated Fields.) **Note**: Use the Mathematical expression operation rather than Left Operand Field, which will become obsolete in a future release of NEONFormatter.
Right Operand Field	Mark field as a right operand. (Used for Calculated Fields.) **Note**: Use the Mathematical expression operation rather than Right Operand Field, which will become obsolete in a future release of NEONFormatter.
Existence Check Field	Mark field as an existence check field. (Used for Conditional Fields.)
Input Field Exists	The format that the associated field appears in should be output only if the associated input field exists.
Input Field Value =	The format that the associated field appears in should be output only if the associated input field value equals the specified value.

Access Modes

Each output field has an associated Access Mode. Access Modes define how Formatter accesses fields in the input message to generate fields in the output message. You select output field access modes and associated input field

names to tell Formatter how to map fields from the input message to fields in the output message.

The following table provides a description of each access mode supported in Formatter.

Definitions of Access Modes

Access Mode	Description
Not Applicable	Do not access any field instance.
Normal Access	Access the instance in the same repeating component as the current controlling field instance. If there is no controlling field, access the first instance. This behaves just like Access sibling instance.
Access with Increment	A field with this access mode is the controlling field for the repeating component (see *Controlling field*).
Access Using Relative Index	The first field in a repeating component that Formatter encounters with this access mode is the controlling field for the repeating component (see *Controlling field*). Any other field in the repeating component with this access mode behaves as if it has access mode Access sibling instance or Normal access (access the sibling of the controlling field).
Access nth instance of field	Access the nth instance (n = 0 means get the first instance) of the field in the input message.
Controlling field	This field is the controlling field for the repeating component. On each repetition, access the next field instance that is still a child of the current controlling field instance of the parent format. If there is no parent controlling field, the repetitions end with the last field instance from the input message.
Access current instance	Access the same field instance as on the previous access (the first access will get the first instance of the field).
Access next instance	Access the next field instance relative to the previous access.
Access parent instance	Access the instance that is the first ancestor of the current controlling field instance.
Access sibling instance	Access the instance in the same repeating component as the current controlling field instance. If there is no controlling field, access the first instance.

Output Operation Property

An output control usually specifies formatting operations (justification, trimming, default values, field width adjustments, etc.) to be applied to the output field data. This is done by associating an output operation with the output control. The output operation can be a collection of output operations of various types, or can be a single output operation.

Optional/Not Optional Property

An output control can be specified as either optional or not. Optional means Formatter should continue with the output message if the output field value has zero length. The output field may have zero length because there is no associated input field or the associated input field value has zero length and there is no default operation, user exit operation or math expression operation associated with the output field to generate data for it. These operations are called data producing operations.

If the field was not found in the input message or the input field value has zero length and the output control has an associated Default operation, the specified Default value is used.

By default, the output control is not optional, it is mandatory. Mandatory means if output field value has zero length (no mapped input field or input field value has zero length, no default, no math expression, and no user exit), the entire formatting operation fails.

Data type, Base Data Type, and Format Properties

This specifies data type information for the data portion of the output field. All supported native data types are allowed. Formatter converts the output field data into the specified data type before inserting it into the output message.

If the specified output data type is a date/time data type, specify a Base Data Type (underlying native data type of String, EBCDIC, or Numeric) and a Format (date/time format string).

Length Type, Tag Type, and Tag Before Length Properties

If a data type other than Not applicable is chosen for the Length type, Formatter outputs an associated length in bytes of the data in the output field, in the specified data type. Otherwise, no length value is output.

If a data type other than Not applicable is chosen for the Tag type, Formatter outputs an associated tag value in the specified data type. Otherwise, no tag value is output. The tag value is the same tag value as appeared for the corresponding field in the input message.

Normally, the output field components are output in the following order: length, followed by tag, followed by data. If “Tag before length” is selected, then the output field components is output in the following order: tag followed by length followed by data.

Input Tag Value Property

This field applies only if the output control type is Data Field Tag Search. This indicates that the starting value for the output field should be taken from the input field whose tag value is specified by Tag Value (a literal).

Calculation Property

This field applies only if the output control type is Calculated Field. The available calculation choices are: Add, Subtract, Multiply, and Divide.

Note

It is recommended that you do not use Left Operand Field, Right Operand Field and Calculated Field, which will become obsolete in a future release of MQSeries Integrator. Use the Mathematical Expression output operation instead.

Field Value Property

This field applies only if the output control type is Literal or Input Field Value =. For Literal, the Field Value is set to the particular literal that should be output for the field. For Input Field Value =, the Field Value is set to the value, allowing a comparison with the input field value.

Output Operations

Output Operations define the different formatting operations that can be performed on the data in an output field. Using operations, you can change the case of output data, perform mathematical expressions based on input field contents, extract s, and more.

Through the use of output operation collections, you can collect operations to perform them sequentially, so you could left justify and right trim a substring of the contents of an input field. The order in which these operations are defined in the collection reflects the order in which they are performed.

The available types of operations include: Case, Default, Prefix/Suffix, Justify, Length, Math Expression, Substitute, Substring, Trim, and User Exit.

Note

After applying a sequence of formatting operations, you may have a mixture of data in different data types in the data portion of your output field.

Output Operation Type	Description
Case	Case operations affect the case of the field data. You can convert data to all uppercase or all lowercase. The only valid data type for this operation is “String”.

Output Operation Type	Description
Default	Default operations provide a default value for a field. A default is generated if an input field does not exist in the input message or has a length of zero. Otherwise, the default operation has no effect. **Note**: A default operation negates all operations that precede it, because it generates a new value for the field.
Prefix / Suffix	Prefix / Suffix operations enable you to specify a literal for use as a prefix or suffix. Prefixes are added to the beginning of the data portion of an output field. Suffixes are added to the end of the data portion of an output field. For example, you may want to place a less-than sign (<) before and a greater-than sign (>) after the output data. This would require a Prefix of "<" and a Suffix of ">". Any defined literal may be used as a prefix or suffix. If a Prefix / Suffix is specified with the Null Action check, Formatter outputs the Prefix / Suffix in the output message even if the input field isn't present in the input message, or it has zero length.
Justify	Justify operations justify field data to the left, center, or right within the length of the field. Note that a justify operation has no effect on the data, unless the field width has been specified to be greater than the length of data. This field width adjustment is accomplished by including a "Length" operation after the "Justify" operation in the collection.
Length	Length operations adjust the width of an output field. If you define a length of 12 and the data is longer, it is truncated on the right. If it is shorter, you may define a pad character to fill in the empty space. If no pad character is specified, a space character is used. If there is no preceding Justify operation, numeric data (of data type Endian, Numeric, Decimal International, Decimal US, IBM types) is right justified and padded on the left. Other data is left justified and padded on the right. If there is a preceding Justify operation, data is justified and padded according to the parameters of the Justify operation.
Math Expression	Math Expression operations output a value resulting from an arithmetic expression. The expression can be built using arithmetic operators, constants, and input field values. (See section, "Math Expression Syntax" for the proper syntax of a math expression.) **Note**: A math expression operation negates all operations that precede it, because it generates a new value for the field.

Output Operation Type	Description
Substitute	Substitute operations enable you to define a list of input strings to substitute and the output strings to replace them. If no input string is found that exactly matches the data, the original input field value is left unchanged. **Note**: A substitute operation negates all operations that precede it, because it generates a new value for the field.
Substring	Substring operations enable you to extract a portion of an input string. For example, a starting position of 3 with a data length of 4 applied to a string field value of abcdefghi results in the value cdef. Only the String, Numeric, EBCDIC, and Binary data types can be used for substrings. **Note**: If you extract a substring longer than the data, you may specify a padding character. The resulting substring will include the data plus an appended sequence of padding characters to the specified length. If no padding character is specified, a space character is used.
Trim	Trim operations remove a defined trim character to the right or left of the output data.
User Exit	User Exits are external, user-created routines used to compute the value of an output field. If a function you need to perform is outside what the Formatter can currently do, you can write a C function to perform the task. See the ***MQSeries Integrator Programming Reference for NEONFormatter APIs*** for details about creating User Exit routines. **Note**: A user exit operation negates all operations that precede it, because it generates a new value for the field.

Math Expression Syntax

Using Mathematical Expression operations, you can output a value resulting from an arithmetic expression. The expression can be built using arithmetic operators, constants, and input field values.

Available Operators: + - * / ( ) and Unary -

Order of Operator Precedence: ( ) * / + -

Available Operands: Numeric Constants and Input Field Names. (Input field values should contain numeric data.)

Field names with spaces or underscores must be surrounded by single or double quotes. A field with the name Field_1 could be used as "Field_1" or 'Field_1' (but not "Field_1' or 'Field_1").

For example, if an input message is defined with fields InF1, InF2, and InF3 and an output message is defined with field OutF1. You could define a mathematical expression as part of an output control associated with output message field OutF1 as

InF1 + InF2 * -InF3

This expression is evaluated as InF1 + (InF2 *(- InF3)), based on the precedence rules.

Other expression examples include:

- InF1 + -InF2
- InF1 * 8
- InF1 * 9.3
- InF1 * -8
- InF1 * -9.3
- (InF1 + InF2) * 3 / InF3
- (InF1 * (InF2 + InF3) * 4

Output Operation Collections

Output Operation Collections enable you to group and sequence a set of output operations and / or other output operation collections. Operations are executed in the order in which they appear.

For example, if you want the contents of an input field to take a substring of the left-justified, right-trimmed contents of an input field, you need to LEFT_JUSTIFY, LENGTH, RIGHT_TRIM, and then SUBSTRING.

What makes sense when building a collection?

First, some of the output operations are data producing, and negate the previous effects of other operations, so it makes sense to insert one of them into the collection first, and not insert it again:

Output Operations	Description
Default	Generates a default value for the output field if there is no corresponding input field value.
Math Expression	Generates a value for the output field based on the evaluation of a math expression.
Substitute	Substitutes an input field value with an output field value based on a look up in a set of substitute values.
User Exit	Generates an output field value based on the execution of a user written function.

The remaining operations can be inserted into a collection as the first operation, or as some succeeding operation any number of times:

Output Operations	Description
Case	Converts data to uppercase or lowercase.
Prefix / Suffix	Prepends a prefix or appends a suffix to the data.
[Justify] Length	Adjusts the field width and justifies and pads, or truncates the data appropriately. A Justify operation always needs an immediately succeeding Length operation for it to have an effect on the data. A Length operation does not need an accompanying Justify operation. Here is how they work both together and separately: **Rule 1**: A Justify operation without a succeeding Length operation has no effect. **Rule 2**: A Length operation with no preceding Justify operation does the following: left justifies non-numeric data, and right justifies numeric data, using a space character to pad the data to the specified field width. **Rule 3**: A Length operation with a PRECEDING Justify operation does the following: justifies the data as specified by the Justify parameters (left, center, right), regardless of whether the data is numeric or non-numeric. The field is padded with the Justify operation's padding character. **Rule 4**: A Length operation with a SUCCEEDING Justify operation operates like Rule 2. (Justify has no effect; Length works as though there were no corresponding Justify operation.) **Rule 5**: Anytime a Length operation operates on data that is larger than the field width, it truncates the data on the right to the specified length.
Substring	Extracts a substring from the current value of the output field.
Trim	Trims characters from the current value of an output field.
Collection	Applies another collection of operations.

Compound Output Formats

A compound output format is composed of other compound or flat output format components. The components may or may not be repeating.

Types of Compound Output Formats

There are two types of compound output formats:

Compound Output Formats	Description
Ordinal	The component formats of the compound output format always appear in the order in which they are defined.
Alternative	Alternative formats are special compound formats where only one format in a set of alternatives will apply to a message. For example, if an alternative format is named A, it may contain component formats B, C, and D. A message of format A may actually be of variation B, C, or D. Exactly one of the alternatives must apply or the entire alternative format does not apply. When Formatter is processing an alternative output format, it attempts to find fields from the input message that match each alternative. The first successful match it finds is the alternative it chooses. If the alternative is a repeating component, Formatter orders the alternative instances in the same order in the output message that they appeared in the input message.

Inserting a Component Format into a Compound Output Format

You may insert either a flat output format or a compound output format component into a compound output format. You can choose values for two properties of the component:

Compound Output Format	Description
Optional / not optional	If the component is optional, the component format does not need to appear in the output message. The reformat succeeds if the component is not successfully constructed. If the component is mandatory and the mandatory fields and components of the component are not successfully built, the reformat fails.
Repeating / not repeating	The component format repeats in the message. If the component repeats, the user chooses a “Repeat termination of “Not Applicable” or “Delimiter”. “Not Applicable” means that repetitions are generated for as many corresponding repetitions are found in the input message. “Delimited” means the same as “Not Applicable”, with the exception that a specified literal delimiter is placed at the end of the repeating sequence in the output.

Supported Data Types

Data Type	Description
Not Applicable	No data type is assumed.
String	A string of standard ASCII characters. Note that non-printable characters are valid as long as there are in the ASCII character set. (EBCDIC characters outside the valid Ascii String range are not valid Ascii String characters. During a reformat from ASCII to EBCDIC if a character being converted is not in the EBCDIC character set the conversion results in a EBCDIC space (hexadecimal 40)).
Numeric	A string of numeric characters. Only the characters '0' – '9' are allowed. (No '+' or '-'.)
Binary Data	The Binary data type is used to parse any value and transform that value to an ASCII representation of the value internally in Formatter. The internal representation takes each byte of the input value and converts it into a readable form. An example of this is parsing a byte with the value (hexadecimal) 0x9C and transforming that into the internal ASCII representation of 9C, which is the hexadecimal value 3943. If this value is used in an output format with the output control's data type set to String, the value placed in the message is ASCII 9C. If this value is again placed in an output message with the data type Binary, the ASCII value is not printable and occupies one byte with the value of (hexadecimal) 0x9C. Conversely, an input value of ASCII 3B7A parsed with the string data type can be output using the Binary data type. The output value is (hexadecimal) 0x37BA and occupies 2 bytes in the output message. Valid characters that can be converted to Binary from the String data type are 0 through 9 and A through F. All other characters are invalid.
EBCDIC Data	A string of characters encoded using the EBCDIC (Extended Binary Coded Decimal Interexchange Code) encoding used on larger IBM computers.
IBM Packed Integer	Data type on larger IBM computers used to represent integers (>= 0) in compact form. Each byte represents two decimal digits, one in each nibble of the byte. The final nibble is always a hexadecimal 'F'. For example, the number "1234" is stored as a 3-byte value: "01 23 4F" (the number pairs show the hexadecimal values of the nibbles of each byte). The number "12345" is stored as a 3-byte value: "12 34 5F".

Data Type	Description
IBM Signed Packed Integer	Data type on larger IBM computers used to represent integers in compact form. This data type takes into account the sign (positive or negative) of a number. Each byte represents two decimal digits, one in each nibble of the byte. The final nibble is either a hexadecimal 'C' if the number is positive, or a hexadecimal 'D' if the number is negative. For example, the number "1234" is stored as a 3-byte value: "01 23 4C" (the number pairs show the hexadecimal values of the nibbles of each byte). The number "-1234" is stored as a 3-byte value: "01 23 4D".
IBM Zoned Integer	Data type on larger IBM computers used to represent integers (>= 0). Each decimal digit is represented by a byte. The left nibble of the byte is a hexadecimal 'F'. The right nibble is the hexadecimal value of the digit. For example, "1234" is represented as "F1 F2 F3 F4" (the number pairs show the hexadecimal values of the nibbles of each byte).
IBM Signed Zoned Integer	Data type on larger IBM computers used to represent integers. Each decimal digit is represented by a byte. The left nibble of each byte, except the last byte, is a hexadecimal 'F'. The left nibble of the last byte is a hexadecimal 'C' if the number is positive, and a hexadecimal 'D' if the number is negative. The right nibble of each byte is the hexadecimal value of the digit. For example, "1234" is represented as "F1 F2 F3 C4" (the number pairs show the hexadecimal values of the nibbles of each byte). "-1234" is represented as "F1 F2 F3 D4".
Little Endian 2	Two-byte integer where the bytes are ordered with the rightmost byte being the high order or most significant byte. For example, the hexadecimal number "0x0102" is stored as "02 01" (where the number pairs show the hexadecimal values of the nibbles of a byte).
Little Swap Endian 2	Two-byte integer where the two bytes are swapped with respect to a "Little Endian 2" value. For example, the hexadecimal number "0x0102" is stored as "01 02". Little Endian 4Four-byte integer where the bytes are ordered with the rightmost byte being the high order or most significant byte. For example, the hexadecimal number "0x01020304" is stored as "04 03 02 01" (where the number pairs show the hexadecimal values of the nibbles of a byte).
Little Swap Endian 4	Four-byte integer where the two bytes of each word are swapped with respect to a "Little Endian 4" value. For example, the hexadecimal number "0x01020304" is stored as "03 04 01 02".

Data Type	Description
Big Endian 2	Two-byte integer where the bytes are ordered with the leftmost byte being the high order or most significant byte. For example, the hexadecimal number “0x0102” is stored as “01 02” (where the number pairs show the hexadecimal values of the nibbles of a byte).
Big Swap Endian 2	Two-byte integer where the two bytes are swapped with respect to a “Big Endian 2” value. For example, the hexadecimal number “0x0102” is stored as “02 01”. Big Endian 4Four-byte integer where the bytes are ordered with the leftmost byte being the high order or most significant byte. For example, the hexadecimal number “0x01020304” is stored as “01 02 03 04” (where the number pairs show the hexadecimal values of the nibbles of a byte).
Big Swap Endian 4	Four-byte integer where the two bytes of each word are swapped with respect to a “Big Endian 4” value. For example, the hexadecimal number “0x01020304” is stored as “02 01 04 03”.
Decimal, International	Data type where every third number left of the decimal point is preceded by a period. The decimal point is represented by a comma. Numbers right of the decimal point represent a fraction of one unit. For example, the number “12345.678” is represented as “12.345,678”.
Decimal, U.S.	Data type where every third number left of the decimal point is preceded by a comma. The decimal point is represented by a period. Numbers right of the decimal point represent a fraction of one unit. For example, the number “12345.678” is represented as “12,345.678”.
Unsigned Little Endian 2	Like “Little Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Little Swap Endian 2	Like “Little Swap Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Little Endian 4	Like “Little Endian 4”, except that the value is interpreted as an unsigned value.
Unsigned Little Swap Endian 4	Like “Little Swap Endian 4”, except that the value is interpreted as an unsigned value.
Unsigned Big Endian 2	Like “Big Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Big Swap Endian 2	Like “Big Swap Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Big Endian 4	Like “Big Endian 4”, except that the value is interpreted as an unsigned value
Unsigned Big Swap Endian 4	Like “Big Swap Endian 4”, except that the value is interpreted as an unsigned value.

Data Type	Description
Date and Time	Based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS. See the first paragraph of each of the following Date and Time type descriptions for details on representing Date and Time components. Combined dates and times may be represented in any of the following list of data types: Numeric, String and EBCDIC.
Time	Based on the international ISO-8601:1988 standard time notation: HHMMSS where HH represents the number of complete hours that have passed since midnight (between 00 and 23), MM is the number of minutes passed since the start of the hour (between 00 and 59), and SS is the number of seconds since the start of the minute (between 00 and 59). Times are represented in 24-hour format. Times may be represented in any of the following list of data types. The list includes: Numeric, String and EBCDIC.
Date	DateBased on the international ISO-8601:1988 standard date notation: YYYYMNDD where YYYY represents the year in the usual Gregorian calendar, MN is the month between 01 (January) and 12 (December), and DD is the day of the month with a value between 01 and 31. Dates may be represented in any of the following list of data types. The list includes: Numeric, String and EBCDIC.

Data Type	Description
Custom Date and Time	Custom Date and Time enables users to specify different formats of dates, times, and combined dates and times. Date/Time formats are predefined by the Formatter installation and include the following: ■ variations in year (2 or 4 digit year representation: YY or YYYY) ■ variations in month – use of MN for a month number (01-12) or MON for three letter abbreviation (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC) ■ variations in the day of the month – use of a day of the month number, 01 - 31 (DD) ■ variations in hour – 12-hour or 24-hour representation (HH), with or without a meridian indicator (AM or PM) ■ variations in minutes, 00 - 59 (MM) ■ variations in seconds, 00 - 59 (SS) Custom Date and Times may be represented in any of the following list of data types: Numeric, String and EBCDIC **Note**: In version 1.0, users select from an a set of provided custom date/time formats. Users will not be able to create their own custom date/time formats.

How Formatter Converts Data Types

When Formatter reformats a message, it first converts all field data to an intermediate form: the string representation of the data. (The only exception is a data type of “Not Applicable”, where the data is not converted to an intermediate form.) Then the data is converted to its final output data type.

Value Ranges for Each Data Type and Intermediate Representation

This table shows the valid values allowed for data of each type and how the data is converted to its intermediate string representation.

Data Type	Source Data Value Range	Intermediate String Representation
Not Applicable	Any value, any length.	Source value is unchanged.

Data Type	Source Data Value Range	Intermediate String Representation
String	A string of characters of any length, in the native character encoding for the local machine (ASCII on Unix and Windows NT).	Source value is unchanged.
Numeric	A string of characters '0' – '9' (no '+' or '-') of any length, in the native character encoding for the local machine.	Source value is unchanged.
Binary Data	Any value, any length.	The binary string is converted to its string encoded hexadecimal form. For example, the binary string 01 23 AC (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal) is turned into the string: "0x0123AC" Note that the prefix '0x' is prepended to the data.
EBCDIC Data	A string of characters of any length, encoded in EBCDIC.	Each character of the EBCDIC string is converted to its equivalent native encoded value (ASCII on Unix and Windows NT). Characters in the EBCDIC code set that are not in the native code set are converted to a native encoded space character.
IBM Packed Integer	Maximum 16 byte value. Each nibble (except for the last nibble) may contain the hexadecimal value '0' through '9'. The last nibble contains a hexadecimal 'F'.	String that represents the number. For example, the packed integer value 12 34 5F (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal) becomes "12345".

Data Type	Source Data Value Range	Intermediate String Representation
IBM Signed Packed Integer	Maximum 16 byte value. Each nibble (except for the last nibble) may contain the hexadecimal value '0' through '9'.	The last nibble contains a hexadecimal 'C' for positive numbers and a hexadecimal 'D' for negative integers. String that represents the number. For example, the signed packed integer value 12 34 5C (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal) becomes "+12345". The signed packed integer value 12 34 5D becomes "-12345".
IBM Zoned Integer	The left nibble of each byte is a hexadecimal 'F'. The right nibble of each byte is a hexadecimal '0' through '9'.	String that represents the number. For example, the zoned integer value F1 F2 F3 F4 F5 becomes "12345".
IBM Signed Zoned Integer	The left nibble of each byte except for the last byte is a hexadecimal 'F'. The left nibble of the last byte is a hexadecimal 'C' if the number is positive, and a hexadecimal 'D' if the number is negative. The right nibble of each byte is a hexadecimal '0' through '9'.	String that represents the number. For example, the signed zoned integer value F1 F2 F3 F4 C5 becomes "+12345". The signed zoned integer value F1 F2 F3 F4 D5 becomes "-12345".
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	A two byte integer in the range –32768 to 32767 (-(2**15) to (2**15) – 1)	String that represents the integer value (negative, positive or 0).
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	A two byte integer in the range 0 to 65535 (0 to (2**16) – 1)	String that represents the integer value (>= 0).

Data Type	Source Data Value Range	Intermediate String Representation
Decimal, International	A character string representing a number, where every third digit left of the decimal point is preceded by a period. The decimal point is represented by a comma. A '+' or a '-'may precede the value. For example, the number "12345.678" is represented as "12.345,678".Numbers that contain no separators or some subset of separators are considered valid input:12345-12345,123.456789,00	String that represents the number, without 3 digit separators, but with a decimal point (1.234,56 does not become 1,234.56. It becomes 1234.56). If the input value contains no decimal point (','), a decimal point is appended to the end of the value. For example, "12345" in 'Decimal, International' is converted to the string "12345.".
Decimal, U.S.	A character string representing a number, where every third digit left of the decimal point is preceded by a comma. The decimal point is represented by a period. A '+' or a '-' may precede the value. For example, the number "12345.678" is represented as "12,345.678".Numbers that contain no separators or some subset of separators are considered valid input:12345-12345.123,456789.00	String that represents the number, without 3 digit separators, but with a decimal point (1,234.56 does not become 1,234.56. It becomes 1234.56). If the input value contains no decimal point ('.'), a decimal point is appended to the end of the value. For example, "12345" in 'Decimal, U.S.' is converted to the string "12345.".
Date and Time	A 14 byte string that is numeric (each character is in the range '0' to '9'), and represents a valid date in the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS. The base data type may be String, EBCDIC or Numeric.	If the base data type is 'String' or 'Numeric', the input value is unchanged. If the base data type is 'EBCDIC', each EBCDIC character is converted to its native encoding.

Data Type	Source Data Value Range	Intermediate String Representation
Time	A 6 byte string that is numeric (each character is in the range ‘0’ to ‘9’), and represents a valid time in the international ISO-8601:1988 standard datetime notation: HHMMSS. The base data type may be String, EBCDIC or Numeric.The Time value is converted to the "Date and Time" format, with the date portion set to zeroes.	If the base data type is ‘EBCDIC’, each EBCDIC character is converted to its native encoding.
Date	An 8 byte string that is numeric (each character is in the range ‘0’ to ‘9’), and represents a valid date in the international ISO-8601:1988 standard datetime notation: YYYYMNDD. The base data type may be String, EBCDIC or Numeric.	The Date value is converted to the "Date and Time" format, with the time portion set to zeroes. If the base data type is ‘EBCDIC’, each EBCDIC character is converted to its native encoding.

Data Type	Source Data Value Range	Intermediate String Representation
Custom Date and Time	A string in one of the NEON supplied custom date/time formats. (Users may not currently specify their own date/time formats.) The base data type may be String, EBCDIC or Numeric. (See description of "Custom Date and Time" in the "Supported data types" table for a description of the components of a custom date/time format. Here is a list of the formats supplied with Formatter: MN/DD/YYDD/MN/YYMN/DD/YYYYDD/MN/YYYYDD-MON-YYDD-MON-YYYYMON-YYMON-YYYYMN/DD/YY HH:MM PMDD/MN/YY HH:MM PMMN/DD/YY HH:MM DD/MN/YY HH:MM MN/DD/YY HH:MM:SS PMDD/MN/YY HH:MM:SS PMMN/DD/YY HH:MM:SSDD/MN/	The intermediate representation of the custom date/time format is in the default date/time format: "Date and Time" (14 byte numeric string in the form: YYYYMNDDHHMMSS). If the base data type is EBCDIC, each character value is converted to its native encoding.
	YYHH:MM:SSHH:MM PMHH:MMHH:MM:SS PMHH:MM:SSMNDDYY MNDDYYYYDDMONYY DDMONYYYYMONYYM ONYYYYMONDDYYYY MNDDYYHHMMMNDD YYHHMMSSHHMMSS	

Data Type Conversion Constraints

There are some pairs of data type conversions that are not sensible: converting the string "Good morning" to a number, for example. This section discusses the constraints that exist for pairs of data conversions.

Not applicable

A data type of "Not applicable" means that the user does not want data type conversion to take place. The output data type should also be "Not applicable", so that Formatter does not attempt to change the data between the input message and the output message.

String

A string is a sequence of characters encoded in the native encoding (ASCII for UNIX and Windows NT) for the machine that Formatter is executing on.

Output Data Type	Constraints
Not Applicable	The data will remain unchanged between the input message and the output message.
String	The data will remain unchanged between the input message and the output message.
Numeric	Valid only if each character of the field value is 0 – 9. In this case, the data will remain unchanged between the input message and the output message.
Binary Data	Valid only if the string contains only the characters 0 – 9 and A – F, and the string contains an even number of characters. That is because the string is interpreted as the string encoded hexadecimal form of a binary field. For example, the string "0123AC" is turned into the 3-byte binary value: 01 23 AC (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal).
EBCDIC Data	This is a valid conversion. Note that values in the ASCII character set which don't have equivalent values in the EBCDIC character set are converted to an EBCDIC space character.
IBM Packed Integer IBM Zoned Integer	Valid only if the string is an integer with a value >=0 within the allowed range for the IBM type.
IBM Signed Packed Integer IBM signed Zoned Integer	Valid only if the string is an integer with a value within the allowed range for the IBM type.
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	Valid only if the string represents an integer, and has a value that is within the range allowed for Endian 2 types.
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	Valid only if the string represents an integer and has a value that is within the range allowed for Unsigned Endian 2 types.
Little Endian 4Little Swap Endian 4Big Endian 4Big Swap Endian 4	Valid only if the string represents an integer and has a value that is within the range allowed for Endian 4 types.
Unsigned Little Endian 4Unsigned Little Swap Endian 4Unsigned Big Endian 4Unsigned Big Swap Endian 4	Valid only if the string represents an integer and has a value that is within the range allowed for Unsigned Endian 4 types.
Decimal, International Decimal, U.S.	Valid only if the string represents an integer or floating point number.

Output Data Type	Constraints
Date and Time Custom Date and Time Time Date	Valid only if the string is in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

Numeric

A numeric string is a sequence of characters encoded in the native encoding (ASCII for UNIX and Windows NT) for the machine that Formatter is executing on. A numeric string contains only the characters 0 – 9 (no '+' or '-').

Output Data Type	Constraints
Not Applicable	The data will remain unchanged between the input message and the output message.
String	The data will remain unchanged between the input message and the output message
Numeric	The data will remain unchanged between the input message and the output message.
Binary Data	Valid only if the numeric string contains an even number of characters. That is because the string is interpreted as the string encoded hexadecimal form of a binary field. For example, the string "012345" is turned into the 3 byte binary value: 01 23 45 (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal).
EBCDIC Data	This is a valid conversion. Each ASCII character is converted to its EBCDIC equivalent
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	Valid only if the string represents a number with a value that is within the range allowed for the IBM types.
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	Valid only if the string represents a number with a value that is within the range allowed for Endian 2 types.
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	Valid only if the string represents a number with a value that is within the range allowed for Unsigned Endian 2 types
Little Endian 4 Little Swap Endian 4 Big Endian 4 Big Swap Endian 4	Valid only if the string represents a number with a value that is within the range allowed for Endian 4 types

Output Data Type	Constraints
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if the string represents a number with a value that is within the range allowed for Unsigned Endian 4 types
Decimal, International Decimal, U.S.	This is a valid conversion.
Date and Time Custom Date and Time Time Date	Valid only if the string is in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

Binary

Binary indicates a sequence of binary characters.

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	The data is converted to a string encoded hexadecimal format. For example, the binary string 01 23 AC (each pair of numbers represents the 2 nibbles of a byte in hexadecimal) is converted to the string “0x0123AC” Note that the prefix ‘0x’ is prepended to the data.
Numeric	This is a valid conversion only if the string encoded hexadecimal form of the binary string contains only the numbers 0 – 9.
Binary Data	The data will remain unchanged between the input message and the output message.
EBCDIC Data	Same behavior as for “String”, above, except that each character is an EBCDIC encoded character, rather than a native encoded character.
IBM types	This is a valid conversion only if the string encoded hexadecimal form of the binary string contains only the numbers 0 – 9, and has a value within the range allowed for the IBM types.
Endian types	This is a valid conversion only if the string encoded hexadecimal form of the binary string contains only the numbers 0 – 9, and the number is within the range allowed for the various Endian types.
Decimal, International Decimal, U.S.	See “Numeric” above.
Date and Time Custom Date and Time Date Time	Valid only if the string encoded hexadecimal form of the binary value is in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

EBCDIC

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	This is a valid conversion (each EBCDIC character is converted to its ASCII equivalent). Note that values in the EBCDIC character set which don’t have equivalent values in the native encoded character set are converted to a native encoded space character.
Numeric	Valid only if each character of the field value is EBCDIC 0 – 9.
Binary Data	Valid only if the string contains the EBCDIC characters 0 – 9 and A – F, and the string contains an even number of characters. That is because the string is interpreted as the string encoded hexadecimal form of a binary field. For example, the string “0123AC” is turned into the 3 byte binary value: 01 23 AC (where each pair of numbers represents the 2 nibbles of a byte in hexadecimal).
EBCDIC Data	The data will remain unchanged between the input message and the output message.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	Valid only if the string represents an integer and has a value within the ranges allowed for the IBM types.
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	Valid only if the string represents an integer, and has a value within the range allowed for the Endian 2 types.
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	Valid only if the string represents an integer, and has a value within the range allowed for the Unsigned Endian 2 types.
Little Endian 4 Little Swap Endian 4 Big Endian 4 Big Swap Endian 4	Valid only if the string represents an integer and has a value within the range allowed for the Endian 4 types.
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if the string represents an integer and has a value within the range allowed for the Unsigned Endian 4 types.
Decimal, International Decimal, U.S.	Valid only if the string represents an integer or floating point number.

Output Data Type	Constraints
Date and Time Custom Date and Time Time Date	Valid only if the string is in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

IBM Types

This is a numeric type that includes IBM packed, IBM signed packed, IBM zoned, IBM signed zoned.

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	This is a valid conversion. The value, incorporating the implied decimal point, is converted to a string representing the number.
Numeric	Valid only if the number is an integer >= 0.
Binary Data	Valid only if the number is an integer >= 0 and has an even number of digits. The number is first converted to a string, and then the string is interpreted as the string encoded hexadecimal form of the binary value.
EBCDIC Data	This is a valid conversion. The value, incorporating the implied decimal point, is converted to an EBCDIC encoded string representing the number.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion for all pairs of data types, as long as values in the source data type are in the range allowed for the target data type.
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	Valid only if the number is an integer and has a value within the range allowed for the Endian 2 types.
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	Valid only if the number is an integer and has a value within the range allowed for the Unsigned Endian 2 types.
Little Endian 4 Little Swap Endian 4 Big Endian 4 Big Swap Endian 4	Valid only if number is an integer and has a value within the range allowed for the Endian 4 types.
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if number an integer and has a value within the range allowed for the Unsigned Endian 4 types.

Output Data Type	Constraints
Decimal, International Decimal, U.S.	This is a valid conversion.
Date and Time Custom Date and Time Time Date	Valid only if the number converts to a string in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

Endian 2 Types

This is a 2 byte numeric type that includes the following:

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	This is a valid conversion. The value is converted to a string representing the integer.
Numeric	Valid only if the value of the number is an integer with a value >= 0.
Binary Data	Valid only if the value of the number is an integer with a value >= 0 with an even number of digits.
EBCDIC Data	This is a valid conversion. The value is converted to an EBCDIC encoded string representing the integer.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion, as long as values in the source data type are in the range allowed for the target data type.
Endian 2 types	This is a valid conversion, as long as values in the source data type are in the range allowed for the target data type.
Endian 4 types	Valid for conversions signed being to signed. Valid for conversions unsigned to unsigned. Valid for conversions unsigned being to signed. Valid for conversions signed to unsigned, if the number is >= 0.
Decimal, International Decimal, U.S.	This is a valid conversion.
Date and Time Custom Date and Time Time Date	Not a valid conversion. Cannot generate enough digits to represent a date/time value.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion, as long as values in the source data type are in the range allowed for the target data type.
Endian 2 types	This is a valid conversion, as long as values in the source data type are in the range allowed for the target data type.

Output Data Type	Constraints
Endian 4 types	Valid for conversion signed to signed. Valid for conversion unsigned to unsigned. Valid for conversion unsigned being to signed. Valid for conversion signed being to unsigned, if the number is >= 0.
Decimal, International Decimal, U.S.	This is a valid conversion.
Date and Time Custom Date and Time Time Date	Not a valid conversion. (Cannot generate enough digits to represent a date/time value.)

Endian 4 Types

This is a 4 byte numeric type that includes Little Endian 4, Little Swap Endian 4, Big Endian 4, Big Swap Endian 4, Unsigned Little Endian 4, Unsigned Little Swap Endian 4, Unsigned Big Endian 4, and Unsigned Big Swap Endian 4.

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	This is a valid conversion. The value is converted to a string representing the integer.
Numeric	Valid only if the value of the number is an integer with a value >= 0.
Binary Data	Valid only if the value of the number is a positive integer with an even number of digits.
EBCDIC Data	This is a valid conversion. The value is converted to an EBCDIC encoded string representing the integer.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion. For signed numbers being converted to unsigned numbers, only values >= 0 in the correct range are valid. For unsigned numbers being converted to signed numbers, only values in the correct range are valid.
Endian 2 types	Valid for signed being converted to signed only if the Endian 4 number is in the range allowed for signed Endian 2 types. Valid for unsigned being converted to unsigned only if the Endian 4 number is in the range allowed for unsigned Endian 2 types. Valid for signed being converted to unsigned only if the Endian 4 number in the range allowed for unsigned Endian 2 types. Valid for unsigned being converted to signed only if the Endian 4 number is in the range allowed for signed Endian 2 types.

Output Data Type	Constraints
Endian 4 types	Valid for signed being converted to signed. Valid for unsigned being converted to unsigned. Valid for unsigned being converted to signed. Valid for signed being converted to unsigned, only if the number is >= 0.
Date and Time Custom Date and Time Time Date	Valid only if the number converts to a string in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

Decimal International and Decimal US

A Decimal International or Decimal US value is a string representing a number, where every third digit left of the decimal point is preceded by a comma (Decimal US) or a period (Decimal International). The decimal point is represented by a period (Decimal US) or a comma (Decimal International). A '+' or a '-' may precede the value. For example, the number "12345.678" is represented as "12,345.678" in Decimal US and "12.345,678" in Decimal International.

Output Data Type	Constraints
Not Applicable	The data will remain unchanged between the input message and the output message.
String	The data will remain unchanged between the input message and the output message.
Numeric	Invalid conversion. Numeric does not accept '.', ',', '+' or '-'.
Binary Data	Invalid conversion. Binary does not accept '.', ',', '+' or '-'.
EBCDIC Data	The data will remain unchanged between the input message and the output message.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer Endian types Date and Time Custom Date and Time Time Date	Invalid conversion.
Decimal, International Decimal, U.S.	This is a valid conversion. If going between International and US, the '.' character is changed to a ',' and the ',' character is changed to a '.'.

Date and Time

This includes Date and Time, Custom Date and Time, Date, and Time.

Output Data Type	Constraints
Not Applicable	Same behavior as for “String”, following.
String	This is a valid conversion. The value is converted to a string representing the value of the date/time in its default format. (Date and Time, Date, and Time are already in their default format, so there is no change in data. Data in Custom Date and Time format is changed as described.)
Numeric	Same behavior as for “String”, above.
Binary Data	This is a valid conversion. Here is an example: the date/time value “19560601190000” (June 1, 1956 at 7:00 PM) would be converted to the binary string (each pair of numbers represents the 2 nibbles of a byte): 19 56 06 01 19 00 00.
EBCDIC Data	Same behavior as for “String” above, except that each string character is encoded as EBCDIC.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion if the integer number that represents the date/time is within the range allowed for the IBM types. The date/time is converted to its default format which is always an integer. That integer is then converted to the IBM type.
Endian 2 types	This is valid only if the integer number that represents the date/time is within the range allowed for Endian 2 types.
Endian 4 types	This is valid only if the integer number that represents the date/time is within the range allowed for Endian 4 types.
Decimal, International Decimal, U.S.	This is a valid conversion.
Date and Time Custom Date and Time	Date and Time and Custom Date and Time may be converted between each other, and to Time (drops the date portion) and to Date (drops the time portion).
Time	A Time may be converted to a Time, a Date (set to all zeroes), or Date and Time (date portion set to zeroes), or Custom Date and Time (date portion set to zeroes).
Date	A Date may be converted to a Date, a Time (set to all zeroes), or Date and Time (time portion set to zeroes), or Custom Date and Time (time portion set to zeroes).

Automatic Format Conversion

MQSeries Integrator contains some higher-level APIs that can request that NEONFormatter reformat messages just before delivery to the receiving application by invoking dynamic formatting as a get option. Reformatting locations can differ, depending on the location of resources (such as source data) needed to format the new message.

The NNHPutMsg() command uses Formatter by means of a function call, as does the sending process, receiving process, and/or NNHGetMsg() command. Sending and receiving applications remain uninvolved with transaction details.

Using the NEONFormatter Engine

View the NEONFormatter engine like a factory taking in raw materials on one side and producing a finished product on the other. Raw materials include the input messages and the input formats describing how the input messages are to be broken down (parsed), and the output formats describing how the input messages are to be reassembled (formatted). At the end of the process, the Formatter engine "factory" produces parsed input messages and reformatted output messages as the finished product.

One at a time, input messages are given to Formatter, using the AddInputMessage() function. In addition to the message specified in the function call, you must also specify the input format to be used to parse the input message. The input message and format arguments are specified by char* variables providing the application the address of the buffer containing each. The name of the input format provided will be used to retrieve the specified input format from the database.

Output formats describing how to format the parsed input messages are provided to Formatter using the AddOutputFormat() function. Similar to AddInputMessage(), an output format is specified by a char* variable providing the address of the buffer containing the output format name. The output format can then be retrieved from the database.

The general method for formatting a message follows this algorithm:

Instantiate an instance of the DbmsSession class to open a database session.

Instantiate an instance of the Formatter class, passing it the DbmsSession instance.

While there are input messages to format...

For each input message to be formatted, call AddInputMessage() to add the input message along with the input format for the message.

For each desired output message, call AddOutputFormat() to add the output format.

Call Reformat on the Formatter instance.

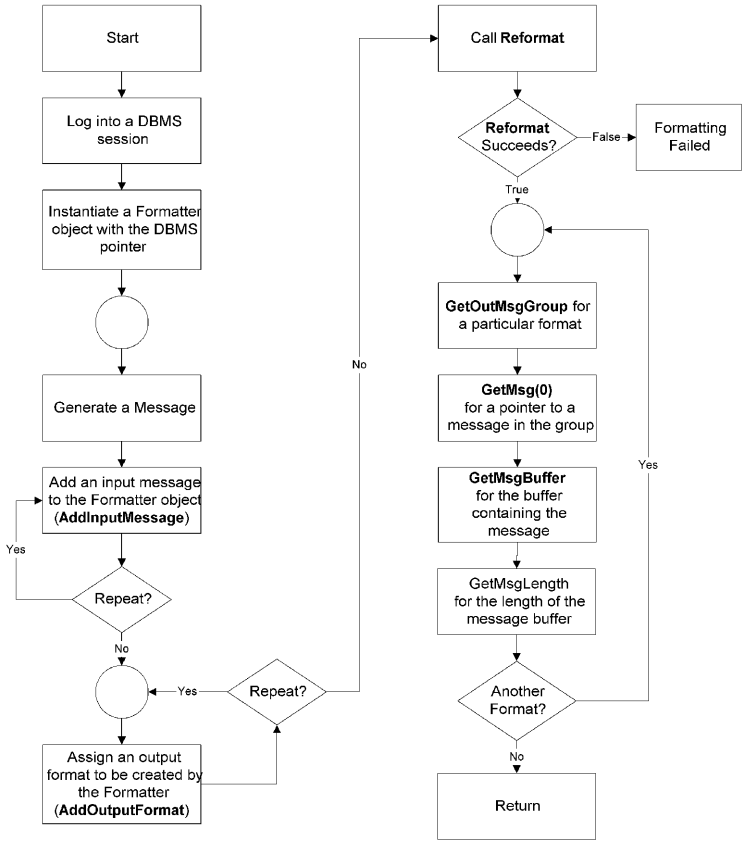
[Formatter will format one output message for each output format specified using AddOutputFormat().]

For each format that was added via AddOutputFormat(), call GetOutMsgGroup() and GetOutMsg() to get the resulting formatted message for the format.

end While

Clean up.

Flow of Calls



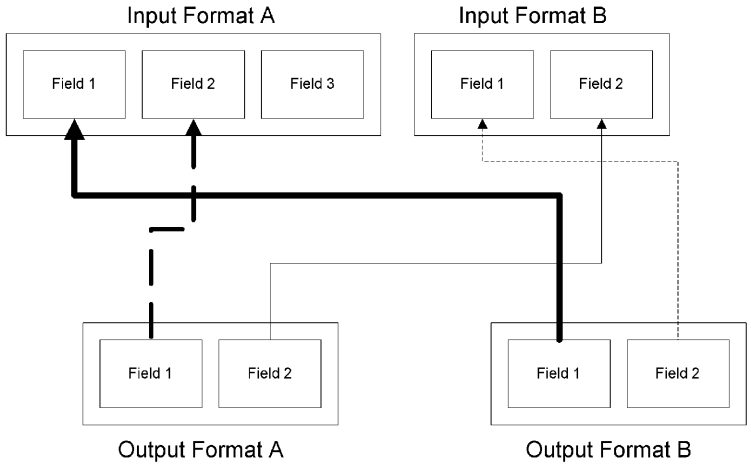
For each output format added using AddOutputFormat(), a formatted message will be produced. For example, if one input message was added using AddInputMessage() and three output formats were specified using AddOutputFormat(), Formatter will generate three formatted messages. The input message will be formatted to fit each of the three output formats.

Each field of an input message (parsed according to the specified input format) you want to appear in the output message must be mapped to a field in an output format. This mapping can be implicit, based on the name of each field in the output format; or it can be explicit.

Field mapping provides flexibility, enabling the combination of different input message/output format field-level mappings. The following examples illustrate some different ways mappings can be combined. Note that these are simple examples and much more complex mappings are possible.

- One output format can map to more than one input format by mapping some of its fields to input format A, some to input format B, and so on, resulting in an output message formed from fields from both input messages A and B.
- One input message can map to more than one output format by mapping the fields of the output formats to one input format, resulting in n output messages (where n is the number of output formats mapped to the input message) formed from fields of the input message.
- Several output formats may also be mapped to several input formats, resulting in n output messages (where n is the number of output formats) with each output message containing formatted fields from some or all of the input messages.

  For example, field 1 of output format A maps to field 2 of input format A; field 2 of output format A maps to field 2 of input format B; field 1 of output format B maps to field 1 of input format A; field 2 of output format B maps to field 2 of input format B. The resulting output is a series of formatted input fields in the following order: field 2 of input format A, field 2 of input format B, field 1 of input format A, and field 2 of input format B. (Notice that in this mapping, field 1 of input format B is ignored.) The following diagram illustrates this example.



Forming Multiple Output Messages from Multiple Input Messages

APIs and Header Files

The NEONFormatter API is made up of the public interfaces for six C++ classes, and interfaces for User Exits and User Callbacks:

Header Files

Object Class	Header File	Description
Formatter	formatter.h	Formatter Class
OutMsgGroup	msgs.h	Output message group contained in Formatter Class
OutMsg	msgs.h	Output message contained in OutMsgGroup Class
ParsedMessage	pmsg.h	Parsed Message Class
ParsedField	pfield.h	Parsed Field Class
NNFMgr	nnfmgr.h	Format Management APIs
--	nnexit.h	User Exits
--	nnuserfunction.h	User Callbacks

Formatter Class Functions

Return Type	Function	Arguments
N/A	(Constructor)	(DbmsSession * session)
void	ResetDbmsSession	(DbmsSession *DatabaseSessionObject)
void	AddInputMessage	(char* FormatName, Char* MsgBuf, int MsgLength)
void	AddOutputFormat	(char* FormatName)
int	PreloadInFormat	(char* pInFormatName)
int	PreloadOutFormat	(char* pOutFormatName)
int	parse	none
int	reformat	none
char*	GetFieldAscii	(char* FieldName, int SequenceNumber)
char*	GetFieldAsciiByTag	(char* pTagName, int SequenceNumber)
int	GetOutMsgCount	none
OutMsgGroup*	GetOutMsgGroup	(char* FormatName)
int	GetParsedInMsgCount	none

Return Type	Function	Arguments
ParsedMessage*	GetParsedInMsg	(int index)
void	SetUserTypeValidationOn	none
void	SetUserTypeValidationOff	none
int	UserTypeValidationIsOn	none
int	GetErrorCode	none
char*	GetErrorMessage	none

OutMsg Class Functions

Return Type	Function	Arguments
char*	GetMsgBuffer	none
int	GetMsgLength	none

OutMsgGroup Class Functions

Return Type	Function	Arguments
OutMsg*	GetMsg	(int index)
int	GetMsgCount	none

ParsedField Class Functions

Return Type	Function	Arguments
char*	GetInfo	none
char*	GetAsciiValue	(int* pDataLength)
char*	GetValue	(int* pDataType, int* pDataLength)

ParsedMessage Class Functions

Return Type	Function	Arguments
int	GetCompCount	none
ParsedMessage*	GetMsgComp	(int index)
char*	GetInfo	(int* pMsgType)
ParsedField*	GetFieldComp	(int index)

Format Management API Functions

Return Type	Function	Arguments
NNFMgr *	NNFMgrInit	(DbmsSession *session)
void	NNFMgrClose	(NNFMgr *pNNFMgr)

Return Type	Function	Arguments
N/A	NNF_CLEAR	(_p)
const short	NNFMgrCreateField	(NNFMgr * pNNFMgr, const NNFMgrFieldInfo * const pFieldInfo)
const short	NNFMgrAppendFieldTo InputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, const NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrAppendFieldTo OutputFormat	(NNFMgr * pNNFMgr, const char * const pFormatName, const NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetFirstField	(NNFMgr *pNNFMgr, NNFMgr * const pFieldInfo)
const short	NNFMgrGetNextField	(NNFMgr *pNNFMgr, NNFMgr * const pFieldInfo)
const short	NNFMgrGetFirstFieldFrom InputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetNextFieldFrom InputFormat	(NNFMgr *pNNFMgr, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetFirstFieldFrom OutputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetNextFieldFrom OutputFormat	(NNFMgr *pNNFMgr, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrCreateDelimeter	(NNFMgr *pNNFMgr, const NFMgrDelimiterInfo * const pDelimiterInfo)
const short	NNFMgrGetDelimiter	(NNFMgr *pNNFMgr, const char * const pDelimiterName, NNFMgrDelimiterInfo * const pDelimiterInfo)
const short	NNFMgrGetFirstDelimiter	(NNFMgr *pNNFMgr, NNFMgrDelimiterInfo *pDelimiterInfo)
const short	NNFMgrGetNextDelimiter	(NNFMgr *pNNFMgr, NNFMgrDelimiterInfo *pDelimiterInfo)

Return Type	Function	Arguments
const short	NNFMgrCreateParse Control	(NNFMgr *pNNFMgr, const NNFMgrParseControlInfo *pParseControlInfo)
const short	NNFMgrGetParseControl	(NNFMgr * pNNFMgr, char * pParseName, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrGetFirstParse Control	(NNFMgr * pNNFMgr, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrGetNextParse Control	(NNFMgr * pNNFMgr, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrCreateOutput Control	(NNFMgr *pNNFMgr, const NNFMgrOutputControlInfo * const pOutputControlInfo)
const short	NNFMgrAppendMath Expression	(NNFMgr *pNNFMgr, const char * const pOutputControl, const NNFMgrMathExpression *pMathExpr)
const short	NNFMgrAppendOutput LookupEntry	(NNFMgr *pNNFMgr, const char * const pOutputControl, const NNFMgrLookupEntry * const pLookupEntry)
const short	NNFMgrGetOutputControl	(NNFMgr *pNNFMgr, const char * const pControlName, NNFMgrOutputControlInfo * const pOutputControlInfo)
const short	NNFMgrGetFirstOutput Control	(NNFMgr *pNNFMgr, NNFMgrOutputControlInfo * const pOutputControlInfo)
const short	NNFMgrGetNextOutput Control	(NNFMgr *pNNFMgr, NNFMgrOutputControlInfo * const pOutputControlInfo)
const short	NNFMgrGetFirstMath Expression	(NNFMgr *pNNFMgr, const char * const pOutputControlName, NNFMgrMathExpression * const pMathExpression)
const short	NNFMgrGetNextMath Expression	(NNFMgr *pNNFMgr, NNFMgrLookupEntry * const pMathExpression)

Return Type	Function	Arguments
const short	NNFMgrGetFirstLookup Entry	(NNFMgr *pNNFMgr, const char * const pOutputControlName, NNFMgrLookupEntry * const pLookupEntry)
const short	NNFMgrGetNextLookup Entry	(NNFMgr *pNNFMgr, NNFMgrLookupEntry * const pLookupEntry)
const short	NNFMgrCreateFormat	(NNFMgr * pNNFMgr, const NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)
const short	NNFMgrAppendFormatTo Format	(NNFMgr *pNNFMgr, const char * const pParentName, const NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)
const short	NNFMgrGetFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)
const short	NNFMgrGetFirstFormat	(NNFMgr * pNNFMgr, NNFMgrFormatInfo * const pFormatInfo)
const short	NNFMgrGetNextFormat	(NNFMgr * pNNFMgr, NNFMgrFormatInfo * const pFormatInfo)
const short	NNFMgrGetFirstChild Format	(NNFMgr *pNNFMgr, const char * const pParentName, NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)
const short	NNFMgrGetNextChild Format	(NNFMgr *pNNFMgr, NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)
const short	NNFMgrCreateUserDefined Type	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrGetUserDefined Type	(NNFMgr *pNNFMgr, const char* const pTypeName, const NNFMgrUserDefTypeInfo * const pTypeInfo)

Return Type	Function	Arguments
const short	NNFMgrGetFirstUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrGetNextUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrAddNameValue Pairs	(NNFMgr *pNNFMgr, const NNFMgrNameValuePairInf o * const pPairInfo)

Libraries

Libraries are archived collections of object files. The following libraries need to be linked with the application object files:

Library Files Needed

Library	Description
librules	Rules Library
libformat	Dynamic Formatter Library
libntools	Generic Tool set
libfmgr	Format Management Library
—	System / Compiler Specific Libraries
—	Database-dependent Libraries

Note

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

Note

THREAD SAFETY: To link with the Thread Safe Formatter, the thread library corresponding to the release will need to be linked. For example, to link with the POSIX pthreads version of Formatter, the pthreads library will need to be linked with the final executable.

Chapter 3

Formatter APIs

Formatter Member Functions

Formatter Constructor

Overview

The Formatter Constructor creates an instance of a new Formatter class.

Syntax#1

```
Formatter::Formatter (DbmsSession*
                      DatabaseSessionObject);
```

Description#1

This overloaded version of the Constructor only needs a session pointer to the input configuration database object.

Parameters#1

Name	Type	Input/ Output	Description
DatabaseSessionObject	DbmsSession*	Input	Name of the currently open database session. See OpenDbmsSession().

Syntax#2

```
Formatter::Formatter (
DbmsSession* DatabaseSessionObject
NNFunctionKeyPairCollection* ValidationCallbackObject);
```

Description #2

This overloaded version of the Constructor is used when there are user-defined type input field validation callback objects to register with Formatter. For information on user callbacks in general, see *User Callback API Functions* on page 133. For information on user-defined type input field validation, see *User-defined Type Input Field Validation* on page 169.

Parameters

Name	Type	Input/ Output	Description
DatabaseSessionObject	DbmsSession*	Input	Name of the currently open database session. See OpenDbmsSession().
ValidationCallback Object	NNFunction KeyPair Collection*	Input	A collection of callback objects and their associated keys to be used for user-defined type input field validation.

Example #1

See Sample Program 1: msgtest.cpp.

Example #2

See Sample Program 1: msgtest.cpp.

See Also

OpenDbmsSession()

Formatter Destructor

Overview

The Formatter destructor is available to clean up any memory allocated by use of any Formatter constructor or associated APIs.

Note

THREAD SAFETY: For multi-threaded applications, Formatter::~Formatter should ONLY be called by the main thread after all threads are done with parsing or reformatting.

Syntax

```
Formatter::~Formatter()
```

Parameters

None.

Remarks

Formatter::~Formatter must be called after Formatter::Formatter and after all formatter processing is complete.

Return Value

None

There are no error handling functions for Formatter::~Formatter.

Example

```
Formatter formatter(Session);
if (handleError("formatter constructor", &formatter)) {
    exit(1);
}
formatter.AddInputMessage(inFormatName,msg,msgLen);
if (handleError("Formatter::AddInputMessage",
    &formatter)) {
    exit(1);
}
// Parse the message
formatter.Parse();
if (!handleError("Formatter::Parse", &formatter)) {
    exit(1);
}
formatter.~Formatter()
```

ResetDbmsSession

Overview

ResetDbmsSession() closes or changes the database session used by Formatter.

WARNING!

Do not attempt to close the database session for any formats loaded that use conditional branching. Conditional branching rules do not get loaded until reformatting time.

Syntax

```
void Formatter::ResetDbmsSession(DbmsSession
                                 *DatabaseSessionObject)
```

Parameters

Name	Type	Input/ Output	Description
DatabaseSessionObject	DbmsSession*	Input	Null pointer or pointer to a new and currently open database session. See OpenDbmsSession().

Remarks

Use this function to do one or both of the following:

1. Let Formatter know the database session is closed:

   ```
   Formatter::ResetDbmsSession((DbmsSession*)0);
   ```

   then, close the database session.

2. And/or open a new database session and instruct Formatter to use the new connection.

Return Value

None

Example

```
// Open a database session...
DbmsSession*myDbmsSession;
myDbmsSession = OpenDbmsSession("format_session_name",
ORACLE7);

// Construct a Formatter instance.
Formatterformatter(myDbmsSession);
```

```
// Close database session and inform Formatter.
CloseDbmsSession(myDbmsSession);
formatter.ResetDbmsSession((DbmsSession *)0);

// Open a new database session and inform Formatter.
myDbmsSession =
OpenDbmsSession("new_format_session_name", ORACLE7);
formatter.ResetDbmsSession(myDbmsSession);
```

See Also

OpenDbmsSession(), CloseDbmsSession()

AddInputMessage

Overview

AddInputMessage() stores a copy of an input message within the Formatter object together with a copy of its format name. The named format must exist in the Formatter database.

Note

THREAD SAFETY NOTE: All input messages added using AddInputMessage() will be processed entirely within the thread from which they were added. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void Formatter::AddInputMessage(char* FormatName,
                                char* MsgBuffer,
                                int MsgLength);
```

Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Identifier to retrieve a format's configuration definition from the database.
MsgBuffer	char*	Input	A pointer to the buffer containing the message being added.
MsgLength	int	Input	Length, in bytes, of the message being added.

Remarks

AddInputMessage() does not validate the format name. The format name is validated when Parse() is called.

If the pointers to FormatName and MsgBuffer have NULL values or MsgLength has a value less than zero (0), AddInputMessage() sets an error message so that when the error handling routines are used to return the error message, the message indicates which parameter had a bad value.

If AddInputMessage() is called after a Reformat() or Parse(), all previous input messages, output messages, and output formats are cleared from the internal buffer.

Note

The user should ensure that the message buffer passed into this function is allocated by the user and not by any Formatter API calls (such as OutMsgGroup::GetMsg). All formatter APIs have the ability and will change buffers allocated by any other formatter APIs.

Return Value

None

Example

See Sample Program 1: msgtest.cpp.

See Also

Parse(), Reformat(), PreloadInFormat()

AddInputMessage

Overview

This overloaded version of AddInputMessage() directs the Formatter to use the caller's input buffer directly. AddInputMessage() stores a copy of an input message within the Formatter object together with a copy of its format name. The named format must exist in the Formatter database.

Syntax

```
void Formatter::AddInputMessage(char* FormatName,
                                char* MsgBuffer,
                                int MsgLength);
```

Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Identifier to retrieve a format configuration definition from the database.
MsgBuffer	char*	Input	A pointer to the buffer containing the message being added.
MsgLength	int	Input	Length, in bytes, of the message being added.
bMakeCopyOfBuffer	int	Input	A pointer to the buffer containing the message being added.

Remarks

AddInputMessage() does not validate the format name. The format name is validated when Parse() is called.

If the bMakeCopyOfBUffer parameter contains a zero (0) value, the caller's buffer is used directly instead of making an internal copy of the caller's buffer. The caller's buffer is not destroyed when the formatter object is destroyed.

WARNING!

The caller must destroy the buffer ONLY after the formatter object has been destroyed.

If the pointers to FormatName and MsgBuffer have NULL values, or MsgLength has a value less than zero (0), AddInputMessage() sets an error message so that when the error handling routines are used to return the error message, the message indicates which parameter had a bad value.

If AddInputMessage() is called after a preformat() or parse(), all previous input messages, output messages, and output formats are cleared from the internal buffer.

Return Value

None

See Also

parse(), preformat(), PreloadInFormat()

AddOutputFormat

Overview

AddOutputFormat() tells Formatter to create an output message of the type specified by FormatName when reformatting.

Note

THREAD SAFETY NOTE: All output formats added using AddOutputFormat() will be used to specify formatting rules for all input messages added within the same thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void Formatter::AddOutputFormat(char* FormatName);
```

Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Identifier to retrieve a format's configuration definition from the database.

Remarks

AddOutputFormat() does not validate FormatName. The format name is validated when Reformat() is called.

If the pointer to FormatName has a NULL value, AddOutputFormat() sets an error message so that when the error handling routines are used to return the error message, the message indicates that FormatName had a bad value.

Return Value

None

Example

See Sample Program 1: msgtest.cpp.

See Also

PreloadOutFormat()

RemoveOutputFormat

Overview

This API removes the output format from the list of output formats to be reformatted.

Note

THREAD SAFETY NOTE: All output formats added using AddOutputFormat() will be used to specify formatting rules for all input messages added within the same thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void Formatter::RemoveOutputFormat(char* pFormatName)
```

Parameters

Name	Type	Input/ Output	Description
pFormatName	char*	Input	Format name to remove.

Remarks

GetOutMsgGroup will no longer return an OutMsgGroup for this format name.

Return Value

Void

PreloadInFormat

Overview

PreloadInFormat() preloads an input format into memory.

PreloadInFormat() preloads an input format. If you don't use this function call, input formats get loaded from the database automatically during a call to Parse() or Reformat(). This function forces the load to happen immediately. While use of this function does not reduce the total amount of time spent by an application that uses Formatter, calling it allows the application programmer to control where during the application time will be spent to access the database.

Note

THREAD SAFETY NOTE: PreloadInFormat() only loads input formats for the thread from which it is called. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

WARNING!

THREAD SAFETY WARNING: If this function is called from the main thread, its results will not be available to any other threads.

Syntax

```
int Formatter::PreloadInFormat (char *pInFormatName);
```

Parameters

Name	Type	Input/ Output	Description
pInFormatName	char*	Input	Identifier to retrieve an input format's configuration definition from the database.

Return Value

Returns 1 if input format is loaded successfully; zero (0) on failure.

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

Let's say you have built input formats "IFFormat1", "IFFormat2" and "IFFormat3" using the Formatter GUI tool or the Format Management API functions, and that your Formatter application makes use of these formats. You could preload these format definitions prior to calling the other Formatter functions by adding the following calls to your application program:

```
// Construct Formatter instance
FormattermyFormatter;

// Preload input formats.
myFormatter.PreLoadInFormat(“IFFormat1”);
myFormatter.PreLoadInFormat(“IFFormat2”);
myFormatter.PreLoadInFormat(“IFFormat3”);

// Rest of application logic...
myFormatter.AddInputMessage...
myFormatter.Parse...
```

See Also

PreloadOutFormat()

PreloadOutFormat

Overview

PreloadOutFormat() preloads an output format. If you don't use this function call, output formats get loaded from the database automatically during a call to Reformat(). This function forces the load to happen immediately. While use of this function does not reduce the total amount of time spent by an application that uses Formatter, calling it allows the application programmer to control where during the application time will be spent to access the database.

Note

THREAD SAFETY NOTE: PreloadOutFormat() only loads output formats for the thread from which it is called. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

WARNING!

THREAD SAFETY WARNING: If PreloadOutFormat() is called from the main thread, its results will not be available to any other threads.

Syntax

```
int Formatter::PreloadOutFormat (char *pOutFormatName);
```

Parameters

Name	Type	Input/ Output	Description
pOutFormatName	char*	Input	Identifier to retrieve an output format's configuration definition from the database.

Return Value

Returns 1 if output format is loaded successfully; zero (0) on failure.

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

Let's say you have built output formats "OFFormat1", "OFFormat2" and "OFFormat3" using the Formatter GUI tool or the Format Management API functions, and that your Formatter application makes use of these formats. You could preload these format definitions prior to calling the other Formatter functions by adding the following calls to your application program:

```
// Construct Formatter instance
FormattermyFormatter;
```

```
// Preload output formats.
myFormatter.PreLoadOutFormat("OFFormat1");
myFormatter.PreLoadOutFormat("OFFormat2");
myFormatter.PreLoadOutFormat("OFFormat3");

// Rest of application logic...
myFormatter.AddInputMessage...
myFormatter.AddOutputFormat...
myFormatter.reformat...
```

See Also

PreloadInFormat()

StartDebug

Overview

StartDebug() initializes the parse debugger.

Note

You must provide a valid ostream, and verbose level for this function to work properly.

Syntax

```
int Formatter::StartDebug (
       const NN_DEBUG_CATEGORY pDebugCategory,
       const NN_DEBUG_VERBOSE_LEVEL pVerboseLevel,
       ostream& pOutputBuffer);
```

Parameters

Name	Type	Input/ Output	Description
pDebugCategory	const NN_DEBUG_CATEGORY	Input	Indicates which Formatter functionality the debugger will target. Currently, only PARSE applies.
pVerboseLevel	const NN_DEBUG_VERBOSE_LEVEL	Input	Indicates how detailed the debug information will be. currently, only BASIC applies.
pOutputBuffer	ostream&	Input	User-provided buffer. all debug information will be stored here.

Remarks

NN_DEBUG_CATEGORY has only one value for now (PARSE), but it is extendable for the future.

NN_VERBOSE_LEVEL also has only one value for now (BASIC), but it is extendable for the future.

Return Value

Returns 1 on successful initialization of the debugger; zero (0) on failure (for example, bad output stream).

See Also

StopDebug()

StopDebug

Overview

StopDebug() stops the debugger and cleans up memory used by the debugging process.

Note

StopDebug() must be called after each call to StartDebug().

Syntax

```
int Formatter::StopDebug();
```

Parameters

None

Return Value

None

See Also

StartDebug()

parse

Overview

parse() deconstructs input messages added by AddInputMessage() into their component fields. Individual field data is then accessible for processing by user applications.

Note

THREAD SAFETY NOTE: parse() only parses the input messages added with AddInputMessage() within the thread from which it is called. The parsed message will only be available from within this thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int Formatter::parse();
```

Parameters

None

Remarks

parse() can be called without reformatting the input messages into output messages. The Formatter will attempt to parse the input message but will not create any output message. This enables the user to call other message access calls such as GetFieldAscii() or GetFieldValue().

reformat() will call this function if it has not already been called for the current set of input messages.

If no input messages have been added using AddInputMessage(), parse() fails. When the error handling routines are used to return the error message, a "no input message" error will be returned.

WARNING!

It is not recommended that you call parse() twice without an intervening call to AddInputMessage(). The second call (and any subsequent calls) to parse() adds a duplicate parsed message (or set of parsed messages). For example, if you call AddInputMessage() three times to add three messages, call parse(), then call parse() again without an intervening AddInputMessage() call, you will end up with 6 parsed messages (two sets of the same three parsed messages).

Return Value

Returns 1 if parse is successful for all messages; zero (0) if any parse fails.

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

See Sample Program 2: apitest.cpp.

See Also

AddInputMessage(), GetFieldAscii(), reformat()

reformat

Overview

reformat() translates input messages (input using AddInputMessage()) into output messages (specified using AddOutputFormat()). Output messages are formatted into dynamically allocated character buffers.

Note

THREAD SAFETY NOTE: reformat() only formats input messages added with AddInputMessage() within the thread from which it is called. The formatted message will only be available from within this thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int Formatter::reformat();
```

Parameters

None

Remarks

If no input messages have been added (using AddInputMessage()), reformat() fails. When the error handling routines are used to return the error message, a "no input message" error will be returned.

If no output formats have been added (using AddOutputFormat()), reformat() fails. When the error handling routines are used to return the error message, a "no output formats" error will be returned.

WARNING!

Do not call reformat() twice without an intervening call to AddInputMessage() or AddOutputFormat(). The second call (and any subsequent calls) to reformat() adds a duplicate formatted message to the resulting OutMsgGroup.

When a field is being formatted using reformat(), and a substitute string is being used in the output control, the input field value must be found in the set of substitute string entries or the output field will not be output. If the input field value isn't found in the set of substitute string entries, the original input field value is unchanged for the output. In both cases, the reformat() succeeds.

Return Value

Returns 1 if successful; zero (0) if translation fails.

Use GetErrorCode() to check for an error; then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

See Sample Program 1: msgtest.cpp.

See Also

AddInputMessage(), AddOutputFormat(), parse()

GetFieldAscii/GetFieldString

Overview

GetFieldAscii() allows direct access to field contents based on the field name. This must be called after parse().

Note

THREAD SAFETY NOTE: GetFieldAscii() retrieves the results of a parse() within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* Formatter::GetFieldAscii(
                char* FieldName,
                int SequenceNumber);
```

Parameters

Name	Type	Input/ Output	Description
FieldName	char*	Input	NULL-terminated string specifying the field name of the desired field.
SequenceNumber	int	Input	Index specifying which field to reference if a field appears more than once in a message. This index starts at and defaults to zero (0), incrementing by one as you move left to right through the input message buffer.

Remarks

GetFieldString() performs the same function that GetFieldAscii() does except it is a portable version that should be used if porting across differing types of platforms.

Return Value

Returns a pointer to a NULL-terminated ASCII representation of the field's contents; returns a pointer to the first field if the field is not found.

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

You have an input format IFFormat with a field named F1 and you want to get the value of the input field F1. Here is an example sequence of Formatter function calls, including a call to GetFieldAscii():

```
// Construct Formatter instance
FormattermyFormatter;

// Declare variables.
char   myBuffer[BUFSIZ];
char   *pFieldValue;

// Load buffer with a message
strcpy(myBuffer, "This is some message text whose format
is IFFormat");

// Parse a message and get the value of field "F1".
myFormatter.AddInputMessage("IFFormat", myBuffer,
strlen(myBuffer));
myFormatter.parse();
pFieldValue = GetFieldAscii("F1");
```

See Also

GetFieldAsciiByTag()

GetFieldAsciiByTag/GetFieldStringByTag

Overview

GetFieldAsciiByTag() allows direct access to a tagged field contents by tag name. This must be called after parse().

Note

THREAD SAFETY NOTE: GetFieldAsciiByTag() retrieves the results of a parse() within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* Formatter::GetFieldAsciiByTag (char* pTagName,
                                     int SequenceNumber);
```

Parameters

Name	Type	Input/ Output	Description
pTagName	char*	Input	NULL-terminated string specifying the tag name of the desired field.
SequenceNumber	int	Input	Index specifying which field to reference if a field appears more than once in a message. This index starts at and defaults to zero (0), incrementing by one as you move left to right through the input message buffer.

Remarks

GetFieldStringByTag() performs the same function that GetFieldAsciiByTag does except it is a portable version that should be used if porting across differing types of platforms.

Return Value

Returns a NULL-terminated ASCII representation of the tag's contents; returns a pointer to the first field if the field is not found.

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

Let's say you have an input format "IFFormat" with a field named "F1". Field F1 is a tagged field, and the value of the tag is "TagForF1". You use Formatter to parse a message with this input format, and you want to get the value of the input field "F1". But you want to refer to F1 not by its name, but by its tag value. Here is an example sequence of Formatter function calls, including a call to GetFieldAsciiByTag():

```
// Construct Formatter instance.
FormattermyFormatter;

// Declare variables.
char   myBuffer[BUFSIZ];
char   *pFieldValue;

// Load buffer with a message.
strcpy(myBuffer, "This is some message text whose format
is IFFormat");

// Parse a message and get the value of field "F1" by
referring to its tag value.
myFormatter.AddInputMessage("IFFormat", myBuffer,
strlen(myBuffer));
myFormatter.parse();
pFieldValue = GetFieldAsciiByTag("TagForF1");
```

See Also

GetFieldAscii()

GetOutMsgCount

Overview

GetOutMsgCount() returns the number of output message groups in the Formatter object.

Note

THREAD SAFETY NOTE: GetOutMsgCount() returns the resulting number of output message groups after calling reformat() within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int Formatter::GetOutMsgCount();
```

Parameters

None

Return Value

Returns the number of output message groups in the Formatter object. There will be one output message group for each output format added using AddOutputFormat().

Use GetErrorCode() to check for an error, then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

See Sample Program 1: msgtest.cpp.

See Also

AddOutputFormat()

GetOutMsgGroup

Overview

GetOutMsgGroup() returns a pointer to the group of output messages for a particular format.

Note

THREAD SAFETY NOTE: GetOutMsgGroup() returns the specified resulting output message group from calling Reformat() within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
OutMsgGroup* Formatter::GetOutMsgGroup(char* FormatName);
```

Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Name of the output format whose OutMsgGroup is being identified.

Remarks

After GetOutMsgGroup() returns a pointer to an output message group, the application program can iterate through the messages using calls to GetMsg(int index). After a successful reformat, there will be 1 instance of OutMsg per OutMsgGroup.

Return Value

Returns a pointer to the OutMsgGroup identified by the FormatName parameter.

Use GetErrorCode() to check for an error; then use GetErrorMessage() to retrieve the error message associated with that error number.

Example

See Sample Program 1: msgtest.cpp.

See Also

GetMsg()

GetParsedInMsgCount

Overview

GetparsedInMsgCount() returns the number of input messages parsed by the Formatter. The number should equal the number of input messages added by AddInputMessage().

Note

THREAD SAFETY NOTE: GetParsedInMsgCount() returns the number of input messages parsed by Formatter within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int Formatter::GetParsedInMsgCount();
```

Parameters

None

Return Value

There are no error handling functions for GetParsedInMsgCount().

Example

See Sample Program 2: apitest.cpp.

See Also

AddInputMessage()

GetParsedInMsg

Overview

GetParsedInMsg() returns a pointer to a parsed input message at the specified index.

Note

THREAD SAFETY NOTE: GetParsedInMsg() returns the specified parsed input message resulting from calling Parse() within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
ParsedMessage* Formatter::GetParsedInMsg(int index);
```

Parameters

Name	Type	Input/ Output	Description
index	int	Input	Index of parsed input message to return.

Remarks

Index relates to the order in which messages were added using AddInputMessage(), starting at zero (0) for the first message and incrementing by one for each following message. For example, to access the third message added, index should equal 2.

Return Value

Returns a pointer to a parsed message; NULL if supplied with a bad index.

There are no error handling functions for GetParsedInMsg().

Example

See Sample Program 2: apitest.cpp.

See Also

parse(), AddInputMessage()

SetUserTypeValidationOn

Overview

SetUserTypeValidationOn() turns user-defined type input field validation on. (On is the default state.) This function sets the validation state of all fields defined in terms of user-defined types; the validation state of individual fields cannot be set.

Note

THREAD SAFETY NOTE: No matter what thread this function is called from, it sets the validation state for all threads of a Formatter instance. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void Formatter::SetUserTypeValidationOn();
```

Parameters

None

Return Value

None.

There are no error handling functions for SetUserTypeValidationOn().

Example

See Sample Program 1: msgtest.cpp.

See Also

SetUserTypeValidationOff(), UserTypeValidationIsOn()

SetUserTypeValidationOff

Overview

SetUserTypeValidationOff() turns user-defined type input field validation off. (On is the default state.) This function sets the validation state of all fields defined in terms of user-defined types; the validation state of individual fields cannot be set.

Note

THREAD SAFETY NOTE: No matter what thread this function is called from, it sets the validation state for all threads of a Formatter instance. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void Formatter::SetUserTypeValidationOff();
```

Parameters

None

Return Value

None.

There are no error handling functions for SetUserTypeValidationOff().

Example

See Sample Program 1: msgtest.cpp.

See Also

SetUserTypeValidationOn(), UserTypeValidationIsOn()

UserTypeValidationIsOn

Overview

UserTypeValidationIsOn() returns the current state of user-defined type input field validation.

Note

THREAD SAFETY NOTE: No matter what thread this function is called from, it returns the state of user-defined type input field validation for the current Formatter instance. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int Formatter::UserTypeValidationIsOn();
```

Parameters

None

Return Value

Returns zero(0) if validation is turned off; non-zero if validation is turned on.

There are no error handling functions for SetUserTypeValidationOff().

Example

See Sample Program 1: msgtest.cpp.

See Also

SetUserTypeValidationOn(), SetUserTypeValidationOff()

OutMsg Class Member Functions

GetMsgBuffer

Overview

GetMsgBuffer() returns a pointer to the buffer containing message text for a particular output message.

Note

THREAD SAFETY NOTE: GetMsgBuffer() returns a pointer to the buffer within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* OutMsg::GetMsgBuffer();
```

Parameters

None

Return Value

Returns a pointer to the internal message buffer retrieved with the preceding GetMsg() call. This buffer was allocated by the Formatter object and will be not be allocated by the Formatter object. If you need persistence beyond that of the next call to Formatter::AddInputMessage() or beyond the scope of the Formatter object, allocate memory and copy the buffer.

There are no error handling functions for GetMsgBuffer().

Example

See Sample Program 1: msgtest.cpp.

See Also

AddInputMessage(), GetMsgLength()

GetMsgLength

Overview

GetMsgLength() returns the length (in bytes) of the internal message buffer returned by a call to OutMsg::GetMsgBuffer().

Note

THREAD SAFETY NOTE: GetMsgLength() returns the length (in bytes) of the internal message buffer within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int OutMsg::GetMsgLength();
```

Parameters

None

Return Value

Returns the length (in bytes) of the internal message buffer.

There are no error handling functions for GetMsgLength().

Example

See Sample Program 1: msgtest.cpp.

See Also

GetMsgBuffer()

OutMsgGroup Class Member Functions

GetMsg

Overview

GetMsg() returns a pointer to an output message in an output message group.

Note

THREAD SAFETY NOTE: GetMsg() returns a pointer to an output message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
OutMsg* OutMsgGroup::GetMsg(int index);
```

Parameters

Name	Type	Input/ Output	Description
index	int	Input	Index into OutMsg array inside OutMsgGroup.

Return Value

Returns a pointer to the OutMsg at the index position in the internal outMsgs array; NULL if not present. Currently, an OutMsgGroup only holds one OutMsg (zero(0) is the only valid index value).

There are no error-handling functions for GetMsg().

Example

See Sample Program 1: msgtest.cpp.

GetMsgCount

Overview

GetMsgCount() returns the number of messages in an output message group.

Note

THREAD SAFETY NOTE: GetMsgCount() returns the number of messages in an output message group within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int OutMsgGroup::GetMsgCount();
```

Parameters

None

Return Value

Returns the number of messages in an output message group. There will be zero (0) or 1 message(s) in an output message group.

There are no error-handling functions for GetMsgCount().

Example

See Sample Program 1: msgtest.cpp.

See Also

GetMsg()

GetParsedOutMsgCount

Overview

GetParsedOutputMsgCount() returns count of output messages constructed by Formatter.

Note

THREAD SAFETY NOTE: GetParsedOutMsgCount() returns the number of messages in an output message group within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int OutMsgGroup::GetParsedOutMsgCount()
```

Parameters

None

Return Value

GetParsedOutMsgCount returns an integer that represents the number of output messages.

Example

See Sample Program 1: msgtest.cpp.

See Also

GetMsg()

GetParsedOutMsg

Overview

GetParsedOutMsg() returns the parsed output message at the specified index. After this is called, the parsed message and field API functions from pmsg.h and pfield.h may be used.

Note

THREAD SAFETY NOTE: GetParsedOutMsg() returns the number of messages in an output message group within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
ParsedMessage OutMsgGroup::GetParsedOutMsg()
```

Parameters

None

Return Value

ParsedMessage *. This returns a pointer to the ParsedMessage class from with those API calls can be made.

Example

See Sample Program 1: msgtest.cpp.

See Also

GetMsg()

ParsedField Class Member Functions

GetInfo

Overview

GetInfo() returns a pointer to the name of the field in a parsed message.

Note

THREAD SAFETY NOTE: GetInfo() returns a pointer to the name of the field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* ParsedField::GetInfo();
```

Parameters

None

Return Value

Returns a pointer to the name of the specified field.

There are no error handling functions for GetInfo().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse()

GetAsciiValue/GetStringValue

Overview

GetAsciiValue() returns the ASCII value of the specified field in a parsed message. GetStringValue() performs the same function that GetAsciiValue does except it is a portable version that should be used if porting across differing types of platforms.

Note

THREAD SAFETY NOTE: GetAsciiValue() returns the ASCII value of the specified field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* ParsedField::GetAsciiValue(int* pDataLength)
```

Parameters

Name	Type	Input/ Output	Description
pDataLength	int*	Output	Address of integer variable to receive data length.

Return Value

Returns the value of the specified field in ASCII format.

There are no error handling functions for GetAsciiValue().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse()

GetValue

Overview

GetValue() returns the value of the specified field in a parsed message in its original data type. This function returns the buffer of a formatted message.

Note

THREAD SAFETY NOTE: GetValue() returns the value of the specified field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
char* ParsedField::GetValue(int* pDataType,
int* pDataLength);
```

Parameters

Name	Type	Input/ Output	Description
pDataType	int*	Output	Address of integer variable to receive data type. See Supported Data Types in the Appendix for a list of supported data types for this release.
pDataLength	int*	Output	Address of integer variable to receive data length

Return Value

Returns the value of the specified field in its original data type.

There are no error handling functions for GetValue().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse()

GetFmtValue

Overview

This function returns the buffer of a formatted message.

Note

THREAD SAFETY NOTE: GetFmtValue() returns the value of the specified field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
void GetFmtValue(char *pBuffer)
```

Parameters

Name	Type	Input/ Output	Description
pBuffer	char *	Output	Return buffer

Return Value

None

GetFmtValueLen

Overview

Returns length of formatted submessage in bytes.

Note

THREAD SAFETY NOTE: GetFmtValueLen() returns the value of the specified field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int ParsedFieldInterface::GetFmtValueLen()
```

Parameters

None

Return Value

The integer value containing the length in bytes of the formatted submessage.

GetByteOffset

Overview

Returns byte offset in original message where field was found.

Note

THREAD SAFETY NOTE: GetByteOffset() returns the value of the specified field in a parsed message within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int ParsedFieldInterface::GetByteOffset()
```

Parameters

None

Return Value

The integer value containing the byte offset in original message where the field was found.

ParsedMessage Class Member Functions

GetCompCount

Overview

GetCompCount() returns the number of components (messages or fields) in a parsed message.

Note

THREAD SAFETY NOTE: GetCompCount() returns the number of components in a parsed message within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 173 for more information.

Syntax

```
int ParsedMessage::GetCompCount();
```

Parameters

None

Return Value

Returns the number of the components (other parsed messages or fields) in a parsed message.

There are no error handling functions for GetCompCount().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse()

GetMsgComp

Overview

GetMsgComp() returns the message component at the specified index.

Note

THREAD SAFETY NOTE: GetMsgComp() returns the message component within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 173 for more information.

Syntax

```
ParsedMessage* ParsedMessage::GetMsgComp(int index);
```

Parameters

Name	Type	Input/ Output	Description
index	int	Input	Index of parsed message component to return.

Return Value

Returns the parsed message component at the specified index; NULL if supplied with a bad index.

If the ParsedMessage is of type FLAT_FORMAT, then the function returns NULL. In this case, the user should call GetFieldComp().

There are no error handling functions for GetMsgComp().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse(), GetFieldComp()

GetInfo

Overview

GetInfo() returns the format name of the parsed message.

Note

THREAD SAFETY NOTE: GetInfo() returns the format name of the parsed message within the current thread. See the *Using the Thread Safe Release of the Formatter Engine* on page 173 for more information.

Syntax

```
char* ParsedMessage::GetInfo(int* pMsgType);
```

Parameters

Name	Type	Input/ Output	Description
pMsgType	int*	Output	Address of integer variable to receive type of parsed message: FLAT_FORMAT or COMPOUND_FORMAT.

Return Value

Returns the format name of the parsed message.

There are no error handling functions for GetInfo().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse()

GetFieldComp

Overview

GetFieldComp() returns the component field at the index specified.

Note

THREAD SAFETY NOTE: GetFieldComp() returns the component field within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
ParsedField* ParsedMessage::GetFieldComp(int index);
```

Parameters

Name	Type	Input/ Output	Description
index	int	Input	Index of the field to return.

Return Value

Returns a pointer to the field at the specified index; NULL if supplied with a bad index.

If the ParsedMessage is of type COMPOUND_FORMAT, the function returns NULL. In this case, the user should call GetMsgComp().

There are no error handling functions for GetFieldComp().

Example

See Sample Program 2: apitest.cpp.

See Also

Parse(), GetMsgComp()

GetFmtValLen

Overview

Returns length in bytes of formatted submessage.

Note

THREAD SAFETY NOTE: GetFmtValLen() returns the component field within the current thread. See the *How to Use the Thread Safe Release* of the Formatter Engine section for more information.

Syntax

```
int ParsedMessage::GetFmtValLen()
```

Parameters

None

Return Value

The integer containing the length in bytes of the formatted submessage.

GetFmtVal

Overview

Returns buffer containing the formatter submessage.

Note

THREAD SAFETY NOTE: GetFmtVal() returns the component field within the current thread. See the How to Use the Thread Safe Release of the Formatter Engine section for more information.

Syntax

```
void ParsedMessage::GetFmtVal(char * pBuffer)
output, char *
```

Parameters

Name	Type	Input/ Output	Description
pBuffer	char *	Output	Return buffer

Return Value

Buffer containing the formatter submessage.

User Exit API Functions

The User Exit API has two parts: the Lookup Interface and the Exit Function Developer Interface. In both parts, the API defines how the user is expected to construct functions callable by the Formatter rather than define functions to be called by the user application.

To use User Exits, include the nnexit.h header file, which includes the ses.h, nnparsedflds.h, and nnexitret.h header files. Additional database-specific header files are defined in the database handle table in the User Exit Function Specification section. You must nclude one of the files from that table in a User Exit routine if the session handle will be used to access the current connection to the database.

WARNING!

If you use the thread-safe Formatter functions in a multi-threaded environment, you must write thread-safe User Exit functions.

WARNING!

Pointers to data returned to Formatter functions must be thread-specific. Pointers to shared data SHOULD NOT be returned by User Exits.

User Exit Lookup Interface

The User Exit Lookup Interface facilitates run-time lookup and registration of User Exit and Exit Cleanup functions. When a Formatter instance encounters a User Exit as part of the reformatting process, the Formatter tries to resolve the exit name into a callable function address. Since the function name and address are developed outside the scope of the Formatter, the Formatter has to ask (call) the user application to provide the function address. This means the Formatter needs to know the name of the Lookup function, its arguments, and possible return values.

To facilitate a User Exit Lookup, a function must be defined with a specific name, arguments, and return type. Internally a stub lookup function is defined that is replaceable by a user application. The user application doesn't have to do anything special to replace the function. It creates the function, compiles it, and then links it with the MQSeries Integrator libraries.

NNGetUserExitFuncPtrs()

Overview

The application developer must define a function named NNGetUserExitFuncPtrs to replace the lookup stub if a Formatter instance is to be able to resolve a function address. The user application need only define this function if it expects the Formatter to call a User Exit. A call to Formatter::Reformat() returns an error if an undefined User Exit function is encountered on an output format control as part of the reformat process.

Function NNGetUserExitFuncPtrs() is required to return User Exit and Exit Cleanup function pointers given an exit function name.

Syntax

```
void NNGetUserExitFuncPtrs(
    char* acFuncName,
// <in> exit function name
    NN_EXIT_FUNC_t &rUEptr,
// <out> exit function pointer
    NN_EXIT_CLEANUP_FUNC_t&rUEClUpPtr);
// <out> exit clean up func pointer
```

Parameters

Name	Type	Input/ Output	Description
acFuncName	char*	Input	Exit function name associated with an output format control encountered during the reformat process.
rUEptr	NN_EXIT_ FUNC_t &	Output	Exit function pointer. The Formatter expects this parameter to be returned as NULL or a valid exit function address of type NN_EXIT_FUNC_t (MQSeries Integrator Exit Function type).
rUEClUpPtr	NN_EXIT_ CLEANUP_ FUNC_t &	Output	Exit Cleanup function pointer. If rUEptr is not NULL, this parameter can be NULL. A Cleanup function is used to cleanup any memory allocations not already cleaned up as part of the User Exit.

Remarks

The body of NNGetUserExitFuncPtrs() is user-defined.

Example

As an example, suppose a user application creates User Exit functions UEfuncA() and UEfuncB(), and Cleanup function UEClUpFuncA() is

associated with UEfuncA(). An example body for function NNGetUserExitFuncPtrs() would look like:

```
.....
#include "nnexit.h"
.....

void
NNGetUserExitFuncPtrs(
    char* acFuncName,
// <in> exit function name
    NN_EXIT_FUNC_t &rUEptr,
// <out> exit function pointer
    NN_EXIT_CLEANUP_FUNC_t &rUEClUpPtr)
// <out> exit clean up func pointer
{
    if (strcmp(acFuncName, "UEfuncA") == 0) {
           rUEptr = UEfuncA;
           rUEClUpPtr = UEClUpFuncA;
    } // endif UEfuncA

    else if (strcmp(acFuncName, "UEfuncB") == 0) {
           rUEptr = UEfuncB;
           rUEClUpPtr = NULL;
    } // endelse UEfuncB

    else {
           rUEptr = NULL;
           rUEClUpPtr = NULL;
    } // else no match
}
```

Note

The nnexit.h header file must be included to resolve NN_EXIT_FUNC_t and NN_EXIT_CLEANUP_FUNC_t types. These types are explained in subsequent sections.

User Exit Function Developer Interface

The User Exit Function Developer Interface specifies the calling and return conventions for creating User Exit functions acceptable to the Formatter.

User Exit Function Specification

Overview

The User Exit function developer can only create non-member function User Exits. The exit function developer is restricted to creating exit functions that match a defined calling convention.

A typedef exists to ensure the User Exit function created by the user application conforms to the required call convention. If not, a compiler error will be generated if it is used to initialize the User Exit return value, rUEptr, in lookup function NNGetUserExitFuncPtrs(). The typedef can be found in the nnexit.h header file.

Syntax

```
typedef NNExitRet (*NN_EXIT_FUNC_t)
       (const DbmsSession &rSession,
        const NNParsedFields &rFields);
```

Parameters

Name	Type	Input/ Output	Description
rSession	const DbmsSession &	Input	Current DBMS session. When a User Exit is called, the current database session object and the set of parsed input fields are passed to the User Exit function as input arguments. The User Exit can access the current session handle by referencing the DbmsSession member function rSession.Handle().
rFields	const NNParsedFields &	Input	Object that allows access to all parsed input field values using NNParsedFields member functions.

Remarks

A User Exit can access a parsed input message field value by referencing any of the NNParsedFields Class member functions. The member functions of this class allow access to these values.

When using rSession.Handle(), the type of return value must be cast to the appropriate database handle type as shown in the following table:

Database	Header File	Handle Type
Oracle	orases.h	OraSesHandle*
Sybase	sybdb.h sybfront.h	DBPROCESS*
SQLServer	sqldb.h	DBPROCESS*

Example

For example, if the MQSeries Integrator portion of the user application has an open connection to a Sybase RDBMS and the User Exit function needs to use the connection, a fragment of the user exit code would look like:

```
.....
.....
DBPROCESS* pDBsession;
....
....
pDBsession = (DBPROCESS*)rSession.Handle();
....
```

NNParsedFields Class Member Functions

GetFieldAscii/GetFieldString

Overview

GetFieldAscii() returns the ASCII value of the specified input parsed field. GetFieldString() performs the same function as GetFieldAscii, except it is a portable version that should be used if porting across differing types of platforms.

Syntax

```
const char * GetFieldAscii(char * pFieldName, int iIndex)
const
```

Parameters

Name	Type	Input/ Output	Description
pFieldName	char * *	Input	Name of input parsed field to return ASCII value for.
iIndex	int	Input	Number indicating the instance (zero-based index) of input parsed field to return value for (in cases where a field name is used more than once in one or more input formats to construct the output message).

Return Value

Returns the ASCII value of the specified parsed input field.

GetCurrInFldName

Overview

GetCurrInFldName() returns the name of the input field associated with the current output field for which the user exit is being called.

Syntax

```
const char * GetCurrInFldName() const
```

Parameters

None

Return Value

Returns the name of the input field.

GetCurrOutFldName

Overview

GetCurrOutFldName() returns the name of the current output field for which the user exit is being called.

Syntax

```
const char * GetCurrOutFldName() const
```

Parameters

None

Return Value

Returns the name of the output field.

GetCurrInFldData

Overview

GetCurrInFldData() returns the raw value of the input field associated with the current output field for which the user exit is being called.

Syntax

```
const char * GetCurrInFldData() const
```

Parameters

None

Return Value

Returns the raw data value of the input field.

GetCurrInFldAsciiData

Overview

GetCurrInFldAsciiData() returns the ASCII value of the input field associated with the current output field for which the user exit is being called.

Syntax

```
const char * GetCurrInFldAsciiData() const
```

Parameters

None

Return Value

Returns the ASCII value of the input field.

GetCurrInFldLength

Overview

GetCurrInFldLength() returns the length of the data (in its original data type) of the input field associated with the current output field for which the user exit is being called.

Syntax

```
const int GetCurrInFldLength() const
```

Parameters

None

Return Value

Returns the length of the data of the input field in its original type.

GetCurrInFldType

Overview

GetCurrInFldType() returns the original data type of the input field associated with the current output field for which the user exit is being called.

Syntax

```
const int GetCurrInFldType() const
```

Parameters

None

Return Value

Returns the original data type of the input field.

GetUserExitRoutineName

Overview

GetUserExitRoutineName() returns the name of the user exit routine specified for the corresponding output format control.

Syntax

```
const char * GetUserExitRoutineName() const
```

Parameters

None

Return Value

Returns the name of the user exit routine.

Example

The following is an example of using GetUserExitRoutineName() to retrieve the name of the user exit routine:

```
NNExitRet MyUserExit(const DbmsSession &rSession,
                     const NNParsedFields &rFields)
{
...
   const char* pUserExitRoutineName =
               rFields.GetUserExitRoutineName();
...
}
```

User Exit Return Object

The NNExitRet class only supports scalar return result types long, double, and byte array. Each type has an associated constructor and assignment operator. Internally, NNExitRet stores the return value, the return status, and an error message.

An error status can be passed into several of the methods. MQSeries Integrator defines general purpose status value NN_ERSTATUS_ERROR to indicate an error has occurred, and NN_ERSTATUS_OK to indicate success. The values of these error status constants is defined in header file nnexitret.h. The user is allowed to override either of these error status values. However, the User Exit developer is discouraged from doing so because the Formatter checks to see if the return status is equal to NN_ERSTATUS_OK. If any other value is returned, the Formatter will assume an error has occurred and fail the reformat process. If possible, the user application developer should treat these MQSeries Integrator return status settings as reserved values.

Constructors

Constructor (Long Return Result Type)

Syntax

```
NNExitRet(
      const long lVal, // <in> long return value const int
      iERstatus);      // <in> return error status
```

Parameters

Name	Type	Input/ Output	Description
lVal	const long	Input	Long return value.
iERstatus	const int	Input	Return error status.

Example

```
NNExitRet
UE_LongEx(
// <out> exit return object
const DbmsSession rSession,
// <in> current Integrator session
const ParsedFields &rFields)
// <in> parsed input msg fields
{
    .....
    long l;
    .....
    .....
    return(NNExitRet(l, NN_ERSTATUS_OK));
}
```

Constructor (Double Return Result Type)

Syntax

```
NNExitRet(
    const double dVal,    // <in> double return value
    const int iERstatus); // <in> return error status
```

Parameters

Name	Type	Input/ Output	Description
dVal	const double	Input	Double return value.
iERstatus	const int	Input	Return error status.

Example

```
NNExitRet
UE_DoubEx(
// <out> exit return object
const DbmsSession rSession,
// <in> current Integrator session
const ParsedFields &rFields)
// <in> parsed input msg fields
{
    .....
    double d;
    .....
    .....
    return(NNExitRet(d, NN_ERSTATUS_OK));
}
```

Constructor (Byte Array Return Result Type)

Syntax

```
NNExitRet(
    const char* pabVal,  // <in> pointer to array of bytes
    const long lValLen,  // <in> length of array of bytes
    const int iERstatus);// <in> return error status
```

Parameters

Name	Type	Input/ Output	Description
pabVal	const char*	Input	Pointer to byte array.
lValLen	const long	Input	Length of byte array.
iERstatus	const int	Input	Return error status.

Remarks

Internally, memcpy() is used to copy the bytes from the input character array to the return objects internal data member. This ensures that a non-ASCII or ASCII string containing control characters can be duplicated in the return result.

Example

```
NNExitRet
UE_ByteArrayEx(
// <out> exit return object
   const DbmsSession rSession,
// <in> current session
   const ParsedFields &rFields)
// <in> parsed input msg fields
{
    .....
    char acStr = new char [256];
    .....
    .....
    NNExitRet oER(acStr, 256, NN_ERSTATUS_OK);
    delete [] acStr;
    return oER;
}
```

Constructor (General Case)

Overview

This constructor provides the most general case. This constructor coupled with other member functions provides the most flexible way to define a User Exit return, assign the return result, assign the error status, and define an error message, if applicable.

Syntax

```
NNExitRet();
```

Parameters

None

Remarks

In addition to the previously described constructors, there is also a copy constructor. It takes a NNExitRet reference as an input argument and returns a NNExitRet instance.

Example

The following example illustrates usage and flexibility:

```
NNExitRet
UE_FlexEx(
// <out> exit return object
const DbmsSession rSession,
// <in> current Integrator session
const ParsedFields &rFields)
// <in> parsed input msg fields
{
    .....
    NNExitRet oER;
     // instance of exit return object
    char* pacFldVal;
     // pointer to array of chars containing field value

    // Get the field value...
    pacFldVal = rFields.GetFieldAscii("InField1", 0);

    // If field value is numeric then...
    .....
    {
        // If field value has decimal then...
        .....
        {
            oER = atof(pacFldVal);
        }
        // endif numeric field has decimal

        // Else no decimal so...
        .....
```

```
            {
                oER = atol(pacFldVal);
            }
            // endelse no decimal
        }
        // endif field is numeric

        // Else non numeric field so...
        .....
        {
            oER.SetByteArrayValue(pacFldVal,
                                  strlen(pacFldVal));
        }
        // endelse field is non numeric

        return oER;
    }
```

Operator Overloads

The equals '=' operator is overloaded for NNExitRet long and double values. The General Case Constructor example illustrates the use of long and double overloaded equals operator methods. All equals operator overload functions initialize (and reinitialize) error status and error message to NN_ERSTATUS_OK and NULL respectively.

Other Public Methods

Other public methods include NNExitRet::SetByteArrayValue() and NNExitRet::SetError().

SetByteArrayValue

Overview

SetByteArrayValue() allows the User Exit developer to set the exit return instance value to a byte array.

Syntax

```
void NNExitRet::SetByteArrayValue(
    const char* pabVal,  // <in> pointer to array of bytes
    const long ILen);    // <in> length of array of bytes
```

Parameters

Name	Type	Input/ Output	Description
pabVal	const char*	Input	Pointer to byte array.
Ilen	const long	Input	Length of byte array. If the array is a NULL-terminated ASCII string, and the NULL termination is a valid part of the return result, then the length should include the NULL terminator.

Example

See the example for NNExitRet()

SetError

Overview

SetError() allows the user to set the return error status and error message. The error message is optional.

Note

NN_ERSTATUS_OK is not allowed as a valid iERstatus value and will be interpreted as NN_ERSTATUS_ERROR.

Syntax

```
void NNExitRet::SetError(
    const int iERstatus,    // <in> exit return status
    const char* pMsg);      // <in> error message
```

Parameters

Name	Type	Input/ Output	Description
iERstatus	const int	Input	Exit return status.
pMsg	const char*	Input	Error message. A NULL value is the default and is a valid (acceptable) value.

Example

```
NNExitRet
UE_Example(
// <out> exit return object
const DbmsSession rSession,
// <in> current Integrator session
const ParsedFields &rFields)
// <in> parsed input msg fields
{
   .....
   NNExitRet oER;
   // instance of exit return object
   char* pacFldVal;
   // pointer to array of chars containing field value

   // Look up field value...
   pacFldVal = rFields.GetFieldAscii("InField1", 0);

   // If field not found then...
   if (pacFldVal == NULL) {
       oER = "";
       oER.SetError(NN_ERSTATUS_ERROR,
       "InField1 not found!");
   }
```

```
   else {
      oER.SetByteArrayValue(pacFldVal,
      strlen(pacFldVal));
   }

   return oER;
}
```

User Exit Cleanup Function Specification

Overview

The User Exit developer can optionally create non-member User Exit Cleanup functions. The Exit Cleanup function developer is restricted to creating functions that match a defined calling convention.

A typedef exists to ensure the User Exit Cleanup function conforms to the required call convention. If not, a compiler error will be generated if it is used to initialize the User Exit return value, rUEClUpPtr, in lookup function NNGetUserExitFuncPtrs(). The typedef can be found in header file nnexit.h.

Syntax

```
typedef long(*NN_EXIT_CLEANUP_FUNC_t)();
```

Parameters

N/A

Remarks

The Formatter checks to see if the return value is equal to NN_ERSTATUS_OK. If not, the Formatter assumes an error condition has occurred and a non-fatal error condition is set.

Example

```
.....
.....
UserObject* pUObj;
.....
.....
long
UEClUp_Example(void)         // <out> error status
{
    if (pUObj) delete pUObj;
    return NN_ERSTATUS_OK;
}
```

User Exit API Summary

To create and use User Exit functions:

1. Create User Exit and User Exit Cleanup functions. Functions must conform to the NN_EXIT_FUNC_t and NN_EXIT_CLEANUP_FUNC_t types defined in header file nnexit.h.
2. Create a routine named NNGetUserExitFuncPtrs() so that a Formatter instance can look up the function pointers for the User Exit and User Exit Cleanup functions given an exit function name.
3. In the Formatter GUI, specify the name of the exit routine in the Exit Routine field on the Field Format Output Control Tool screen.

The following pseudo-code describes the behavior of a Formatter instance when it encounters a User Exit as part of the reformat process:

```
user calls Formatter::Reformat()
formatter detects user exit defined as part of output
     format control
formatter checks registry to determine if already cached
IF not in registry THEN
   call NNGetUserExitFuncPtrs()
   IF exit function pointer is not NULL THEN
      exit function and exit clean up function pointers
          added to registry
    ENDIF
ENDIF
IF exit function pointer is not NULL THEN
   call user exit
   IF user exit returns NN_ERSTATUS_OK error status THEN
      IF user exit clean up defined THEN
         call user exit clean up function
         IF user exit clean up fails THEN
            set nonfatal error condition
         ENDIF
      ENDIF
   ELSE
      set fatal error condition
   ENDIF
ELSE
   set fatal error condition
END
```

Note

A User Exit Cleanup failure does not cause the Formatter reformat process to fail.

Rebuilding msgtest and ruleng for User Exits

To use the Formatter program msgtest or the Rules Engine daemon ruleng with User Exits, you must relink these executables with your library of User Exit functions. To do this, follow these steps:

1. Edit the Makefile in the examples directory.

   For Oracle installations, copy the file Makefile.solaris-2.4.SparkWorks3.0.1.Oracle-7.1.6 to Makefile.

   For Sybase installations, copy the file Makefile. Solaris-2.4.SparkWorks3.0.1.Sybase-4.9 to Makefile.

   The copied Makefile is a sample that provides a good place to start customizing a Makefile for rebuilding msgtest and ruleng. This Makefile assumes a Solaris 2.4 operating system with a SparcWorks 3.0.1 C++ compiler. The Makefile includes some hard-coded paths that must be replaced with the paths for your particular compiler and database installation.

   Substitute the following paths with your own paths:

   `ROOT`
   Location of your MQSeries Integrator installation

   `COMPILER.inc`
   Location of your compiler include files

   `COMPILE.lib`
   Location of your compiler libraries

   `DB_HOME`
   Location of your Sybase or Oracle installation

   After you substitute your paths for the existing ones, add your object library of user exit routines ($(YOUR_LIBRARY)) to the link stream for msgtest and ruleng. You can probably place this reference after the last reference to NEON.a.

```
ruleng:ruleng.o $(NEON.a)
$(C++) ruleng.o \
$(GLDFLAGS) \
-lruleng \
$(NEON.lib) \
$(DBMS.lib) \
$(NEON.A) \
$(YOUR_LIBRARY) \
$(DBLIB) \
- lnsl \
- lm \
- o $@
```

2. Make msgtest and ruleng

   To make msgtest and ruleng, issue the following commands:

   make msgtest

make ruleng

These executables will be placed in the current directory. You may copy them to the directory where your MQSeries Integrator binaries reside (bin).

Note

If you encounter linking errors that indicate duplicate symbols, there are User Exit functions named the same as existing MQSeries Integrator functions. Rename the User Exit functions to avoid conflicts and recompile.

User Callback API Functions

The User Callback API provides a simple, flexible mechanism for defining functions Formatter can call to perform various functions such as user-defined type input field validation. This API consists of two parts– the Callback class definitions and the Callback object collection– both defined in the nnuserfunction.h header file.

You define User Callbacks as methods in a callback class derived from a NEON-defined abstract base class. Objects of your callback class are then passed to Formatter at construction.

In addition to Formatter-created data, user callbacks can also be passed user-defined parameters: static (predefined in the database) and dynamic (created by the user's code at run time). Static data is defined at format definition time and referred to as name/value pairs in the Formatter GUI. Dynamic data is created by the user application at run time and referred to as Parameter Name in the Formatter GUI.

To use the User Callback API, include the nnuserfunction.h header file, which includes the ses.h and nnparsedflds.h header files. Additional database-specific header files are defined in the following database handle table. You must include one of the files from that table in a file that defines your callbacks if the session handle will be used to access the current connection to the database.

Database	Header File	Handle Type To Cast To
Oracle	orases.h	OraSesHandle*
Sybase	sybdb.h sybfront.h	DBPROCESS*
SQLServer	sqldb.h	DBPROCESS*

See *Sample Program 1: msgtest.cpp* on page 307 for an example of how to use User Callbacks.

Note

THREAD SAFETY NOTE: The callback and lookup methods described later in this section must be written to be “thread-safe” since Formatter may have multiple threads running through it. For the purposes of this section, thread-safety is satisfied by not using static or global variables.

WARNING!

Formatter performance can be severely degraded if callback and lookup methods are not written with care.

User Callback API Structures

NameValuePair

Overview

NameValuePair is the basic element of the array type passed into the callback methods described. In general, a database object, like a parse control, will have a set of these pairs defined, and that set is collected into an array of NameValuePairs to pass to the callbacks. The last element of such an array will have its name and value fields set to NULL. This data is the static, predefined data passed to the callback.

Syntax

```
struct NameValuePair
{
public:
    const char* name;
    const char* value;

    NameValuePair();
    // default, stringLength = NAME_LENGTH + 1
    NameValuePair(int stringLength);
    NameValuePair( const NameValuePair& rhs );
    // copy
    NameValuePair& operator=( const NameValuePair& rhs );
    // assignment
    ~NameValuePair();

    // Deallocate name and value, set them to NULL.
    // Useful if you make an array of these, and want to
    // set the last element as having NULL fields to mark
    // the end of the array.
    void MakeNull();
```

```
    // set name = inName, value = inValue
    void Set( const char* inName, const char* inValue );

private:
    int strLength;
};
```

Parameters

Name	Type	Description
name	const char *	The "purpose" of the pair. In general, a callback method which receives an array of these will traverse the array element by element, and use the value portion of the element according to the purpose described in name.
value	const char *	The "value" of the pair. In general, a callback method which receives an array of these will traverse the array element by element, and use the value portion of the element according to the purpose described in name.

Member Functions

Name	Prototype	Description
NameValuePair	()	Default constructor.
NameValuePair	(int stringLength)	Alternate constructor.
NameValuePair	(const NameValuePair& rhs)	Copy constructor.
operator=	(const NameValuePair& rhs)	Assignment operator.
~NameValuePair	()	Destructor.
MakeNull	()	See description.
Set	(const char* inName, const char* inValue)	Sets name and value to the input values.

NameValuePair Member Functions

NameValuePair (Default Constructor)

Overview

Default constructor. Length of name and value will be set to NAME_LENGTH + 1.

Syntax

```
NameValuePair::NameValuePair()
```

Parameters

None

NameValuePair (Alternate Constructor)

Overview

Alternate constructor. Length of name and value will be set to stringLength.

Syntax

```
NameValuePair::NameValuePair(int stringLength)
```

Parameters

Name	Type	Input/ Output	Description
stringLength	int	Input	Length of name and value.

NameValuePair (Copy Constructor)

Overview

Copy constructor.

Syntax

```
NameValuePair::NameValuePair(const NamevaluePair& rhs)
```

Parameters

Name	Type	Input/ Output	Description
rhs	const NameValuePair&	Input	Address of name/value pair to be copied.

NameValuePair (Assignment Operator)

Overview

Assignment operator.

Syntax

```
NameValuePair::NameValuePair& operator=(
                    const NameValuePair& rhs)
```

Parameters

Name	Type	Input/ Output	Description
rhs	const NameValuePair&	Input	Address of name/value pair to be copied.

NameValuePair (Destructor)

Overview

Destructor.

Syntax

```
NameValuePair::~NameValuePair()
```

Parameters

None

MakeNull

Overview

Deallocate the name and value fields and set them to NULL. For instance, MakeNull() is used to mark the last element of an array of NameValuePairs.

Syntax

```
void NameValuePair::MakeNull()
```

Parameters

None

Set

Overview

Sets name and value to the input values.

Syntax

```
void NameValuePair::Set(const char* inName,
                        const char*inValue)
```

Parameters

Name	Type	Input/ Output	Description
inName	const char*	Input	Value for Name parameter.
inValue	const char*	Input	Value for Value parameter

User Callback Class Definition

User callback functions are defined as methods in a callback class. You define a callback class by deriving from one of the NEON-supplied abstract base classes which declares the minimal set of methods that must be defined for your derived class.

Depending on the feature you are working with, you will only need to derive from one of the three abstract base classes.

The class hierarchy is as follows:

NNUserFunction

- NNGenericUserFunction abstract base class
  - UserDerivedCallbackClass
- NNDBUserFunction abstract base class
  - UserDerivedCallbackClass
- NNDBFieldsUserFunction abstract base class
  - UserDerivedCallbackClass

class NNUserFunction

NNUserFunction is the class from which all callback classes are derived. It provides an all-encompassing type for passing callback objects into the collection described in the “User Callback Lookup Interface” section.

class NNGenericUserFunction: public NNUserFunction

NNGenericUserFunction is one of the three abstract base classes from which users will derive their own caballed classes. NNUserFunction is used when MQSeries Integrator DbmsSession and MQSeries Integrator NNParsedFields aren't needed.

class NNDBUserFunction: public NNUserFunction

NNDBUserFunction is another of the three abstract base classes from which users will derive their own caballed classes. NNGenericUserFunction is used when MQSeries Integrator DbmsSession is needed.

class NNDBFieldsUserFunction: public NNUserFunction

NNDBFieldsUserFunction is the last of the three abstract base classes from which users will derive their own callback classes. NNDBUserFunction is used when both MQSeries Integrator DbmsSession and MQSeries Integrator NNParsedFields are needed.

Class Name	Dbms Session?	Parsed Fields?	User Parameters?
NNGenericUserFunction	No	No	Yes
NNDBUserFunction	Yes	No	Yes
NNDBFieldsUserFunction	Yes	Yes	Yes

class <UserDerivedCallbackClass>: public <NN...UserFunction>

UserDerivedCallbackClass is a user-derived class named whatever you want. It must inherit publicly from one of the three abstract base classes listed above and define all pure virtual functions for the abstract base class.

NNUserFunction

Overview

NNUserFunction is the top of the callback class hierarchy, not to be used directly. It provides a general class for passing callback objects to NNFunctionKeyPairCollection.

Syntax

```
class NNUserFunction
{
public:
    NNUserFunction(){}
        virtual ~NNUserFunction(){}
};
```

Parameters

None

Remarks

Do not use this class directly; subclass your callback class from one of the three abstract base classes.

NNGenericUserFunction

Overview

NNGenericUserFunction is the most general of the three abstract base classes. Derive your user callback function from this class if the feature you are working with does not pass a database session or parsed fields to your callbacks.

Syntax

```
class NNGenericUserFunction : public NNUserFunction
{
public:
    NNGenericUserFunction(){}
    virtual ~NNGenericUserFunction(){}

    virtual int Callback () = 0;
    virtual int Callback (
        NameValuePair* nameValuePairArray ) = 0;
    virtual int Callback ( void* userRuntimeData ) = 0;
    virtual int Callback (
        NameValuePair* nameValuePairArray,
        void* userRuntimeData ) = 0;

    inline virtual void* RuntimeDataLookup(
        const char* parmName)
        { return 0; }

};
```

Member Functions

Name	Prototype	Description
Callback	()	See description.
Callback	(NameValuePair* nameValuePairArray)	See description.
Callback	(void* userRuntimeData)	See description.
Callback	(NameValuePair* nameValuePairArray, void* userRuntimeData)	See description.
RuntimeDataLookup	(const char* parmName)	See description.

NNGenericUserFunction Member Functions

Callback (No Parameters)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are no user parameters to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNGenericUserFunction::Callback()
```

Parameters

None

Callback (nameValuePairArray)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are name/value pairs to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNGenericUserFunction::Callback(
       NameValuePair* nameValuePairArray)
```

Parameters

Name	Type	Input/ Output	Description
nameValuePair Array	NameValue Pair*	Input	Array of name/value pairs retrieved from the database.

Callback (userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are both name/value pairs and user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNGenericUserFunction::Callback(
      NameValuePair* nameValuePairArray,
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
nameValuePair Array	NameValue Pair*	Input	Array of name/value pairs retrieved from the database.
userRuntimeData	void*	Input	User-allocated run-time data obtained by the RuntimeDataLookup() method.

Callback (userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there is user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNGenericUserFunction::Callback(
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
userRuntimeData	void*	Input	User-allocated run-time data obtained by the RuntimeDataLookup() method.

RuntimeDataLookup

Overview

Formatter will call RuntimeDataLookup() after looking up a callback object of this type to obtain a pointer to user-allocated run-time data, to be passed into one of the callback methods, as appropriate.

Syntax

```
void* NNGenericUserFunction::RuntimeDataLookup(
        const char* parmName)
```

Parameters

Name	Type	Input/ Output	Description
parmName	char*	Input	Formatter obtains this name, depending on the feature being supported, and passes it to Lookup(). Lookup() may or may not use it to determine the data address to pass back.

NNDBUserFunction

Overview

Derive from this class if the feature you are working with passes a database session to the callbacks, in addition to user parameters.

Syntax

```
class NNDBUserFunction : public NNUserFunction
{
public:
    NNDBUserFunction(){}
    virtual ~NNDBUserFunction(){}

    virtual int Callback (
            const DbmsSession& dbSession) = 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            NameValuePair* nameValuePairArray) = 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            void* userRuntimeData) = 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            NameValuePair* nameValuePairArray,
            void* userRuntimeData) = 0;

    inline virtual void* RuntimeDataLookup(
            const char* parmName)
        { return 0; }

};
```

Member Functions

Name	Prototype	Description
Callback	(const DbmsSession& dbSession)	See description.
Callback	(const DbmsSession& dbSession, NameValuePair* nameValuePairArray)	See description.
Callback	(const DbmsSession& dbSession, void* userRuntimeData)	See description.
Callback	(const DbmsSession& dbSession, NameValuePair* nameValuePairArray, void* userRuntimeData)	See description.
RuntimeDataLookup	(const char* parmName)	See description.

NNDBUserFunction Member Functions

Callback (dbSession)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are no user parameters to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.

Callback (dbSession, nameValuePairArray)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are name/value pairs to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      NameValuePair* nameValuePairArray)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
nameValuePair Array	NameValue Pair*	Input	Array of name/value pairs retrieved from the database.

Callback (dbSession, nameValuePairArray, userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are both name/value pairs and user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      NameValuePair* nameValuePairArray,
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
nameValuePair Array	NameValuePair*	Input	Array of name/value pairs retrieved from the database.
userRuntimeData	void*	Input	User-allocated run time data obtained by the RuntimeDataLookup() method.

Callback (dbSession, userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there is user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
userRuntimeData	void*	Input	User-allocated run-time data obtained by the RuntimeDataLookup () method.

RuntimeDataLookup

Overview

Formatter will call RuntimeDataLookup() after looking up a callback object of this type to obtain a pointer to user-allocated run-time data, to be passed into one of the callback methods as appropriate.

Syntax

```
void* NNDBUserFunction::RuntimeDataLookup(
        const char* parmName)
```

Parameters

Name	Type	Input/ Output	Description
parmName	char*	Input	Formatter obtains this name, depending on the feature being supported, and passes it to Lookup(). Lookup() may or may not use it to determine the data address to pass back.

NNDBFieldsUserFunction

Overview

Derive user callback functions from this class if the feature you are working with passes a database session and the set of all parsed fields to the callbacks, in addition to user parameters.

Syntax

```
class NNDBFieldsUserFunction : public NNUserFunction
{
public:
    NNDBFieldsUserFunction(){}
    virtual ~NNDBFieldsUserFunction(){}

    virtual int Callback (
            const DbmsSession& dbSession,
            const NNParsedFields& parsedFields) = 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            const NNParsedFields& parsedFields,
            NameValuePair* nameValuePairArray)= 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            const NNParsedFields& parsedFields,
            void* userRuntimeData) = 0;
    virtual int Callback (
            const DbmsSession& dbSession,
            const NNParsedFields& parsedFields,
            NameValuePair* nameValuePairArray,
            void* userRuntimeData)= 0;

    inline virtual void* RuntimeDataLookup(
            const char* parmName)
        { return 0; }

};
```

Member Functions

Name	Prototype	Description
Callback	(const DbmsSession& dbSession, const NNParsedFields& parsedFields)	See description.
Callback	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, NameValuePair* nameValuePairArray)	See description.

Name	Prototype	Description
Callback	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, void* userRuntimeData)	See description.
Callback	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, NameValuePair* nameValuePairArray, void* userRuntimeData)	See description.
RuntimeDataLookup	(const char* parmName)	See description.

NNDBFieldsUserFunction Member Functions

Callback (dbSession, parsedFields)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are no user parameters to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      const NNParsedFields& parsedFields)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsed Fields&	Input	The set of all parsed fields.

Callback (dbSession, parsedFields, nameValuePairArray)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are name/value pairs to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      const NNParsedFields& parsedFields,
      NameValuePair* nameValuePairArray)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsed Fields&	Input	The set of all parsed fields.
nameValuePair Array	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

Callback (dbSession, parsedFields, nameValuePairArray, userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there are both name/value pairs and user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      const NNParsedFields& parsedFields,
      NameValuePair* nameValuePairArray,
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsed Fields&	Input	The set of all parsed fields.
nameValuePair Array	NameValuePair*	Input	Array of name/value pairs retrieved from the database.
userRuntimeData	void*	Input	User allocated runtime data obtained by the RuntimeDataLookup() method.

Callback (dbSession, parsedFields, userRuntimeData)

Overview

A Formatter feature that uses objects derived from this class will call this method if there is user-allocated run-time data to pass to the method. The user's derived class defines this method, returning zero (0) on failure or non-zero on success.

Syntax

```
int NNDBUserFunction::Callback(
      const DbmsSession& dbSession,
      const NNParsedFields& parsedFields,
      void* userRuntimeData)
```

Parameters

Name	Type	Input/ Output	Description
dbSession	const DbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsed Fields&	Input	The set of all parsed fields.
userRuntimeData	void*	Input	User-allocated run-time data obtained by the RuntimeDataLookup() method.

RuntimeDataLookup

Overview

Formatter will call RuntimeDataLookup() after looking up a callback object of this type to obtain a pointer to user-allocated run-time data, to be passed into one of the callback methods as appropriate.

Syntax

```
void* NNDBUserFunction::RuntimeDataLookup(
        const char* parmName)
```

Parameters

Name	Type	Input/ Output	Description
parmName	char*	Input	Formatter obtains this name, depending on the feature being supported, and passes it to Lookup(). Lookup() may or may not use it to determine the data address to pass back.

User Callback Lookup Interface

When the Formatter calls a user callback, it attempts to look up the address of a callback object in the collection of callback objects passed to the Formatter at construction. The collection of objects holds object/key pairs. Depending on the feature being supported, Formatter will obtain a key, do a lookup on the object collection with that key, and receive the address of the corresponding callback object. Formatter will then call one of the methods defined for that object, depending on which parameters are available to pass to the callback method.

NNFunctionKeyPairCollection

Overview

NNFunctionKeyPairCollection is the collection type passed to the Formatter constructor, to register callback objects with the Formatter.

Users do not derive from this class, it is used as is.

Syntax

```
class NNFunctionKeyPairCollection
{
public:
    NNFunctionKeyPairCollection();
    ~NNFunctionKeyPairCollection();
    // non-virtual,
    // not meant to be subclassed

    int AddPair( NNUserFunction* funcObject,
                 const char* key );

    NNUserFunction* Lookup( const char* key );

private:
    NNFuncKeyColl* fkColl;
};
```

Member Functions

Name	Prototype	Description
AddPair	(NNUserFunction* funcObject, const char* key)	See description.
Lookup	(const char* key)	See description.

Private Data Members

Name	Prototype	Description
AddPair	N/A	See description.

NNFunctionKeyPairCollection Member Functions

AddPair

Overview

After constructing an object of this class, call AddPair() repeatedly for every funcObject/key pair required to support the feature you are working with.

Syntax

```
int NNFunctionKeyPairCollection::AddPair(
      NNUserFunction* funcObject,
      const char* key)
```

Parameters

Name	Type	Input/ Output	Description
funcObject	NNUserFunction*	Input	An object of a user's callback class derived from a NEON abstract base class.
key	const char*	Input	The key Formatter uses to look up the funcObject.

Return Value

Returns zero (0) on failure, and non-zero on success.

Lookup

Overview

Formatter calls this method to obtain a pointer to the required callback object.

Syntax

```
NNUserFunction* NNFunctionKeyPairCollection::Lookup(
      const char* key)
```

Parameters

Name	Type	Input/ Output	Description
key	const char*	Input	The key Formatter uses to look up the funcObject.

NNFunctionKeyPairCollection Private Data Member

fkColl

Overview

This is a pointer to the internal implementation of this class.

Syntax

```
NNFuncKeyColl* fkColl
```

Parameters

N/A

User-defined Type Input Field Validation

User-defined Type Input Field Validation is implemented through User Callback API functions, described in the previous section.

Formatter performs Input Field Validation of a user-defined type after an input message is completely parsed. User-defined types are specified using the Formatter GUI or Format Management APIs.

Note

Input Parse Controls and Output Format Controls can both be specified in terms of a user-defined type, but only input fields are subject to user define type validation. User-defined type validation is not performed on output fields.

Note

Validation callbacks are passed an array of NameValuePairs. The end of the array is marked by a NameValuePair with its two fields, name and value, set to NULL.

Note

If you need to allocate an array of NameValuePairs to pass to a MQSeries Integrator function (such as one of the Format Management APIs), be sure to allocate it with one more element than is needed to hold your data and call NameValuePair::MakeNull on the last pair to mark the end of the array.

Description

The underlying mechanisms which validation is based on are described in the User Callback API Functions section, and in the Formatter Member Functions and Format Management APIs sections. We list the relevant APIs and classes here to show usage; for complete general descriptions, refer to the appropriate sections.

Formatter Constructor

Overview

Use the constructor version that takes two arguments. The second argument is a collection of callback objects as described in the User Callback API Functions section.

If you have created callback objects appropriately, put them in a collection and passed the collection to the formatter constructor as above, validation will happen for input fields with parse controls defined in terms of a user-defined type.

The validation functions a user defines in his derived callback class return zero (0) for validation failure, and non-zero for validation success. If any validation callback returns failure, Formatter will fail the entire parse, just as it does now with its own internal validation.

Syntax

```
Formatter::Formatter(
        DbmsSession*,
        NNFunctionKeyPairCollection*);
```

Formatter Validation On/Off Functions

Overview

By default, validation is ON. To turn validation off or on, or to check the current validation state, use one of the following three Formatter member functions.

Syntax

```
void Formatter::SetUserTypeValidationOn();
void Formatter::SetUserTypeValidationOff();
int  Formatter::UserTypeValidationIsOn();
// return zero = off
```

Remarks

Note

You cannot turn validation on or off for individual fields. Validation is on or off for all fields in a message which are defined in terms of a user-defined type.

Note

THREAD SAFETY NOTE: If you are using the multi-threaded version of Formatter, these three functions apply to all threads globally. You cannot turn validation off for some threads and on for others.

User Callback API Functions

There are three User Callback abstract base classes available to implement a feature that uses callbacks. Validation uses only one of them: NNDBFieldsUserFunction.

Users must derive their own callback class from NNDBFieldsUserFunction and define all the pure virtual methods from that class.

Users then create as many objects of their derived class as needed to support validation of all user types they are concerned with. In many cases, just a single object of one derived class can be used. In other cases, it may be necessary for the user to derive a number of classes from NNDBFieldsUserFunction, and create a number of objects of each derived class. It makes no difference to Formatter, as long as all callback objects are of a user class derived from NNDBFieldsUserFunction.

Rough Sketch of Required Code

This is a sparse example of how to accomplish user-defined type input field validation. For a more complete example, see *Sample Program 1: msgtest.cpp* on page 307.

```
class myValidationClass : public NNDBFieldsUserFunction
{
    int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields )
    {
           // my implementation
           return validationResult;
    }
    int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
        NameValuePair* nameValuePairArray )
    {
           // my implementation
           return validationResult;
    }
    int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
        void* userRuntimeData )
    {
           // my implementation
           return validationResult;
    }
    int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
```

```
        NameValuePair* nameValuePairArray,
        void* userRuntimeData )
    {
            // my implementation
            return validationResult;
    }
};
...
char* userTypeOneKeyName = "key1";
char* userTypeTwoKeyName = "key2";

myValidationClass valOneCallbackObject;

myValidationClass valTwoCallbackObject;

NNFunctionKeyPairCollection
  myCollectionOfCallbackObjects;

myCollectionOfCallbackObjects.AddPair(
    &valOneCallbackObject,
    userTypeOneKeyName );
myCollectionOfCallbackObjects.AddPair(
    &valTwoCallbackObject,
    userTypeTwoKeyName );
...
Formatter myFmtr(
    dbSess,
    &myCollectionOfCallbackObjects );
myFmtr.SetUserTypeValidationOff();
if( ! myFmtr.UserTypeValidationIsOn() )
        myFmtr.SetUserTypeValidationOn();
...
myFmtr.Reformat();
...
```

Using the Thread Safe Release of the Formatter Engine

Use of the thread safe Formatter follows the same scheme described previously, with one difference. Each thread has its own set of input messages and resulting output messages. In other words, the Formatter class acts as a controller for formatting, treating each thread as a separate entity that processes its own input messages and retrieves its own resulting formatted messages. What this means is that each thread can process its own data concurrently with the other threads. This also means that one thread does not have access to the results of another thread's work. For example, one thread cannot call GetOutMsgGroup to obtain the output that another thread

has created by calling reformat. Thus, the formatting template defined previously changes slightly:

Instantiate an instance of the DbmsSession class to open a database session.

Instantiate an instance of the Formatter class, passing it the DbmsSession instance.

Create the defined number of threads, passing each thread the same Formatter instance.

For each thread

While there are input messages to format

For each input message to be formatted, call AddInputMessage to add the input message, along with the input format for the message.

For each output message, call AddOutputFormat to add the output format.

Call reformat on the Formatter instance.

[The formatter will format one output message for each output format that was specified by AddOutputFormat.]

For each format that was added via AddOutputFormat, call GetOutMsgGroup to get the resulting formatted message for the format.

end While

Clean up/terminate the thread.

end For

Thread Safety Feature Impacts to Formatter APIs

Only the Formatter class will be affected by Thread Safety. There is no impact on the OutMsg, OutMsgGroup, ParsedField, and ParsedMsg API.) Formatter class API functions impacted by Thread Safety are listed here.

Function	Description
AddInputMessage	All input messagesadded using AddInputMessage() are processed entirely within the thread from which they were added.
AddOutputFormat	The output formats added using AddOutputFormat() are usd to specify formatting rules for all input messages added within the same thread.

Function	Description
RemoveOutputFormat	RemoveOutputFormat() removes all output formats only for the thread from which it is called.
PreloadInFormat	PreloadInFormat() loads input formats only for the thread from which it is called. If PreloadInFormat() is called from the main thread, its results are not available to any other threads.
PreloadOutFormat	PreloadOutFormat() loads output formats only for the thread from which it is called. If PreloadOutFormat() is called from the main thread, its results are not available to any other threads.
parse	The parse() function parses only the input messages added using AddInputMessage() within the thread from which it is called. The parsed message is only available from within this thread.
reformat	The reformat() function formats only the input messages added using AddInputMessage() within the thread from which it is called. The formatted message is only available from within this thread.
GetFieldAscii / GetFieldString	GetFieldAscii() and GetFieldString() retrieve the results of a parse within the current string.
GetFieldAsciiByTag	GetFieldAsciiByTag() retrieves the results of a parse within the current string
GetOutMsgCount	GetOutMsgCount() returns the number of output message groups resulting from calling reformat() within the current thread.
GetOutMsgGroup	GetOutMsgGroup() returns the number of output message groups resulting from calling reformat() within the current thread.
GetOutMsgCount	GetOutMsgCount() returns the specified output message group resulting from calling reformat() within the current thread.
GetParsedInMsgCount	GetParsedInMsgCount() returns the number of input messages parsed by the Formatter within the current thread.
GetParsedInMsg	GetParsedInMsg() returns the specified parsed input message resulting from calling parse() within the current thread.
GetErrorCode	GetErrorCode() returns the error code of the last error that occurred within the current thread.

Function	Description
GetErrorMessage	GetErrorMessage() returns the error message of the last error that occurred within the current thread.

Data Cleanup

When a thread goes through the process of calling AddInputMessage(), AddOutputFormat(), reformat() or parse(), GetOutMsgGroup() or GetParsedInMsg(), and so on, internal buffers are stored for input messages, output messages, etc. After you have retrieved the results of formatting within a thread, you should terminate the thread and free its data from memory.

To free this data, use the mechanism that POSIX pthreads and UI threads provides with the thread-specific data interface.

Note

In an NT environment, this procedure will not perform data cleanup.

Linking

To link with the Thread Safe Formatter, the thread library corresponding to the release will need to be linked in. For example, to link with the POSIX pthreads version of Formatter, the pthreads library will need to be linked with the final executable.

Formatter Error Handling

GetErrorCode

Overview

GetErrorCode() returns the error code of any error that occurred with a Formatter object.

Note

THREAD SAFETY NOTE: GetErrorCode() returns the error code of the last error that occurred within the current thread. See the Using the Thread Safe Release of the Formatter Engine section for more information.

Syntax

```
int Formatter::GetErrorCode();
```

Parameters

None

Return Value

Returns the error code of any error that occurred with a Formatter object.

Example

See Sample Program 1: msgtest.cpp.

See Also

GetErrorMessage()

GetErrorMessage

Overview

GetErrorMessage() returns the error message text corresponding to the error code returned by GetErrorCode().

Note

THREAD SAFETY NOTE: GetErrorMessage() returns the error message of the last error that occurred within the current thread. See the How to Use the Thread Safe Release of the Formatter Engine section for more information.

Syntax

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

Parameters

None

Return Value

Returns the error message text corresponding to the error code returned by GetErrorCode().

Example

See Sample Program 1: msgtest.cpp.

See Also

GetErrorCode()

Formatter Error Messages

General Formatter Errors

Code	Error Name	Error Message	Error Explanation	Response to Error
1000	ERROR_ UNKNOWN_ FORMATTER_ ERROR	Unknown code or no error		
1001	ERROR_ MANDATORY_ OUTPUT_FIELD_ NOT_FOUND_IN_ INPUT	"andatory output field <field_name> not found in input	A field in an output format was defined as mandatory. There was no default supplied and no corresponding input field was found.	Correct the output format by specifying the field as optional, or supplying default if the field is mandatory.
1002	ERROR_INPUT_ FORMAT_NOT_ IN_DATABASE	Input format <format_name> not in database	An input format name was not found in the database.	Supply a correct format name.
1003	ERROR_ TRAILING_ CHARACTERS_ AFTER_ MESSAGE_ PARSE	<number_of_ characters> trailing characters <trailing_ characters> after message parse	The input format has been specified incorrectly with respect to the input message. The parser parsed all the fields specified, and still had unparsed data remaining in the input message.	Check that format was specified correctly.
1004	ERROR_OUTPUT_ FORMAT_NOT_ IN_DATABASE	Output format <format_name> not in database	An output format name was not found in the database.	Supply a correct format name.
1005	ERROR_ MANDATORY_ FIELD_HAS_ INVALID_DATA	Mandatory field <field_name> has invalid data <data>	An input field was encountered whose data did not comply with the specified data type.	Correct the data type for the field.
1006	ERROR_ UNPARSABLE_ DATA	Could not parse entire message into fields		Call NEON tech support.
1007	ERROR_TAG_ FIELD_PARSED_ WITH_ZERO_ LENGTH	Tag <field> parsed with no length		Add length to tag <field>
1008	ERROR_NOT_ ALL_FIELDS_ FOUND	Not all fields found. Missing field: <field_name>	In a random flat format, not all mandatory fields were found.	Check that format was specified correctly.

Code	Error Name	Error Message	Error Explanation	Response to Error
1009	ERROR_ZERO_ LENGTH_ OUTPUT_ MESSAGE	Output message has zero length		Call NEON tech support.
1010	ERROR_END_OF_ SIBLING_ SEQUENCE	End of sibling sequence		Call NEON tech support.
1011	ERROR_ CREATING_ RULES_ENGINE	Error creating rules engine	Internal Formatter error.	Call NEON tech support.
1012	ERROR_ EVALUATING_ RULES	Error evaluating rules	Internal Formatter error.	Call NEON tech support.
1013	ERROR_ READING_ OUTPUT_ CONTROL	Error reading output format control: <control_name> from database	A nonexistent output format control was specified for a field in an output format.	Check that format was specified correctly.
1014	ERROR_RULES_ ERROR	<error_text_ from_rules>	During conditional branching, the Rules component had an error evaluating rules.	Call NEON tech support.
1015	ERROR_RULES_ OPTION_NOT_ FOUND	Option not found after rules evaluation	Internal Formatter error.	Call NEON tech support.
1016	ERROR_RULES_ ACTION_NOT_ FOUND	Action ‘OutputFormat Control' not found after rules evaluation	Internal Formatter error.	Call NEON tech support.
1017	ERROR_RULES_ MULTIPLE_HITS	Multiple controls returned from rules for field: <field_name>. Control <control_name> and control <control_name>	The user specified conditional branching rules that conflicted with each other (more than one output format control was selected based on evaluation of rules).	Correct the field rules for conditional branching.
1018	ERROR_RULES_ INVALID_ FORMAT	Invalid output format name <format_name> returned from rules	A nonexistent output format was used in conditional branching field rules.	Check that format was specified correctly.
1019	ERROR_RULES_ INVALID_ CONTROL	Invalid format control name <control_name> returned from rules	A nonexistent output format control was used in conditional branching field rules.	Check that format was specified correctly.

Code	Error Name	Error Message	Error Explanation	Response to Error
1020	ERROR_NO_ MATHEXPR_TO_ PARSE	No mathematical expression to parse (NULL expression string)	The user picked a format control type of Math Expression in the GUI, without specifying a mathematical expression.	Specify a mathematical expression.
1021	ERROR_ MEPARSE_ FAILED	Mathematical expression parse failed: <expression>	The user used improper syntax when formulating a math expression for an output format control using the Formatter GUI tool.	Correct the expression syntax.
1022	ERROR_INVALID _PRECISION	Invalid mathematical expression precision value: <precision_value>	The user chose a negative value precision for a "Mathematical Expression" type of output format control.	Choose a precision that is non-negative.
1023	ERROR_EXIT_ FAILED	<name_of_exit_ routine>	A user written exit routine failed with an error.	User needs to debug the exit routine to determine why it failed. Use the 'msgtest' utility and a debugger to assist with this.
1024	ERROR_EXIT_ CLEANUP_ FAILED	<name_of_cleanup_ routine>	A user written exit cleanup routine failed with an error.	User needs to debug the exit cleanup routine to determine why it failed. Use the msgtest utility and a debugger to assist with this.
1025	ERROR_EXIT_ NAME_NOT_ FOUND	<name_of_exit_ routine>	The user specified an exit routine using the Formatter GUI tool, but did not link in the routine with the application calling the Formatter.	User should relink the application with the exit routine.
1026	ERROR_NULL_ DBMS_SESSION	Null DBMS session pointer	A zero-valued DbmsSession pointer was passed to the Formatter constructor.	Pass a valid pointer.

Code	Error Name	Error Message	Error Explanation	Response to Error
1027	ERROR_NULL_ INPUTMESSAGE_ PTR	Null input message buffer pointer	A zero-valued message buffer pointer was passed to the Formatter::AddInput Message function.	Pass a valid pointer.
1028	ERROR_NULL_ INPUTFORMAT_ NAME	Null input format name	A zero-valued pointer to an input format name was passed to a Formatter function.	Pass a valid pointer to a format name.
1029	ERROR_NULL_ OUTPUTFORMAT _NAME	Null output format name	A zero-valued pointer to an output format name was passed to a Formatter function.	Pass a valid pointer to a format name.
1030	ERROR_FIELD_ NAME_NOT_ FOUND	Field <field_name> not found	The user supplied a bad field name to Formatter::GetField Ascii().	Supply a correct field name.
1031	ERROR_FIELD_ DOMAIN_NOT_ FOUND	Field in domain <domain_id> not found	The user supplied a bad domain identifier to Formatter::GetField Ascii().	Supply a correct domain identifier.
1032	ERROR_CANNOT _LOAD_FORMAT	Unable to load format <format_name>	There was a problem loading a particular input or output format from the database, possibly due to the format name being incorrect.	Supply a correct format name.
1033	ERROR_ FIELDGROUP_ NOT_FOUND	“Cannot find field group”		Call NEON tech support.
1034	ERROR_ SEQUENCE_ BREAK_ OCCURRED	“Sequence break occurred”		Call NEON tech support.
1035	ERROR_INFINITE _LOOP	Infinite loop detected while parsing format: <format_name>	The input format has been specified incorrectly with respect to the input message. The parser has gotten stuck in the middle of the message and cannot advance the pointer into the input message buffer.	Check that format was specified correctly.

Code	Error Name	Error Message	Error Explanation	Response to Error
1036	ERROR_ NEGATIVE_ MESSAGE_ LENGTH	Negative message length is invalid: <length>	A negative message length was passed to the Formatter::AddInput Message function	Pass a positive message length to the function.
1037	ERROR_REPEAT_ COUNT_FIELD_ NOT_FOUND	Repeat count field not found prior to repeating component	A repeating component of a compound input format was specified as having a repeat count embedded in a preceding field value. This field was not encountered in the message.	Check that format was specified correctly.
1038	ERROR_ BOOLEAN_ CONTROL_ EVALUATED_ FALSE	Boolean control evaluated to FALSE	The boolean test associated with an output format control in the output format did not succeed.	None needed.
1039	ERROR_IBM_ FIELD_TOO_ WIDE	IBM Field specified too wide	The four IBM types, zoned and packed, signed and unsigned, are restricted to 16 bytes in width. The parse control was specified as larger than 16 bytes.	Use the API or GUI to correct the field's width.
1040	ERROR_IBM_ DECIMAL_ LOCATION_OUT_ OF_RANGE	IBM Field Decimal Location specified out of range	The parse control was specified with a decimal location "wider" than the field's width would allow. For a specified field width of N, IBM Zoned fields are restricted to a decimal location of zero to N; IBM Packed fields are restricted to a decimal location of (2*N - 1).	Use the API or GUI to correct the field's decimal location.
1041	ERROR_NO_ INPUT_ MESSAGES	Parse or Reformat called without supplying an input message"	AddInputFormat() was not used to add an input message prior to parsing or reformatting.	Use AddInputFormat() to add a message prior to using Parse() or Reformat().

Code	Error Name	Error Message	Error Explanation	Response to Error
1042	ERROR_NO_ OUTPUT_ FORMATS	Reformat called without specifying an output format	AddOutputFormat() was not used to add an output format prior to reformatting.	Use AddOutputFormat() to add an output format prior to using Reformat().
1044	ERROR_FATAL_ THREAD_ERROR	Fatal error encountered in thread library component	Fatal error encountered in thread library component.	Troubleshoot operating system to find error.
1045	ERROR_OUT_OF_ MEMORY	Memory allocation attempt failed	Memory allocation attempt failed.	Troubleshoot operating system to find error.
1046	ERROR_THREAD_ SAFE_INIT_ FAILED	Initialization of thread-safe mechanism failed	Initialization of thread-safe mechanism failed.	Troubleshoot operating system to find error.
1047	ERROR_USER_ DEF_TYPE_IN_ VAL_NO_VAL_ FUNC	User-defined type, input field validation, no validation function found, validation name (of validation routine)	Either the user type validation version of the Formatter constructor was not called or the collection supplied to the constructor did not contain a callback function object whose key matched the User-defined Type Validation Routine name for the current input field's user-defined type.	Call the correct Formatter constructor or add the required object / key pair to the collection before calling the constructor.
1048	ERROR_USER_ DEF_TYPE_IN_ VAL_PAIR_AND_ RUNTIME	User defined type, input field validation, name / value pair and runtime pointer callback returned failure, validation name (of validation routine)	The version of validation callback that takes a name / value pair array and pointer to run-time data returned failure.	If the validation callback is written correctly, fix the input message. If the input message is correct, fix the validation callback.
1049	ERROR_USER_ DEF_TYPE_IN_ VAL_PAIR	User defined type, input field validation, name / value pair callback returned failure, validation name (of validation routine)	The version of validation callback that takes a name / value pair array returned failure.	If the validation callback is written correctly, fix the input message. If the input message is correct, fix the validation callback.

Code	Error Name	Error Message	Error Explanation	Response to Error
1050	ERROR_USER_ DEF_TYPE_IN_ VAL_RUNTIME	User-defined type, input field validation, runtime pointer callback returned failure, validation name (of validation routine)	The version of validation callback that takes a pointer to run-time data returned failure.	If the validation callback is written correctly, fix the input message. If the input message is correct, fix the validation callback.
1051	ERROR_USER_ DEF_TYPE_IN_ VAL_NO_PARMS	User-defined type, input field validation, no-user-parameters callback returned failure, validation name (of validation routine)	The version of validation callback which takes no user parameters returned failure.	If the validation callback is written correctly, fix the input message. If the input message is correct, fix the validation callback.
1052	ERROR_ CORRUPT_ FORMATTER_DB	Corrupt formatter DB data detected		
1053	ERROR_OUTPUT_ CONVERSION_ FAILED	Conversion to output type failed for <type of error>	There was an error translating the tag, length, data or an error in conversion.	If error indicates tag, length, or data, the conversion was attempted and resulted in invalid datatype output, check the format datatype and the format definition to ensure they correspond. If the error indicates out data type, the conversion was not attempted.
1054	ERROR_REPEAT_ TERMINATION_ DELIMITER_NOT _FOUND	Repeat termination delimiter <delimiter value> not found for repeating format <format>	The data field has termination delimiter defined in the definition and the input message does not.	Check the input message to ensure that the data field has the termination delimiter defined in the input format definition, or change the input format definition to match the data. Run apitest –d to help isolate the field missing the termination delimiter.

Code	Error Name	Error Message	Error Explanation	Response to Error
1055	ERROR_INFINITE_LOOP_MAPPING	Infinite loop detected mapping fields for format <format> (possible cause: compound format using normal access mode controls)	Failed to terminate mapping portion defined by the access modes.	Check the access modes to ensure that they match the data.
1056	ERROR_MANDATORY_INPUT_FIELD_MISSING	Mandatory input field <field> not found	Field defined as mandatory and does not exist on input	Check input message data field (use apitest –d to help isolate the specific field) or the input format definition.
1057	ERROR_OUTPUT_FORMAT_NOT_FOUND	Output format <format> not found	Attempt to remove an output format that does not exist.	No impact.
1058	ERROR_NO_CONTROLLING_FIELD	Access mode specified for field <field> in format <format> requires controlling field	No longer used.	N/A
1059	ERROR_ZERO_LENGTH_SUBSTRING_INVALID	Zero length substring for output field <field> invalid	No longer used.	N/A
1060	ERROR_FIELD_HAS_INVALID_DATA	<field> field (data type <datatype>) has invalid data <data>	The output data is invalid for the datatype.	Check the output field datatype against the data coming from the associated input data.
1061	ERROR_OUTPUT_FIELD_WITH_INFIELD_HAS_INVALID_DATA	Output field <field> (data type <datatype>) mapped to input field <field> at byte offset <byte offset> has invalid data <data>	There is a corresponding input field with data but the output resulting from this input field is invalid for it's datatype.	Check the datatype of the output field and check the parsed input field data.
1062	ERROR_UNKNOWN_MACHINE_TYPEINT4	Failed to determine correct machine type endian of type Int4	Processing unknown Endian type.	Call NEON technical support.
1063	ERROR_CONVERSION_FAILED	Conversion of (data type <datatype>) failed for <value>, of length <length>	Input field data does not match input field definition.	Check input field data against input control to ensure they are compatible in definition.
1064	ERROR_DATA_TYPE_INVALID	Data Type (<datatype>) invalid	Unable to create datatype conversion.	Call NEON technical support.

Code	Error Name	Error Message	Error Explanation	Response to Error
1065	ERROR_ OVERFLOW	Data Overflow for data <data value> (datatype <datatype>)	Exceeded overflow for datatype.	Check format definition, input field data and corresponding output field format definition and change accordingly.
1066	ERROR_ MANDATORY_ FORMAT_ CONTAINS_NO_ FIELDS	Output Mandatory format <format> contains no fields	Compound output format is mandatory with no fields found.	Check the output format definition and / or the input data fields used in the output format.
1067	ERROR_BINARY_ ODD_LENGTH	Data (<data>) has odd length (<length>) or cannot be converted to Binary	Binary data must be of an even length.	Check input data to ensure the length is even.
1068	ERROR_ CONVERSION_ Y2K_ISO_FAILED	Y2K-Conversion of (data type <datatype>, base data type <base datatype>) to Internal ISO AsciiString representation failed for <data>, Format Attr Id <format>, Century <century>, Year C ut-off <year cutoff>	The problem is that the year is empty or the year string is greater than 2 digits or the year is negative or year cutoff is not between 0 and 100 inclusive.	Check the input message field data and input format definition.
1069	DEBUG_ERROR_ BAD_OUTPUT_ STREAM	[Parse Debugger] Bad output stream	The connection to the open file stream was lost.	
1070	DEBUG_ERROR_ DEBUG_MODE_ ALREADY_ON	[Parse Debugger] Debug mode is already on	N / A	No longer used.
1071	DEBUG_ERROR_ FAILED_TO_ ASSIGN_DEBUG_ OBJECT	[Parse Debugger] Failed to assign debug object	Thread specific data not stored correctly.	Troubleshoot threading.
1072	DEBUG_ERROR_ INVALID_DEBUG _CATEGORY	[Parse Debugger] Invalid debug category	N / A	No longer used.
1073	DEBUG_ERROR_ INVALID_ VERBOSE_LEVEL	[Parse Debugger] Invalid verbose level	N / A	No longer used.
1074	ERROR_SUBCTRL _DATABASE_ LOAD	Error loading <subcontrol>	Operation load failure.	Check the operation specified in the error message for correctness.

Code	Error Name	Error Message	Error Explanation	Response to Error
1075	ERROR_SUBCTRL _BEHAVIOR_ APPLY	Sub-control Error: <subcontrol>	Operations were not successfully applied.	Check the operation definitions and the datatype converted input field data.
1076	ERROR_OUTPUT_ CTRL_DATATYPE	Cannot create datatype <datatype> in output field control <output field control>	Invalid datatype specified.	Check the output control for datatype correctness.

Parsing Errors

Code	Error Name	Error Message	Error Explanation	Response to Error
1000	ERROR_ UNKNOWN_ FORMATTER_ ERROR	Unknown code or no error		Call NEON tech support.
-1	PARSE_ERROR_ EXACT_LENGTH_ TOO_LARGE	Exact length <length> is too large for input field <field_name>	User specified an exact length for a field in a parse control. The actual field data was not of the specified length.	Correct the length in the parse control.
-2	PARSE_ERROR_ LENGTH_IN_ MESSAGE_TOO_ LARGE	Message length <length> is too large	The user specified a length component of a field in a parse control. The length does not match the actual length of the data.	Correct the length in the parse control.
-3	PARSE_ERROR_ MINIMUM_ LENGTH_TOO_ LARGE	Minimum length <length> is too large for input field <field_name>	The user specified a parse control where the data termination was "Minimum Length + Delimiter" or "Minimum Length + White Space". The minimum length specified was too large.	Correct the length in the parse control.
-4	PARSE_ERROR_ DELIMITER_NOT _FOUND	Delimiter <delimiter_value> not found for input field <field_name>	The user specified a delimited field in a parse control. The delimiter was not encountered in the message.	Check that format was specified correctly.

Code	Error Name	Error Message	Error Explanation	Response to Error
-5	PARSE_ERROR_ WHITE_SPACE_ NOT_FOUND	White space not found for input field <field_name>	The user specified a white space delimited field in a parse control. White space was not encountered in the message.	Check that format was specified correctly.
-6	PARSE_ERROR_ LITERAL_NOT_ FOUND	Literal <literal_value> not found for input field <field_name>	The user specified a parse control of type "Literal". The literal value was not encountered in the message.	Check that format was specified correctly.
-7	PARSE_ERROR_ LITERAL_ DELIMITER_NOT _FOUND	Literal delimiter <delimiter_value> not found for input field <field_name>	The user specified a parse control of type Literal, and that the literal is terminated by a delimiter. The delimiter was not encountered in the message.	Check that format was specified correctly.
-8	PARSE_ERROR_ REGEXP_NOT_ FOUND	No match for regular expression <value> for input< field>	Regular cxpression not found.	Check to regular expression in the input control.
-9	PARSE_ERROR_ REGEXP_ DELIMITER_NOT _FOUND	No match for <type of regular expression> for <type of input field>		
-10		Regular expression for <type of delimiter> not found for <type of input field>		

Chapter 4
Format Management APIs

When adding formats, users must define format components in the following order:

1. fields
2. literals
3. user defined data types
4. parse (input) controls
5. output operations
6. operation collections
7. output master operations
8. output format controls
9. input flat formats
10. output master formats
11. input compound formats
12. output compound formats

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

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

General Format Management APIs

NNFMgrInit

Overview

NNFMgrInit() allocates and returns a pointer to an instance of NNFMgr tied to the DBMS specified by session.

Syntax

```
NNFMgr * NNFMgrInit(DbmsSession *session);
```

Parameters

Name	Type	Input/ Output	Description
session	DbmsSession*	Input	Name of the currently open database session. See OpenDbmsSession().

Return Value

Returns non-zero if the instance of NNFMgr is created successfully; zero (0) on failure.

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

See Also

NNFMgrClose()

NNFMgrClose

Overview

NNFMgrClose() frees resources associated with a session previously returned by NNFMgrInit(). NNFMgrClose() removes the user's ability to perform format management.

Note

NNFMgrClose() only cleans up resources claimed by NNFMgrInit() and does not close the DBMS session.

Syntax

```
void NNFMgrClose(NNFMgr *pNNFMgr);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid FMgr session previously returned by NNFMgrInit().

Remarks

NNFMgrClose() should be the last call made when all format management has been completed.

WARNING!

No other format management calls should be made after NNFMgrClose() has been called, unless a new Format Manager session is created by NNFMgrInit().

See Also

NNFMgrInit()

NNF_CLEAR

Overview

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

Syntax

```
NNF_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any format management structure	Input	Any structure used during format management (see structure descriptions for details).

Return Value

N/A

Example

```
struct NNFMgrFormatInfo f_info;

NNF_CLEAR(&app);
```

Field Management APIs

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

Guide for information on using NNFie and the ***MQSeries Integrator User Guide*** for information on using the Formatter GUI.

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

Field Management API Structures

NNFMgrFieldInfo

Overview

NNFMgrFieldInfo is a structure containing field information.

Syntax

```
typedef struct NNFMgrFieldInfo {
    unsigned char fieldName[33];
    unsigned char fieldDescription[129];

    long initFlag;
};
```

Parameters

Name	Type	Description
fieldName[33]	unsigned char	Name of field to add. Null terminated string of length 1 to 32 inclusive.
fieldDescription[129]	unsigned char	Comment about this field. Null terminated string of length zero (0) to 128 inclusive.
initFlag	long	Uninitialized structure check value.

Field Management APIs

NNFMgrCreateField

Overview

NNFMgrCreateField() adds a field to the database.

Syntax

```
const short NNFMgrCreateField (
              NNFMgr * pNNFMgr,
              const NNFMgrFieldInfo * const pFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid FMgr session previously returned by NNFMgrInit().
pFieldInfo	const NNFMgrField Info * const	Input	Information about the field to add (see Field Management API Structures for information about NNFMgrFieldInfo).

Remarks

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

Return Value

Returns non-zero if the field is created successfully; zero (0) on failure.

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

See Also

NNFMgrGetFirstField(), NNFMgrGetNextField()

NNFMgrGetFirstField

Overview

NNFMgrGetFirstField() retrieves field information for the first field from the database. To iterate through all the defined fields a call to NNFMgrGetFirstField() must be followed by calls to NNFMgrGetNextField() with the same NNFMgr session handle until NNFMgrGetNextField() returns an error.

Syntax

```
const short NNFMgrGetFirstField(
                NNFMgr *pNNFMgr,
                NNFMgr * const pFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFieldInfo	NNFMgr * const	Input	Information about the field (See Field Management API Structures for information about NNFMgrFieldInfo).

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateField(), NNFMgrGetNextField()

NNFMgrGetNextField

Overview

NNFMgrGetNextField() retrieves field information for all but the first field from the database. To iterate through all the defined fields a call to NNFMgrGetFirstField() must be followed by calls to NNFMgrGetNextField() with the same NNFMgr session handle until NNFMgrGetNextField() returns an error.

Syntax

```
const short NNFMgrGetNextField(
                NNFMgr *pNNFMgr,
                NNFMgr * const pFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFieldInfo	NNFMgr * const	Input	Information about the (See Field Management API Structures for information about NNFMgrFieldInfo).

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateField(), NNFMgrGetFirstField()

Literal Management APIs

Literal Management API Structures

NNFMgrLiteralInfo

Overview

NNFMgrLiteralInfo is a structure containing information regarding literals.

Syntax

```
typedef struct NNFMgrLiteralInfo {
        unsigned char literalName[33];
        unsigned char literalValue[127];
        unsigned short literalLength;

        long initFlag;
};
```

Parameters

Name	Type	Description
literalName[33]	unsigned char	Name of literal to create. NULL-terminated string length 1 to 32 inclusive.
literalValue[127]	unsigned char	Binary literal value; not necessarily NULL-terminated string.
literalLength	unsigned short	Length in bytes of literalValue. Valid range is 1 to 127 inclusive.
initFlag	unsigned short	Uninitialized structure check value.

Literal Management APIs

NNFMgrCreateLiteral

Overview

Creates a new Literal using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreateLiteral(
      NNFMgr* pNNFMgr,
      NNFMgrLiteralInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr LiteralInfo* const	Input / Output	Pointer to structure that provides data about NNFMgrCreateLiteral. See API structure section for information about NNFMgrLiteralInfo* const.

Return Value

Return a non-zero integer value on success, and zero (0) on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetLiteral

NNFMgrGetLiteral

Overview

Gets a single Literal from the database.

Syntax

```
const short NNFMgrGetLiteral(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrLiteralInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr LiteralInfo* const	Input / Output	Pointer to structure that contains data about NNFMgrGetLiteral. See API structure section for information about NNFMgrLiteralInfo* const.

Return Value

Return a non-zero integer value on success, and zero (0) on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateLiteral

User-Defined Data Type Management APIs

Note

User-defined data types may ONLY be assigned to the data_type portion of parse and format controls. They may NOT be used for the length_type or tag_type portions of parse or format controls.

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

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

User-Defined Data Type Management API Structures

NNFMgrUserDefTypeInfo

Overview

NNFMgrUserDefTypeInfo is a structure containing user-defined type information.

Syntax

```
typedef struct NNFMgrUserDefTypeInfo {
    char userDefTypeName[33];
    char nativeTypeName[33];
    char validationRoutineName[33];

    long initFlag;
} NNFMgrUserDefTypeInfo;
```

Parameters

Name	Type	Description
userDefTypeName[33]	char	Name of the user-defined type being defined.
nativeTypeName[33]	char	Name of the native type the user-defined type is being based on.
validationRoutineName[33]	char	Name (key) of callback function object to be used for most field validation.
initFlag	unsigned short	Uninitialized structure check value.

NNFMgrNameValuePairInfo

Overview

NNFMgrNameValuePairInfo associates an array of name/value pairs with an object (parse control) name and a usage type name (user-defined type input field validation).

Syntax

```
typedef struct NNFMgrNameValuePairInfo {
    char objectName[33];
    char pairType[33];
    NameValuePair* pairs;

    long initFlag;
} NNFMgrNameValuePairInfo;
```

Parameters

Name	Type	Description
objectName[33]	char	Name of object associated with this structure's name/value pair array.
pairType[33]	char	Type the name/value pair array used by this structure. For example, user-defined type input field validation is type IPC_DATA_VAL.
pairs	NameValuePair*	Array of name/value pairs.
initFlag	unsigned short	Uninitialized structure check value.

User-Defined Data Type Management APIs

NNFMgrCreateUserDefinedType

Overview

NNFMgrCreateUserDefinedType() adds a new user-defined type to the database.

Syntax

```
const short NNFMgrCreateUserDefinedType (
         NNFMgr * pNNFMgr,
         const NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid FMgr session previously returned by NNFMgrInit().
pTypeInfo	const NNFMgrUserDef TypeInfo * const	Input	Associates a user-defined type name with a native type name and a validation routine name.

Remarks

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

Return Value

Returns non-zero if the user-defined type is created successfully; zero (0) on failure.

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

See Also

NNFMgrGetUserDefinedType(), NNFMgrGetFirstUserDefinedType(), NNFMgrGetNextUserDefinedType()

NNFMgrAddNameValuePairs

Overview

NNFMgrAddNameValuePairs() adds a set of name / value pairs to an existing object such as a parse control.

Syntax

```
const short NNFMgrAddNameValuePairs (
       NNFMgr * pNNFMgr,
       const NNFMgrNameValuePairInfo * const pPairInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input / Output	Valid FMgr session previously returned by NNFMgrInit().
pPairInfo	const NNFMgrNameValue PairInfo * const	Input	Associates a name / value pair array with an object name and a usage type.

Remarks

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

Return Value

Returns non-zero if the name / value pair was added to the object named in pPairInfo; zero (0) on failure.

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

NNFMgrGetUserDefinedType

Overview

NNFMgrGetUserDefinedType() retrieves user-defined type information for the user-defined type named in pTypeName.

Syntax

```
const short NNFMgrGetUserDefinedType (
       NNFMgr * pNNFMgr,
       const char * const pTypeName,
       const NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input / Output	Valid FMgr session previously returned by NNFMgrInit().
pTypeName	const char * const	Input	Name of user-defined type to retrieve. NULL-terminated string length 1 to 32 (inclusive).
pTypeInfo	const NNFMgrUserDefType Info * const	Output	Information about the user-defined type (see *User-Defined Data Type Management APIs* on page 203).

Remarks

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

Return Value

Returns non-zero if the user-defined type information was retrieved successfully; zero (0) on failure.

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

See Also

NNFMgrCreateUserDefinedType(), NNFMgrGetFirstUserDefinedType(), NNFMgrGetNextUserDefinedType()

NNFMgrGetFirstUserDefinedType

Overview

NNFMgrGetFirstUserDefinedType() retrieves user-defined type information from the database. To iterate through all user-defined types, a call to NNFMgrGetFirstUserDefinedType() must be followed by calls to NNFMgrNextUserDefinedType() with the same NNFMgr session handle until NNFMGrGetNextUserDefinedType() returns an error.

Syntax

```
const short NNFMgrGetUserDefinedType (
       NNFMgr * pNNFMgr,
       const NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid FMgr session previously returned by NNFMgrInit().
pTypeInfo	const NNFMgrUserDefType Info * const	Output	Information about the user-defined type (See *User-Defined Data Type Management APIs* on page 203.).

Remarks

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

Return Value

Returns non-zero if the user-defined type information was retrieved successfully; zero (0) on failure.

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

See Also

NNFMgrCreateUserDefinedType(), NNFMgrGetUserDefinedType(), NNFMgrGetNextUserDefinedType()

NNFMgrGetNextUserDefinedType

Overview

NNFMgrGetNextUserDefinedType() retrieves user-defined type information from the database. To iterate through all user-defined types, a call to NNFMgrGetFirstUserDefinedType() must be followed by calls to NNFMgrNextUserDefinedType() with the same NNFMgr session handle until NNFMGrGetNextUserDefinedType() returns an error.

Syntax

```
const short NNFMgrGetNextUserDefinedType (
       NNFMgr * pNNFMgr,
       const NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input / Output	Valid FMgr session previously returned by NNFMgrInit().
pTypeInfo	const NNFMgrUserDef TypeInfo * const	Output	Information about the user-defined type. See *User-Defined Data Type Management APIs* on page 203.

Remarks

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

Return Value

Returns non-zero if the user-defined type information was retrieved successfully; zero (0) on failure.

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

See Also

NNFMgrCreateUserDefinedType(), NNFMgrGetUserDefinedType(),

NNFMgrGetFirstUserDefinedType()

Parse Control Management APIs

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

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

Parse Control Management API Structures

NNFMgrParseControlInfo

Overview

NNFMgrParseControlInfo is a structure containing parse control information.

Syntax

```
typedef struct NNFMgrParseControlInfo {
        char parseName[33];
        unsigned char optionalInd;
        short fieldType;
        short dataType;
        short dataTermination;
        char dataDelimiter[33];
        unsigned dataLength;
        short tagType
        short tagTermination;
        unsigned tagLength;
        unsigned char tagValue[33];
        char tagDelimiter[33];
        short lengthType;
        short lengthTermination;
        unsigned lengthLength;
        char lengthDelimiter[33];
        unsigned short decimalLocation;
        char validationParamName  [33];
        NameValuePair* userDefInValNameValuePairArray;
        long initFlag;
        // Date/Time data type fields
        char dataAttr[33];
        short baseDataType;
        short yearCutoff;
        short useZeroYearCutoffInd;
} NNFMgrParseControlInfo;
```

Parameters

Name	Type	Description
parseName[33]	unsigned char	Name of parse control. NULL-terminated string length 1 to 32 inclusive.
optionalInd	unsigned char	Zero (0) if the control is mandatory, non-zero if optional.
fieldType	short	How this parse control is to act. One of: INFIELD_PARSE_Data_Only INFIELD_PARSE_Tag_Data INFIELD_PARSE_Tag_Length_Data INFIELD_PARSE_Length_Data INFIELD_PARSE_Repetition_Count INFIELD_PARSE_Literal INFIELD_PARSE_Length_Tag_Data INFIELD_PARSE_Regexp
dataType	short	See *Data Type Descriptions* on page 333 for a list of supported data types for this release. Any user-defined data type codes defined using User-defined Type Input Field Validation may also be used.
dataTermination	short	One of: TERMINATION_Not_Applicable TERMINATION_Delimiter TERMINATION_Exact_Length TERMINATION_White_Space_Delimited TERMINATION_Minimum_Length_ Delimiter TERMINATION_Minimum_Length_ White_Space
dataDelimiter[33]	char	Name of delimiter if dataTermination is TERMINATION_Delimiter or TERMINATION_Minimum_Length_ Delimiter.
dataLength	unsigned	Length of data portion of field if dataTermination is TERMINATION_Exact_Length or TERMINATION_Minimum_Length_ Delimiter or TERMINATION_Minimum_Length_ White_Space.
tagType	short	Same possible values as dataType.
tagTermination	short	Same possible values as dataTermination.
tagValue[33]	char	Tag value, literal value, or regular expression.
tagDelimiter[33]	char	Name of delimiter of tagTermination is TERMINATION_Delimiter or TERMINATION_Minimum_Length_Deli miter.

Name	Type	Description
lengthType	short	Same possible values as dataType.
lengthTermination	short	Same possible values as dataTermination.
lengthLength	unsigned	Length of data portion of field if lengthTermination is TERMINATION_Exact_Length or TERMINATION_Minimum_Length_ Delimiter or TERMINATION_Minimum_Length_ White_Space.
lengthDelimiter[33]	char	Name of delimiter of lengthTermination is TERMINATION_Delimiter or TERMINATION_Minimum_Length_ Delimiter.
decimalLocation	int	Number indicating the decimal point location. Zero (0) indicates no decimal point change. Positive numbers indicate the number of digits in the input field considered to be right of the decimal point. Must not be longer than the maximum number of digits the field can hold. Negative numbers are not allowed.
validationParam Name[33]	char	String value that user-defined type input field validation code will pass to the callback runtime data lookup function.
userDefInValName ValuePairArray	NameVal uePair*	An array of name/value pairs to be associated with this parse control's user-defined type input field validation operations. The values in this array will be passed to a validation callback function.
initFlag	unsigned short	Uninitialized structure check value.
dataAttr	char	When using custom date/time formats, dataAttr should be set to the custom date/time format string you wish to use. This field is ignored when using standard date/time formats.
baseDataType	short	For all date/time formats, baseDataType is used to convert the raw data for the field into an internal date/time format. The baseDataType field can be Ascii_String, Ascii_Numeric, EBCDIC, or User Defined data types.

Name	Type	Description
yearCutoff	short	The yearCutoff field is used to indicate a cutoff year when dealing with 2 digit years on input. If a two digit year is >= year cutoff, it will be prefixed with 19. If a two digit year < year cutoff, it will be prefixed with 20. The valid range of values for yearCutoff is 0 to 100 inclusive.
useZeroYearCutoffInd	short	To specify a yearCutoff of 0, the user must specify yearCutoff = 0, AND specify useZeroYearCutoffInd = 1. The useZeroYearCutoffInd field indicates the user intentionally set yearCutoff to zero. By default, yearCutoff is always set to 0 in the call the NNF_CLEAR.

Note

The new Date/Time fields are only used when the user specifies a dataType of date, time, default date/time, or custom date/time. When using custom date/time formats, dataAttr should be set to the custom date/time format string you wish to use. For all date/time formats, baseDataType is used to convert the raw data for the field into an internal date/time format. The baseDataType field can be ASCII string, ASCII numeric, EBCDIC, or User Defined data types. The fieldType field must be Data Only, or Tag and Data. Data termination must be Exact Length, and length must match the length of the date/time format selected.

WARNING!

Year 2000 Compliance

The yearCutoff field is used to indicate a cutoff year when dealing with two-digit years on input. The following logic controls the century assigned to two-digit years.

Two-digit year >= year cutoff, prefix with 19
Two-digit year < year cutoff, prefix with 20

The valid range of values for yearCutoff is 0 to 100 inclusive. Using a yearCutoff of 100 forces all two-digit years to be prefixed with 20. Using a yearCutoff of 0 forces all two-digit years to be prefixed with 19. The user is required to specify a valid yearCutoff when a custom date/time format containing a two-digit year is selected via the dataAttr field.
To specify a yearCutoff of 0, the user must specify yearCutoff = 0, AND specify useZeroYearCutoffInd = 1. The useZeroYearCutoffInd field indicates

the user intentionally set yearCutoff to zero. By default, yearCutoff is always set to 0 in the call the NNF_CLEAR.

Note

All mandatory fields must parse correctly and have valid data (for the specific datatype). Optional fields do not have to parse successfully.

Parse Control Management APIs

NNFMgrCreateParseControl

Overview

NNFMgrCreateParseControl() adds a new parse control to the database.

Syntax

```
const short NNFMgrCreateParseControl(
       NNFMgr *pNNFMgr,
       const NNFMgrParseControlInfo *pParseControlInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid FMgr session previously returned by NNFMgrInit().
pParseControl Info	const NNFMgrParse ControlInfo *	Input	Information about the parse control to add. See Parse Control Management API Structures for information about NNFMgrParseControlInfo.

Remarks

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

If dataType in the NNFMgrParseControlInfo structure is set to:

DATA_TYPE_IBM_Packed_Integer,
DATA_TYPE_IBM_Signed_Packed_Integer,
DATA_TYPE_IBM_Zoned_Integer, or
DATA_TYPE_IBM_Signed_Zoned_Integer,

Delimiter information is ignored and NNFMgrCreateParseControl() will fail if dataTermination is not exact_length or dataLength is not between 1 and 16. It will also fail if decimalLocation is outside the range zero (0) to 16 for DATA_TYPE_IBM_Zoned_Integer, or
DATA_TYPE_IBM_Signed_Zoned_Integer, or zero (0) to 31 for
DATA_TYPE_IBM_Packed_Integer,
DATA_TYPE_IBM_Signed_Packed_Integer.

Return Value

Returns non-zero if the parse control is created successfully; zero (0) on failure.

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

See Also

NNFMgrGetFirstParseControl(), NNFMgrGetNextParseControl()

Note

The new Date/Time fields are only used when the user specifies a dataType of date, time, default date/time, or custom date/time. When using custom date/time formats, dataAttr should be set to the custom date/time format string you want to use. For all date/time formats, baseDataType is used to convert the raw data for the field into an internal date/time format. The baseDataType field can be ASCII string, ASCII numeric, EBCDIC, or User Defined data types. The fieldType field must be Data Only, or Tag and Data. Data termination must be Exact Length, and length must match the length of the date/time format selected.

WARNING!

Year 2000 Compliance

The yearCutoff field is used to indicate a cutoff year when dealing with two-digit years on input. The following logic controls the century assigned to two-digit years.

Two-digit year >= year cutoff, prefix with 19
Two-digit year < year cutoff, prefix with 20

The valid range of values for yearCutoff is 0 to 100 inclusive. Using a yearCutoff of 100 forces all two-digit years to be prefixed with 20. Using a yearCutoff of 0 forces all two-digit years to be prefixed with 19. The user is required to specify a valid yearCutoff when a custom date/time format containing a two-digit year is selected via the dataAttr field.
To specify a yearCutoff of 0, the user must specify yearCutoff = 0, AND specify useZeroYearCutoffInd = 1. The useZeroYearCutoffInd field indicates the user intentionally set yearCutoff to zero. By default, yearCutoff is always set to 0 in the call the NNF_CLEAR.

NNFMgrGetParseControl

Overview

NNFMgrGetParseControl() retrieves information about a parse control from the database.

Syntax

```
const short NNFMgrGetParseControl(
       NNFMgr * pNNFMgr,
       char * pParseName,
       NNFMgrParseControlInfo * const pParseControlInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseName	char *	Input	Name of parse control. Null terminated string length 1 to 32 (inclusive).
pParseControl Info	NNFMgrParse ControlInfo * const	Output	Information about the parse control. See Parse Control Management API Structures for information about NNFMgrParseControlInfo.

Return Value

Returns a non-zero integer value if the parse control information was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateParseControl(), NNFMgrGetFirstParseControl(), NNFMgrGetNextParseControl()

NNFMgrGetFirstParseControl

Overview

NNFMgrGetFirstParseControl() retrieves parse control information from the database. To iterate through all the defined parse controls a call to NNFMgrGetFirstParseControl() must be followed by calls to NNFMgrGetNextParseControl() with the same NNFMgr session handle until NNFMgrGetNextParseControl() returns an error.

Syntax

```
const short NNFMgrGetFirstParseControl(
       NNFMgr* pNNFMgr,
       NNFMgrParseControlInfo* const pParseControlInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseControl Info	NNFMgrParse ControlInfo * const	Output	Information about the parse control. See Parse Control Management API Structures for information about NNFMgrParseControlInfo.

Return Value

Returns a non-zero integer value if the parse control information was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateParseControl(), NNFMgrGetParseControl(), NNFMgrGetNextParseControl()

NNFMgrGetNextParseControl

Overview

NNFMgrGetNextParseControl() retrieves parse control information from the database. To iterate through all the defined parse controls a call to NNFMgrGetFirstParseControl() must be followed by calls to NNFMgrGetNextParseControl() with the same NNFMgr session handle until GetNextParseControl() returns an error.

Syntax

```
const short NNFMgrGetNextParseControl(
       NNFMgr * pNNFMgr,
       NNFMgrParseControlInfo * const pParseControlInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseControl Info	NNFMgrParse ControlInfo * const	Output	Information about the parse control. See Parse Control Management API Structures for information about NNFMgrParseControlInfo.

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateParseControl(), NNFMgrGetFirstParseControl(),

NNFMgrGetParseControl()

Output Format Control Management APIs

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

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

Output Format Control Management API Structures

NNFMgrOutMstrCntlInfo

Overview

NNFMgrOutMstrCntlInfo is a structure containing output control information.

Syntax

```
typedef struct NNFMgrOutMstrCntlInfo {
        char cntlName[NAME_LENGTH+1];
        short fieldType;
        // Indicates open or field mapsearch type
        short optionalInd;
        short dataType;
        // Output field data typechar
        dataAttr[NAME_LENGTH+1];
        // Format for special types(Example:date)
         short baseDataType;
        // Base data type for special types
        short tagType;
        // Data type of tag to output
        char tagLitrlName[NAME_LENGTH+1]
        unsigned char tagValue[LITRL_LENGTH+1];
        // Value of tag to output
        unsigned short tagValueLen;
        // Length of value in tagValue
        short tagBeforeLengthInd;
        // Should we output tag/len or len/tag
        short lengthType;
        // Data type of output length field (if any)
        short operationType;
        // Valid only for calculated field fieldType
        char fldLitrlName[NAME_LENGTH+1];
         unsigned char fldValue[LITRL_LENGTH+1];
         // Value for IF= Type, or Literal
         unsigned short fldValueLen;
         // Length of value in fldValue
         char childCntlName[NAME_LENGTH+1];
         // What control to use in formatting
         NNCntlType childCntlType;
         // The type of the control to use
         long initFlag;
} NNFMgrOutMstrCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
fieldType	short	The type of this output master control. Valid fieldTypes are: OUTFIELD_FORMAT_Data_Field_Name_ Search, OUTFIELD_FORMAT_Data_Field_Tag_ Search, OUTFIELD_FORMAT_Left_Operand_Field, OUTFIELD_FORMAT_Right_Operand_ Field, OUTFIELD_FORMAT_Calculated_Field, OUTFIELD_FORMAT_Conditional_Field, OUTFIELD_FORMAT_Existence_Check_ Field, OUTFIELD_FORMAT_Rules_Field, OUTFIELD_FORMAT_Input_Field_Exists, OUTFIELD_FORMAT_Input_Value_Equals OUTFIELD_FORMAT_Literal
optionalInd	short	Indicates whether this control is optional or required. Set optionalInd = 1 to indicate an optional field.
dataType	short	A valid MQSeries Integrator data type.
dataAttr	char[33]	Used only for dataType = DATA_TYPE_Custom_DateTime. For all other data types, this field is ignored. For custom date/time data types, this field should contain the format string for the custom date/time type. shortbaseDataTypeUsed only when dataType above is one of the date, time, date/time, or custom date/time types. For all other data types, baseDataType should be 0. baseDataType is used to determine the data type of the underlying field data used to represent the date/time information. Valid values are: DATA_TYPE_ASCII_String, DATA_TYPE_ASCII_Numeric, DATA_TYPE_EBCDIC_Data, or User Defined data types

Name	Type	Description
tagType	short	The data type of the tag portion of this field. Used only for field type of OUTFIELD_FORMAT_Data_Field_Tag_Search. Must be one of the valid MQSeries Integrator data types.
tagLitrlName	short	The name of the literal to use for output tag. Used only for field type of OUTFIELD_FORMAT_Data_Field_Tag_Search.
tagValue	unsigned char[128]	The literal value to use for output tag. Used only for field type of OUTFIELD_FORMAT_Data_Field_Tag_Search. If tagLitrlName is specified, this field is ignored.
tagBeforeLengthInd	short	Used only for field with both tag and length information. Indicates whether the tag is to be written to the output field before length. If tagBeforeLengthInd = 1, tag is written before length. Otherwise, length is written before tag.
lengthType	short	Used only for field with both tag and length information. Indicates the data type of the length portion of this field. Must be one of the valid MQSeries Integrator data types.
operationType	short	Used only for fieldType = OUTFIELD_FORMAT_Calculated_Field. Valid values are: OPERATION_Add, OPERATION_Subtract, OPERATION_Multiply, OPERATION_Divide
fldLitrlName	char[33]	Used only for fieldType = OUTFIELD_FORMAT_Input_Field_Equals. Used to specify the name of a literal to use as the comparison value.
fldValue	char [LITRL_ LENGTH]	Used only for fieldType = OUTFIELD_FORMAT_Input_Field_Equals. Used to specify the comparison value. If fldLitrlName is specified, this field is ignored.
childCntlName	char[33]	Name of the child control to associate with this master control. Must be specified and the control it names must exist in the database. To specify no child control used with this output master control, set childCntlName to “NONE”.

Name	Type	Description
childCntlType	NNCntl Type	Indicates the type of the child control named by childCntlName. The child control can be any valid control type. To specify no child control is to be used with this output master control, set childCntlType to NO_CNTL. The valid values for the ChildCntlType are as follows: NO_CNTL = 0 SUBSTITUTE_CNTL = 1 PRE_POST_FIX_CNTL= 2 DEFAULT_CNTL= 3 LENGTH_CNTL = 4 SUBSTRING_CNTL= 5 CASE_CNTL= 6 USER_EXIT_CNTL= 7 MATH_EXP_CNTL= 8 JUSTIFY_CNTL= 10 COLLECTION_CNTL= 11 TRIM_CNTL= 12
initFlag	short	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrSubstituteCntlInfo

Overview

Houses information used to create a new substitute control or gets an existing substitute control.

Syntax

```
typedef struct NNFMgrSubstituteCntlInfo {
        char cntlName[33];
        char inputLitrlName[33];
        unsigned char inputValue[128];
        unsigned short inputValueLen;
        char outputLitrlName[33];
        unsigned char outputValue[128];
        unsigned short outputValueLen;
        short outputValueType;
        // Data type of output value
        long initFlag;
} NNFMgrSubstituteCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
inputLitrlName	char[33]	Name of the input literal to use for the input value of this substitute.
inputLitrlName	char[33]	Name of the input literal to use for the input value of this substitute.
inputValue	char[128]	The value to use as the input value of this substitute. If inputLitrlName is specified, this field is ignored.
inputValueLen	unsigned short	The length of the valid data (in bytes) stored in inputValue.
outputLitrlName	char[33]	Name of the literal to use for the output value of this substitute.
outputValue	char[128]	The value to use as the output value of this substitute. If outputLitrlName is specified, this field is ignored.
outputValueLen	unsigned short	The length of the valid data (in byte) stored in outputValue.

Name	Type	Description
outputValueType	short	The data type of the output field data. Must be one of the valid MQSeries Integrator data types.
initFlag	short	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrUserExitCntlInfo

Overview

Houses information needed to create a new user exit control or gets an existing user exit control.

Syntax

```
typedef struct NNFMgrUserExitCntlInfo {
        char cntlName[33];
        char exitRoutine[33];
        long initFlag;
} NNFMgrUserExitCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
exitRoutine	char[33]	The name of the User Exit routine for this control.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrMathExpCntlInfo

Overview

Houses information needed to create a new math expression control or gets an existing math expression control. This structure deals only with the parent math expression control, and not the math expression segments for the control. Math expression segments are handled with the NNFMgrMathExpCntlSegmentInfo structure.

Syntax

```
typedef struct NNFMgrMathExpCntlInfo {
        char cntlName[33];
        unsigned short decimalPrecision;
        unsigned short roundingMode;
        long initFlag;
} NNFMgrMathExpCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
decimalPrecision	unsigned short	The decimal precision to carry with this math expression
roundingMode	unsigned short	1 = round up, 0 = round down.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrMathExpCntlSegmentInfo

Overview

Houses information used to create a new math expression segment or gets an existing math expression segment.

Syntax

```
typedef struct NNFMgrMathExpCntlSegmentInfo{
        char expression[256];
        long initFlag;
} NNFMgrMathExpCntlSegmentInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
expression	char[256]	The actual text for this math expression segment.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrPrePostFixCntlInfo

Overview

Houses information used to create a new pre / post fix control or gets an existing pre / post fix control.

Syntax

```
typedef struct NNFMgrPrePostFixCntlInfo {
        char cntlName[33];
        char litrlName[33];
        unsigned char value[128];
        unsigned short valueLen;
        NNFPrePostFix place;
        // PREFIX or POSTFIX
        short nullActionInd;
        // 0 or 1
        long initFlag;
} NNFMgrPrePostFixCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
litrlName	char[33]	The name of the literal which contains the data to be added to the output field is a prefix or posfix (suffix).
value	unsigned char[128]	The value data to be added to the output field is a prefix or posfix (suffix). If litrlName is specfied, this field is ignored.
valueLen	unsigned char	The length of the valid data in value (in bytes).
place	NNFPrePostFix	PREFIX or POSTFIX
nullActionInd	short	Flags this control to be used in the case of null input field data. Set this field to 1 to activate it for null action.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrDefaultCntlInfo

Overview

Houses information used to create a new default control or gets an existing default control.

Syntax

```
typedef struct NNFMgrDefaultCntlInfo {
        char cntlName[33];
        char litrlName[33];
        unsigned char value[128];
        unsigned short valueLen;
        long initFlag;
} NNFMgrDefaultCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
litrlName	char[33]	Name of the literal that contains the data for this default.
value	unsigned char[128]	The value for this default. If litrlName is specified, this field is ignored.
valueLen	unsigned short	The length of the valid data in value.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrLengthCntlInfo

Overview

Houses information to create a new length control or gets an existing length control.

Syntax

```
typedef struct NNFMgrLengthCntlInfo {
        char cntlName[33];
        char padLitrlName[33];
        unsigned char padValue[128];
        unsigned short padValueLen;
        unsigned long dataLen;
        long initFlag;
} NNFMgrLengthCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
padLitrlName	char[33]	The name of the literal that contains the data to be used to pad to the length specified in dataLen (if needed).
padValue	unsigned char[128]	The value to be used to pad to the length specified in dataLen (if needed). If padLitrlName is specified, this field is ignored.
padValueLen	unsigned short	The length of the valid data in padValue (in bytes).
dataLen	unsigned long	The length for this length control.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrSubStringCntlInfo

Overview

Houses information used to create a new substring control or gets an existing substring control.

Syntax

```
typedef struct NNFMgrSubStringCntlInfo {
        char cntlName[33];
        unsigned short start;
        unsigned short len;
        char padLitrlName[33];
        unsigned char padValue[128];
        unsigned short padValueLen;
        long initFlag;
} NNFMgrSubStringCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
start	unsigned short	The start position of the substring (1 = first char)
len	unsigned short	The number of characters to include in the substring.
padLitrlName	char[33]	The name of the literal that contains the data to be used to pad to the length specified in len (if needed).
padValue	unsigned char[128]	The value used to pad to the length specified in len (if needed). If padLitrlName is specified, this field is ignored.
padValueLen	unsigned short	The length of the valid data in padValue (in bytes).
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrCaseCntlInfo

Overview

Houses information used to get an existing case control. Users never create case controls.

Syntax

```
typedef struct NNFMgrCaseCntlInfo {
        char cntlName[33];
        NNFCase caseId;
        // LOWER_CASE or UPPER_CASE
        long initFlag;
} NNFMgrCaseCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
caseId	NNFCase	NNFCasecaseIdIndicates whether this case control is an upper or lower case control. Valid values are UPPER_CASE, LOWER_CASE.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrJustifyCntlInfo

Overview

Houses information used to get an existing justify control. Users never create justify controls.

Syntax

```
typedef struct NNFMgrJustifyCntlInfo {
        char cntlName[33];
        NNFJustify justify;
        // LEFT_JUSTIFY, RIGHT_JUSTIFY, or CENTER_JUSTIFY
        long initFlag;
} NNFMgrJustifyCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
justify	NNFJustify	Indicates what type of justification to do. Valid values are LEFT_JUSTIFY, RIGHT_JUSTIFY, CENTER_JUSTIFY
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrCollectionCntlInfo

Overview

Houses information used to create a new collection control or fetch an existing collection control. This structure deals only with the collection control itself, not its children. Child controls are added to or retrieved from collections with the NNFMgrCntlInfo structure.

Syntax

```
typedef struct NNFMgrCollectionCntlInfo {
        char cntlName[33];
        long initFlag;
} NNFMgrCollectionCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[33]	The name of the control. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. If you generate a default name, the cntlName field is populated with the default name upon return from the API call.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

Output Format Control Management APIs

NNFMgrCreateOutMstrCntl

Overview

Creates a new output master control, and associates it to a single child control (which may be a collection control containing any number of controls). The output master control is created using information given in the pInfo structure. The child control to associate with the new output master control is designated by the childCntlName and childCntlType members of the pInfo structure.

Syntax

```
const short CreateOutMstrCntl(
      NNFMgrOutMstrCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pInfo	NNFMgrOut MstrCntlInfo*	Input / Output	Pointer to structure that provides data about CreateOutMstrCnt. See API structure section for information about NNFMgrOutMstrCntInfo*.

Remarks

You can specify literal names to use for tag, and field value (used with Input Value Equals and Literal field types) by populating the tagLitrlName, and fldLitrlName fields, respectively, of the pInfo structure.

Alternatively, you can specify literal values to use for tag, and field value by populating the tagValue, and fldValue fields, respectively, of the pInfo structure. If you specify both values and names, names take precedence. If literal names are specified, the named literals must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success, and zero (0) on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetOutMstrCntl

NNFMgrGetOutMstrCntl

Overview

Gets a single output master control from the database. Only the child control name and type are returned in pInfo (not the actual child control data).

Syntax

```
const short GetOutMstrCntl(
      NNGetOp OpCode,
      NNFMgrOutMstrCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pInfo	NNGetOp OpCode	Input / Output	Pointer to structure that contains data about NNFMgrGetOutMstrCntl. See API structure section for information about NNGetOp OpCode.
pInfo	NNFMgrOut MstrCntlInfo*	Input / Output	Pointer to structure that contains data about NNFMgrGetOutMstrCntl. See API structure section for information about NNFMgrOutMstrCntInfo*.

Remarks

The location of the returned master control within this table is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Returns a non-zero integer value. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error.

See Also

CreateOutMstrCnt

Output Operations

Substitute Controls

Substitute controls can contain one or more substitute entries. The first substitute entry for a substitute control is created in the call NNFMgrCreateSubstituteCntl. Subsequent substitute entries may be appended to the existing substitute control by calling NNFMgrAppendEntryToSubstituteCntl and setting the cntlName member of the NNFMgrSubstituteCntlInfo structure to the same name as the existing substitute control.

NNFMgrCreateSubstituteCntl

Overview

Creates a new substitute control using the information in the pInfo structure. This call creates the first substitute entry for this substitute control. Additional substitute entries may be added to this control by calling NNFMgrAppendEntryToSubstituteCntl with the cntlName of the structure set to the name of this control.

Syntax

```
const short NNFMgrCreateSubstituteCntl(
      NNFMgr* pNNFMgr,
      NNFMgrSubstituteCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrSubstituteCntlInfo*	Input/ Output	Pointer to structure that provides data about NNFMgrCreateSubstituteCntl. See API structure section for information about NNFMgrSubstituteCntlInfo*.

Remarks

You can specify literal values to use for input and output by populating the inputValue, and outputValue fields of the pInfo structure.

Alternatively, you can specify literal names to use for input and output by populating the inputLitrlName, and outputLitrlName fields of the pInfo structure. If you specify both values and names, names take precedence. If literal names are specified, the named literals must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success and 0 on failure. 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

See *Code Example for Substitute Controls* on page 327.

See Also

NNFMgrAppendEntryToSubstituteControl, NNFMgrGetSubstituteCntl, NNFMgrGetNextEntryFromSubstituteCntl

NNFMgrAppendEntryTo SubstituteControl

Overview

Appends a substitute entry to an EXISTING substitute control named by the cntlName of the pInfo structure. NNFMgrCreateSubstitureCntl should have been called with the same cntlName before making this call. An error is returned if no control exists by this name.

Syntax

```
const short NNFMgrAppendEntryToSubstituteControl(
      NNFMgr* pNNFMgr,
      const NNFMgrSubstituteCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrSubstituteCntlInfo*	Input / Output	Pointer to structure that provides data about NNFMgrCreateSubstituteCntl. See API structure section for information about NNFMgrSubstituteCntlInfo*.

Return Value

Return a non-zero integer value on success and 0 on failure. 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

See *Code Example for Substitute Controls* on page 327.

See Also

NNFMgrCreateSubstituteCntl, NNFMgrGetSubstituteCntl, NNFMgrGetNextEntryFromSubstituteCntl

NNFMgrGetSubstituteCntl

Overview

Gets the first substitute entry from a single Substitute control in the database. The number of remaining entries in this control is returned in the NumRemainingEntries argument. You must use NNFMgrGetNextEntryFromSubstituteCntl to get the remaining (second, third, and so on) substitute entries for this control. The location of the returned control within the list of all controls of this type is determined by the OpCode argument.

Syntax

```
const short NNFMgrGetSubstituteCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrSubstituteCntlInfo* const pInfo,
      int* const NumRemainingEntries)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control within the list of all controls of this type is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr SubstituteCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSubstituteCntl. See API structure section for information about NNFMgrSubstituteCntlInfo* const.
NumRemaining Entries	int* const	Input/ Output	The number of remaining entries in this control is returned in the NumRemainingEntries argument.

Remarks

The number of remaining entries in this control is returned in the NumRemainingEntries argument. You must use

NNFMgrGetNextEntryFromSubstituteCntl to get the remaining (second, third, and so on) substitute entries for this control.

Return Value

Return a non-zero integer value on success and 0 on failure. 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

See *OpCode* on page 331.

See Also

NNFMgrCreateSubstituteCntl, NNFMgrAppendEntryToSubstituteControl, NNFMgrGetNextEntryFromSubstituteCntl

NNFMgrGetNextEntryFromSubstituteCntl

Overview

Gets the next entry from the substitute control named by pInfo->cntlName.

Syntax

```
const short NNFMgrGetNextEntryFromSubstituteCntl(
      NNFMgr* pNNFMgr,
      NNFMgrSubstituteCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrSubstituteCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetNextEntryFromSubstituteCntl. See API structure section for information about NNFMgrSubstituteCntlInfo* const.

Remarks

You must have called NNFMgrGetSubstituteCntl prior to calling this routine, and the cntlName field of pInfo must be the same as the value used in the NNFMgrGetSubstituteCntl call.

Return Value

Return a non-zero integer value on success and 0 on failure. 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

See *Code Example for Substitute Controls* on page 327.

See Also

NNFMgrCreateSubstituteCntl, NNFMgrAppendEntryToSubstituteControl, NNFMgrGetSubstituteCntl

User Exit Controls

NNFMgrCreateUserExitCntl

Overview

Creates a new User Exit control using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreateUserExitCntl(
      NNFMgr* pNNFMgr,
      NNFMgrUserExitCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrUser ExitCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateUserExitCntl. See API structure section for information about NNFMgrUserExitCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetUserExitCntl

NNFMgrGetUserExitCntl

Overview

Gets a single UserExit control from the NNF_YYY table.

Syntax

```
const short NNFMgrGetUserExitCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrUserExitCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control within this table is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgrUser ExitCntlInfo* const	Input / Output	Pointer to structure that contains data about NNFMgrGetUserExitCntl. See API structure section for information about NNFMgrUserExitCntlInfo*.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateUserExitCntl

Math Expression Controls

Math expression controls can contain any length of mathematical expression. This is possible because the actual data is stored in a set of ordered segments in a separate table. Thus math expression controls are a form of collection. However, users are only allowed to append segments to a math expression, and they are only allowed to access those segments sequentially from the first to the last segment. The parent math expression control is managed via the standard Create/Get APIs.

NNFMgrCreateMathExpCntl

Overview

Creates a new MathExp control using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreateMathExpCntl(
      NNFMgr* pNNFMgr,
      NNFMgrMathExpCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrMath ExpCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateMathExpCntl. See API structure section for information about NNFMgrMathExpCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetMathExpCntl, NNFMgrAppendSegToMathExpCntl, NNFMgrGetSegFromMathExpCntl

NNFMgrGetMathExpCntl

Overview

Gets a single MathExp control from the database.

Syntax

```
const short NNFMgrGetMathExpCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrMathExpCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input/ Output	The location of the returned control within the list of all math expressions is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgrMath ExpCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetMathExpCntl. See API structure section for information about NNFMgrMathExpCntlInfo*const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateMathExpCntl, NNFMgrAppendSegToMathExpCntl, NNFMgrGetSegFromMathExpCntl

NNFMgrAppendSegToMathExpCntl

Overview

Appends a single segment to the math expression control named by CntlName parameter, using the information given in the NNFMgrMathExpCntlSegmentInfo structure.

Syntax

```
const short NNFMgrAppendSegToMathExpCntl(
      NNFMgr* pNNFMgr,
      const char* const CntlName,
      const NNFMgrMathExpCntlSegmentInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	constNNFMgr MathExpCntl SegmentInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSegFromMathExpCntl. See API structure section for information about NNFMgrMathExpCntlSegmentInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateMathExpCntl, NNFMgrGetMathExpCntl, NNFMgrGetSegFromMathExpCntl

NNFMgrGetSegFromMathExpCntl

Overview

Gets a single segment from the math expression control named by CntlName.

Syntax

```
const short NNFMgrGetSegFromMathExpCntl(
      NNFMgr* pNNFMgr,
      const char* const CntlName,
      NNGetOp OpCode,
      NNFMgrMathExpCntlSegmentInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrMath ExpCntl SegmentInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSegFromMathExpCntl. See API structure section for information about NNFMgrMathExpCntlSegmentInfo* const.

Remarks

The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateMathExpCntl, NNFMgrGetMathExpCntl, NNFMgrAppendSegToMathExpCntl

Pre/PostFix Controls

PrePostFix controls are used to add used-definable information to the front (prefix) and/or back (postfix or suffix) of a field value. If the input data for an output field is null (field not present on input), you can add a prefix, postfix, or both to the output field data. To force a prefix when the field data is null, create a PrePostFix control with placeId = PREFIX, and nullActionInd = 1, add this control to a collection, and associate the collection with your output master control for this field. To force a postfix(suffix) when the field data is null, create a PrePostFix control with placeId = POSTFIX, and nullActionInd = 1, add this control to a collection, and associate the collection with your output master control for this field. If nullActionInd is 0, no action is taken for a PrePostFix control in the case of null input data. If the input data for a field is NOT null, the prefix or postfix described by a PrePostFix control is applied regardless of the value of nullActionInd.

You can have any number of PrePostFix controls in a collection. The controls are evaluated in the order they appear within the collection. In addition, any of these controls can have nullActionInd = 1. If the input data for a field is null, the controls that are flagged with nullActionInd = 1 are applied to the field data in the order they appear in the collection.

NNFMgrCreatePrePostFixCntl

Overview

Creates a new PrePostFix control using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreatePrePostFixCntl(
      NNFMgr* pNNFMgr,
      NNFMgrPrePostFixCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrPrePostFixCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreatePrePostFixCntl. See API structure section for information about NNFMgrPrePostFixCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetPrePostFixCntl

NNFMgrGetPrePostFixCntl

Overview

Gets a single PrePostFix control from the database.

Syntax

```
const short NNFMgrGetPrePostFixCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrPrePostFixCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgrPrePostFixCntlInfo * const	Input / Output	Pointer to structure that contains data about NNFMgrGetPrePostFixCntl. See API structure section for information about NNFMgrPrePostFixCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreatePrePostFixCntl

Default Controls

NNFMgrCreateDefaultCntl

Overview

Creates a new default control using the information in the pInfo structure.

Syntax

```
const short
NNFMgrCreateDefaultCntl(NNFMgr* pNNFMgr,
NNFMgrDefaultCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr DefaultCntl Info* const	Input / Output	Pointer to structure that provides data about NNFMgrCreateDefaultCntl. See API structure section for information about NNFMgrDefaultCntlInfo* const.

Remarks

You can specify a literal value to use for the default by populating the value field of the pInfo structure.

Alternatively, you can specify a literal name to use for the default by populating the litrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetDefaultCntl

NNFMgrGetDefaultCntl

Overview

Gets a single Default control from the database.

Syntax

```
const short NNFMgrGetDefaultCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrDefaultCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr DefaultCntl Info* const	Input / Output	Pointer to structure that provides data about NNFMgrGetDefaultCntl. See API structure section for information about NNFMgrDefaultCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateDefaultCntl

Length Controls

NNFMgrCreateLengthCntl

Overview

Creates a new Length control using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreateLengthCntl(
      NNFMgr* pNNFMgr,
      NNFMgrLengthCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrLengthCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateLengthCntl. See API structure section for information about NNFMgrLengthCntlInfo* const.

Remarks

You can specify a literal value to use for the pad character by populating the padValue field of the pInfo structure. Note that only the first byte of the literal is used for padding.

Alternatively, you can specify a literal name to use for the pad character by populating the padLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetLengthCntl

NNFMgrGetLengthCntl

Overview

Gets a single Length control from the database.

Syntax

```
const short NNFMgrGetLengthCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode
      NNFMgrLengthCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr LengthCntl Info* const	Input / Output	Pointer to structure that contains data about NNFMgrGetLengthCnt. See API structure section for information about NNFMgrLengthCntlInfo* const.

Remarks

The location of the returned control within the list of all Length controls is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateLengthCntl

SubString Controls

NNFMgrCreateSubStringCntl

Overview

The substitute string sub-control allows the user to replace an incoming field value that exactly matches a specified substitute input value with a substitute output string. The incoming field buffer and the substitute input string must match byte for byte. If the incoming field buffer and the substitute string do not exactly match the input field buffer is not changed and processing continues.

Syntax

```
const short NNFMgrCreateSubStringCntl(
      NNFMgr* pNNFMgr,
      NNFMgrSubStringCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr SubStringCntl Info* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateSubStringCntl. See API structure section for information about NNFMgrSubStringCntlInfo* const.

Remarks

You can specify a literal value to use for the pad character by populating the padValue field of the pInfo structure. Note that only the first byte of the literal is used for padding.

Alternatively, you can specify a literal name to use for the pad character by populating the padLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetSubStringCntl

NNFMgrGetSubStringCntl

Overview

Gets a single SubString control from the database.

Syntax

```
const short NNFMgrGetSubStringCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrSubStringCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgrSubStringCntlInfo* const	Input / Output	Pointer to structure that contains data about NNFMgrGetSubStringCntl. See API structure section for information about NNFMgrSubStringCntlInfo* const.

Remarks

The location of the returned control within the list of all SubString controls is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateSubStringCntl

Case Controls

NNFMgrGetCaseCntl

Overview

Gets a single Case control from the database.

Syntax

```
const short NNFMgrGetSubStringCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrCaseCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input / Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgrCase CntlInfo* const	Input / Output	Pointer to structure that contains data about NNFMgrGetCaseCntl. See API structure section for information about NNFMgrCaseCntlInfo* const.

Remarks

The location of the returned control within the list of all Case controls is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Justify Controls

NNFMgrGetJustifyCntl

Overview

Gets a single Justify control from the database.

Syntax

```
const short NNFMgrGetJustifyCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrJustifyCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr JustifyCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetJustifyCntl. See API structure section for information about NNFMgrJustifyCntlInfo* const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Trim Controls

NNFMgrTrimCntlInfo

Overview

NNFMgrTrimCntlInfo is a structure that houses iinformation needed to create a trim control.

Syntax

```
NNFMgrTrimCntlInfo {
      char cntlName[NAME_LENGTH+1];
      char trimCharLitrlName[NAME_LENGTH+1];
      unsigned char trimChar[LITRL_LENGTH+1];
      unsigned short trimCharLen;
      NNFTrim trim;
      long initFlag;
} NNFMgrTrimCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[NAME_LENGTH+1}	The name of the control. For create APIs, this field should contain the name you wish to give the new control, or a zero length string (“”) if you wish to have a default name generated for you. If you request a default name, the cntlName field is populated with the default name upon return from the API call.
trimCharLitrlName	char[NAME_LENGTH+1	The name of the Literal formatter component defining the pad character for the trim operation. If no name is entered (i.e. ""), a new literal component will be created based on the trimChar and trimCharLen.
trimChar	unsigned char [LITRL_LENGTH+1]	The value of the trim pad character; same value as the referenced literal.
trimCharLen	unsigned short	The length of the pad character literal. NOTE: Even though the pad literal value may be greater than 1 byte, only the first byte of the value is used.

Name	Type	Description
trim	NNtrim	This value is an enumerated type identifying which side of the field to trim: Left_TRIM, Right_TRIM, and BOTH_TRIM.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrCreateTrimCntl

Overview

Creates a new Trim control using the information in the pInfo structure.

Syntax

```
const short NNFMgrCreateTrimCntl(
      NNFMgr* pNNFMgr,
      NNFMgrTrimCntlInfo* pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr TrimCntlInfo*	Input	Pointer to structure that provides data about NNFMgrCreateTrimCntl. See API structure section for information about NNFMgrTrimCntlInfo.

Remarks

You can specify a literal value to use for the trim character by populating the trimChar field of the pInfo structure. Note that only the first byte of the literal is used to designate the trim character.

Alternatively, you can specify a literal name to use for the trim character by populating the trimCharLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetTrimCntl

NNFMgrGetTrimCntl

Overview

Gets a single Trim control from the database.

Syntax

```
const short NNFMgrGetTrimCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrTrimCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr TrimCntlInfo * const	Input/ Output	Pointer to structure that contains data about NNFMgrGetTrimCntl. See API structure section for information about NNFMgrTrimCntlInfo*const.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateTrimCntl

Collection Controls

Collection controls can contain zero or more individual controls or collections of controls. The parent collection control is created / gotten using the standard Create / Get APIs described below. The set of child controls is maintained with the AddCntlToCollection and GetCntlFromCollection APIs.

NNFMgrCntlInfo

Overview

Houses the name and type of an output control type.

Syntax

```
typedef struct Ntypedef struct NNFMgrCntlInfo {
        char cntlName[NAME_LENGTH+1];
        NNCntlType cntlType;
        long initFlag;
} NNFMgrCntlInfo;
```

Parameters

Name	Type	Description
cnt1Name	char	Name of control.
cnt1Type	NNcnt1Type	Enumerated type having the following values: NO_CNTL = 0 SUBSTITUTE_CNTL = 1 PRE_POST_FIX_CNTL= 2 DEFAULT_CNTL= 3 LENGTH_CNTL = 4 SUBSTRING_CNTL= 5 CASE_CNTL= 6 USER_EXIT_CNTL= 7 MATH_EXP_CNTL= 8 JUSTIFY_CNTL= 10 COLLECTION_CNTL= 11 TRIM_CNTL= 12
initflag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrCreateCollectionCntl

Overview

Creates a new Collection control using the information in the pInfo structure.

Syntax

```
NNFMgrCreateCollectionCntl(
     NNFMgr* pNNFMg,
     NNFMgrCollectionCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr Collection CntlInfo* const	Input/ Output	Pointer to structure that provides data aboutNNFMgrCreateCollectionCntl. See API structure section for information about NNFMgrCollectionCntlInfo* const.

Remarks

Collection controls are created as empty collections (no child controls). Use the NNFMgrAddCntlToCollection API to populate a collection control with child controls.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrGetCollectionCntl, NNFMgrAddCntlToCollection, NNFMgrGetCntlFromCollection

NNFMgrGetCollectionCntl

Overview

Gets a single Collection control (not its children) from the database.

Syntax

```
const short NNFMgrGetCollectionCntl(
      NNFMgr* pNNFMgr,
      NNGetOp OpCode,
      NNFMgrCollectionCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr Collection CntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetCollectionCntl. See API structure section for information about NNFMgrCollectionCntlInfo* const.

Remarks

You can retrieve child controls associated with this collection by using the NNFMgrGetCntlFromCollection API.

The location of the returned Collection control within the list of all Collection controls is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateCollectionCntl, NNFMgrAddCntlToCollection, NNFMgrGetCntlFromCollection

NNFMgrAddCntlToCollection

Overview

Adds an existing control of any type to the collection control named by the CollName parameter, using the name and type information given in the NNFMgrCntlInfo structure. The control is added at the position indicated by the SeqNum parameter.

Syntax

```
const short NNFMgrAddCntlToCollection(
      NNFMgr* pNNFMgr,
      const char* const CollName,
      int SeqNum,
      const NNFMgrCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
CollName	const char* const	Input/ Output	The name of the collection to which the control will be added.
pInfo	const NNFMgrCntl Info* const	Input/ Output	Describes the information for the control that will be added.

Remarks

If SeqNum is <= 0 or > number of items currently in the collection, SeqNum is calculated to append the control after the last item currently in the collection. Otherwise, the control is inserted before the item located at position SeqNum in the collection. The first item in the collection is at SeqNum = 1, the second is at SeqNum = 2, and so on.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateCollectionCntl, NNFMgrGetCollectionCntl, NNFMgrGetCntlFromCollection

NNFMgrGetCntlFromCollection

Overview

Gets a single control from the collection named by the CollName parameter.

Syntax

```
const short NNFMgrGetCntlFromCollection(
      NNFMgr* pNNFMgr,
      const char* const CollName,
      NNGetOp OpCode,
      NNFMgrCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
pInfo	NNFMgr CntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetCntlFromCollection. See API structure section for information about NNFMgrCntlInfo* const

Remarks

The location of the returned control within the collection is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success and 0 on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrCreateCollectionCntl, NNFMgrGetCollectionCntl, NNFMgrAddCntlToCollection

Literals

Output controls have been designed to reuse literal strings as much as possible. These strings were formerly used primarily for input/output field delimiters, and thus were referred to as delimiters. The new controls refer to these literal strings as literals. Starting with MQSeries Integrator, all delimiter APIs and structures are deprecated, and should not be used for new development. They will be supported for some time to ensure backward compatibility.

Users do not provide literal id numbers in the structures passed to the control creation APIs (of the form NNFMgrCreateXXXCntl). Instead, they either pass the literal value or the literal name to use. If the literal value is used, and this literal value does not already exist, a new literal will be created that has this value and will be given a default name. The form is NNDef_Literal_<Counter>, where Counter = 1 greater than the number of literals currently in the NNF_LITRL table.

When a user specifies a literal value, the Formatter management APIs check to see if a literal of this value already exists. If a matching literal can be found that has been given a default literal name, then this literal is reused, and no new literal is created. Only literals with default names (not specified directly by the user) will be considered for reuse.

Default Control Name API

MQSeries Integrator allows you to work with date and time fields directly in the Formatter and defines several date/time formats.

GetDefaultCntlName

Overview

Ease of use is one major focus of the MQSeries Integrator GUIs and APIs. Naming individual controls can be time-consuming. In some cases, the user may not care what name is given to an individual control, and it is unnecessary to force the user to decide on a name in these cases.

Syntax

```
const short GetDefaultCntlName(
      NNCntlType Type,
      char* CntlName)
```

Parameters

Type Parameter Value	Root Value Used in Default Name	Reuse a control if possible and default name is requested?
OUT_MASTER_CNTL	OutMstr	No
SUBSTITUTE_CNTL	Substitute	No
USER_EXIT_CNTL	UserExit	Yes
MATH_EXP_CNTL	MathExp	No
PRE_POST_FIX_CNTL	PrePostFix	Yes
DEFAULT_CNTL	Default	Yes
LENGTH_CNTL	Length	Yes
SUBSTRING_CNTL	SubString	Yes
TRIM_CNTL	Trim	Yes
COLLECTION_CNTL	Collection	No

Remarks

Default names can be generated in two ways:

1. All the NNFMgrXXXCntlInfo structures (data structures for creating/getting controls) contain a cntlName member that names the control. If the user sets this name to an empty string (“”), the corresponding NNFMgrCreateXXXCntl API function will detect the fact that no name has been given, and will automatically call GetDefaultCntlName(), storing the generated name in the cntlName field of structure. This is the simplest way to get a default control

name, and is the method NEON recommends. The user can retrieve the name given to the newly created control by simply looking at the cntlName field of the structure passed into the NNFMgrCreateXXXCntl API function.

2. The user may call GetDefaultCntlName() directly, and store the generated default control name in the cntlName member of the NNFMgrXXXCntlInfo structure. If this method is used, it is possible that another process could generate and use the same default name before the current process can use the default name in creating the new control. If this happens, a duplicate key error will occur. For this reason, it is recommended that method 1 be used to generate default control names. If the user still chooses to use method 2, care should be taken to handle duplicate key errors, or to lock the index within a transaction that encloses both the GetDefaultCntlName() and NNFMgrCreateXXXCntl() calls. This will lock out other transactions and prevent duplicate key errors.

Return Value

Return a non-zero integer value on success, and 0 on failure. 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

Assume the user has previously created 10 substring controls (regardless of what or how they are named). The following code fragments illustrate how to create a new substring control with a default name. In both examples, the generated default control name is “NNDef_SubString_11”:

Example 1

```
NNFMgrSubStringCntlInfo myInfo;
NNF_CLEAR(&myInfo);
strcpy(myInfo.cntlName, "");
// Request a default control name
 myInfo.start = 10; myInfo.len = 15;
strcpy(myInfo.padValue, "X");
 myInfo.padValueLen = 1;
 short ret = NNFMgrCreateSubStringCntl(&myInfo);
if (!ret){
   // error }
else printf("The default name generated is: %s",
myInfo.cntlName);
```

Example 2

```
NNFMgrSubStringCntlInfo myInfo;
NNF_CLEAR(&myInfo);
GetDefaultCntlName(SUBSTRING_CNTL, myInfo.cntlName);
// store default name in myInfo.cntlName
 myInfo.start = 10; myInfo.len = 15;
strcpy(myInfo.padValue, "X");
 myInfo.padValueLen = 1;
 short ret = NNFMgrCreateSubStringCntl(&myInfo);
```

Custom Date/Time Formats

NNFMgrGetDateTimeFormatString

Overview

Gets a single date/time format string from the database.

Syntax

```
const short NNFMgrGetDateTimeFormatString(
      NNFMgr *pNNFMgr,
      NNGetOp OpCode,
      short customFlag,
      char* const pFormatStr)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.
customFlag	short	input	Indicates whether the date/time format is a custom format. 1=true; 0=false. (Date/time format is one of the standard date/time formats.)
pFormatStr	char* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetDateTimeFormatString. See API structure section for information about char* const.

Remarks

The location of the returned format string within the list of all format strings is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 331.

Return Value

Return a non-zero integer value on success, and zero (0) on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

Recursion Check

When dealing with compound formats or collections of controls, it is conceivable to have a parent object that refers back to itself or one of its parent objects. This is referred to as recursion.

NNFMgrIsRecursiveFormat

Overview

Checks the format given by FormatName for recursion.

Syntax

```
const short NNFMgrIsRecursiveFormat(
      NNFMgr *pNNFMgr,
      const char* const FormatName,
      short * const IsRecursive)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
FormatName	const char* const	Input	The name of the format that will be checked for recursion.
IsRecursive	short * const	Output	Indicates whether the format was recursive. 1 indicates true; 0 indicates false.

Remarks

If the format is recursive, the IsRecursive argument will be set to 1; otherwise, IsRecursive is set to 0.

Return Value

Return a non-zero integer value on success, and zero (0) on failure. 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

When dealing with compound formats or collections of controls, it is conceivable to have a parent object that refers back to itself or one of its parent objects. This situation is referred to as recursion. An example of a recursive compound format follows:

1. Compound Format A
   - Compound Format B
   - Flat Format C
2. Compound Format B
   - Compound Format A
3. Flat Format C

In this example, Compound Format A consists of Compound Format B and Flat Format C. At this level, everything appears to be OK. However, Compound Format B consists of Compound Format A, which consists of Compound Format B, etc., etc. This situation causes an infinite loop when trying to traverse the children of Compound Format A. Thus, Compound Format A is said to be a “Recursive Format”.

See Also

NNFMgrIsRecursiveCollection

NNFMgrIsRecursiveCollection

Overview

Checks the collection given by FormatName for recursion.

Syntax

```
const short NNFMgrIsRecursiveCollection(
      NNFMgr *pNNFMgr,
      const char* const CollectionName,
      short * const IsRecursive)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
Collection Name	const char* const	Input/ Output	The name of the collection that will be checked for recursion.
IsRecursive	short * const	Input/ Output	Indicates whether the collection is recursive. 1 indicates true; 0 indicates false.

Remarks

If the collection is recursive, the IsRecursive argument will be set to 1;. otherwise, IsRecursive is set to 0. As with NNFMgrIsRecursiveFormat, if a child of a collection contains any one of its ancestors, the collection is recursive.

Return Value

Return a non-zero integer value on success, and zero (0)on failure. Use GetErrorNo() to retrieve the number for the error that occurred, then use GetErrorMessage() to retrieve the error message associated with that error number.

See Also

NNFMgrIsRecursiveFormat

Format Management APIs

WARNING!

If you are using a case-insensitive database, you cannot name components the same with only a change in case to identify them. For example, you cannot

name one field "f1" and another field "F1". In a case-insensitive environment, you must make each item unique using something other than case differences.

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

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

Format Management API Structures

NNFMgrFormatInfo

Overview

NNFMgrFormatInfo is a structure containing format information.

Syntax

```
typedef struct NNFMgrFormatInfo {
        unsigned char formatName[33];
        unsigned char inputInd;
        unsigned char compoundInd;

        long initFlag;
};
```

Parameters

Name	Type	Description
formatName[33]	unsigned char	Name of format. NULL-terminated string 1 to 32 characters long, inclusive.
inputInd	unsigned char	Set to zero (0) if the format is output type, 1 if input type.
compoundInd	unsigned char	If inputInd =1, can be 0, 1, 2, and 3. If inputInd=0, can be 0, 1, and 3 [Set to zero (0) if the format is flat, 1 if it is compound].
initFlag	long	Uninitialized structure check value.

NNFMgrRepeatFormatInfo

Overview

NNFMgrRepeatFormatInfo is a structure containing repeating format information.

Syntax

```
typedef struct NNFMgrRepeatFormatInfo{
        char childFormatName[33];
        int repeatTermination;
        char repeatDelimiter[33];
        unsigned repeatCount;
        unsigned char repeatFieldName[33];
        unsigned optionalInd;

        long initFlag;
};
```

Parameters

Name	Type	Description
childFormatName[33]	char	Name of child format. Must be a NULL-terminated string 1 to 32 characters long, inclusive.
repeatTermination	int	Termination of repetition. One of: TERMINATION_Not_Applicable TERMINATION_Delimiter TERMINATION_Exact_Length TERMINATION_White_Space_Delimited TERMINATION_Minimum_Length_ Delimiter TERMINATION_Minimum_Length_ White_Space
repeatDelimiter[33]	char	Name of repetition delimiter separator. Ignored unless repeatTermination is TERMINATION_Delimiter. NULL-terminated string length 1 to 32 inclusive.
repeatCount	unsigned	Number of times that format repeats.
repeatFieldName[33]	unsigned char	Name of repeating field. NULL-terminated string length 1 to 32 inclusive.
optionalInd	unsigned	Set to zero (0) for a mandatory component, and 1 for an optional component.
initFlag	long	Uninitialized structure check value.

NNFMgrFlatFormatInfo

Overview

NNFMgrFlatFormatInfo is a structure containing flat format information.

Syntax

```
typedef struct NNFMgrFlatFormatInfo{
        unsigned int decomposition;
        unsigned int length
        unsigned int termination;
        char Delimiter[33];

        long initFlag;
};
```

Parameters

Name	Type	Description
decomposition	unsigned int	Indicator of whether the format is ordered or random. Must be either IN_FORMAT_DECOMP_Ordered or IN_FORMAT_DECOMP_Unordered.
length	unsigned int	Length in bytes of format data.
termination	int	Termination of format. One of: TERMINATION_Not_Applicable TERMINATION_Delimiter TERMINATION_Exact_Length TERMINATION_White_Space_Delimited TERMINATION_Minimum_Length_Delimiter TERMINATION_Minimum_Length_White_ Space
Delimiter[33]	char	Name of format delimiter separator. Ignored unless Termination is TERMINATION_Delimiter. NULL-terminated string length 1 to 32 inclusive.
initFlag	long	Uninitialized structure check value.

NNFMgrInFieldInfo

Overview

NNFMgrInFieldInfo is a structure containing input field information.

Syntax

```
typedef struct NNFMgrInFieldInfo{
        char formatName[33];
        char fieldName[33];
        char controlName[33];

        long initFlag;
}
```

Parameters

Name	Type	Description
fieldName[33]	unsigned char	Name of field to add to format. Null terminated string of length 1 to 32 inclusive.
controlName[33]	unsigned char	Name of output format control associated with new field in format. Null-terminated string of length 1 to 32 inclusive.
formatName[33]	char	The format name to add the field to control mapping to.
initFlag	unsigned short	Uninitialized structure check value.

NNFMgrOutFieldInfo

Overview

NNFMgrOutFieldInfo is a structure containing output field information in an output format.

Syntax

```
typedef struct NNFMgrOutFieldInfo{
        char formatName[33];
        char fieldName[33];
        char controlName[33];
        short accessMode;
        short subscript;
        char inFieldName[33];

        long initFlag;
}
```

Parameters

Name	Type	Description
fieldName[33]	unsigned char	Name of field to add to format. Null-terminated string of length 1 to 32 inclusive.
controlName[33]	unsigned char	Name of output format control associated with new field in format. Null-terminated string of length 1 to 32 inclusive.
accessMode	short	One of: ACCESS_MODE_Not_Applicable ACCESS_MODE_Normal_Access ACCESS_MODE_Access_with_Increment ACCESS_MODE_Reset_then_Normal_Access ACCESS_MODE_Reset_then_Access_with_ Increment ACCESS_MODE_Access_nth_Instance_of_Field ACCESS_MODE_Access_within_Compound ACCESS_MODE_Cycling_Access_stay_in_ Compound ACCESS_MODE_Access_using_Relative_Index ACCESS_MODE_Create_Field It is considered an error to provide a subscript value when accessMode is not ACCESS_MODE_Access_nth_Instance_of_Field.
subscript	short	
initFlag	unsigned short	Uninitialized structure check value.

Format Management APIs

NNFMgrCreateFormat

Overview

NNFMgrCreateFormat() adds information about a new input or output, flat or compound format. NNFMgrCreateFormat() takes information passed in a pFormatInfo structure and creates a format named in the structure pointed to by pFormatInfo.

Note

Protocol ID and Protocol Version will NOT be supported by this API. Both will be defaulted to the value '1'.

Syntax

```
const short NNFMgrCreateFormat(
    NNFMgr * pNNFMgr,
    const NNFMgrFormatInfo * const pFormatInfo;
    const NNFMgrFlatFormatInfo * const  pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr session previously returned by NNFMgrInit().
pFormatInfo	const NNFMgrFormat Info * const	Input	Pointer to a valid NNFMgrFormatInfo structure. See Format Management API Structures for information about NNFMgrFormatInfo. This pointer may not be NULL.
pFlatFormat Info	const NNFMgrFlat FormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure (see Format Management API Structures for information about NNFMgrFormatInfo).

Remarks

A call to NNF_CLEAR for pFlatFormatInfo and pFormatInfo should be made prior to populating the structures or calling this API.

If you are not interested in the contents of the NNFMgrFlatFormatInfo structure, pass a zero (0) pointer as the third argument. Input flat formats will be created with decomposition, length, termination, and delimiter defaulted to zero (0) if no NNFMgrFlatFormatInfo is provided.

Return Value

Returns non-zero if the format is created successfully; zero (0) on failure.

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

See Also

NNFMgrGetFormat(), NNFMgrGetFirstFormat(), NNFMgrGetNextFormat()

NNFMgrAppendFieldToInputFormat

Overview

NNFMgrAppendFieldToInputFormat() adds a field to a flat input format.

formatName should be the name of an existing input flat format. fieldName should be the name of an existing field. NO VALIDITY CHECKING ON THESE PREREQUISITES WILL BE DONE IN THIS RELEASE.

Syntax

```
const short NNFMgrAppendFieldToInputFormat(
      NNFMgr *pNNFMgr,
      const char * const pFormatName,
      const NNFMgrInFieldInfo * const pInFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input / Output	Valid FMgr session previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of the parent format. NULL-terminated string length 1 to 32 inclusive.
pInFieldInfo	const NNFMgrInField Info * const	Input	Information about the field to add. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Remarks

A call to NNF_CLEAR for pInFieldInfo and pFormatName should be made prior to populating the structures or calling this API.

Return Value

Returns non-zero if the field is appended successfully; zero (0) on failure.

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

See Also

NNFMgrGetFirstFieldFromInputFormat(),
NNFMgrGetNextFieldFromInputFormat()

NNFMgrAppendFieldToOutputFormat

Overview

NNFMgrAppendFieldToOutputFormat() adds a field to a flat output format.

formatName should be the name of an existing output flat format. fieldName should be the name of an existing field. NO VALIDITY CHECKING ON THESE PREREQUISITES WILL BE DONE IN THIS RELEASE.

Syntax

```
const short NNFMgrAppendFieldToOutputFormat(
      NNFMgr * pNNFMgr,
      const char * const pFormatName,
      const NNFMgrOutFieldInfo * const pOutFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid FMgr session previously returned by NNFMgrInit().
pFormat Name	const char * const	Input	Name of flat output format to add field to. Must be a NULL-terminated string between 1 and 32 characters in length inclusive.
pOutField Info	const NNFMgrOutField Info * const	Input	Information about the field to add. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Remarks

A call to NNF_CLEAR for pOutFieldInfo and pFormatName should be made prior to populating the structures or calling this API.

Return Value

Returns non-zero if the field is added successfully; zero (0) on failure.

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

See Also

NNFMgrGetFirstFieldFromOutputFormat(), NNFMgrGetNextFieldFromOutputFormat()

NNFMgrAppendFormatToFormat

Overview

NNFMgrAppendFormatToFormat() adds a flat or compound format to a compound format. The child format will be added at the 'end' of all other child formats.

parentFormatName should be the name of an existing compound format. childFormatName should be the name of an existing compound or flat format. NO VALIDITY CHECKING ON PARENT AND CHILD FORMAT NAMES WILL BE DONE IN THIS RELEASE.

Syntax

```
const short NNFMgrAppendFormatToFormat(
  NNFMgr *pNNFMgr,
  const char * const pParentName,
  const NNFMgrRepeatFormatInfo * constpRepeatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid FMgr session previously returned by NNFMgrInit().
pParentName	const char * const	Input	Name of compound format. Must be a NULL-terminated string between 1 and 32 characters in length inclusive.
pRepeat FormatInfo	const NNFMgr Repeat FormatInfo* const	Input	Pointer to insertion information. This pointer may not be NULL. See Format Management API Structures for information about NNFMgrRepeatFormatInfo.

Remarks

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

Return Value

Returns non-zero if the flat or compound format is appended successfully; zero (0) on failure.

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

See Also

NNFMgrGetFormat(), NNFMgrGetFirstFormat(), NNFMgrGetNextFormat(), NNFMgrGetFirstChildFormat(), NNFMgrGetNextChildFormat()

NNFMgrGetFormat

Overview

NNFMgrGetFormat() reads information about an input or output, flat or compound format. To iterate through all formats in the database a call to NNFMgrGetFirstFormat() must be followed by calls to NNFMgrGetNextFormat() with the same session handle until NNFMgrGetNextFormat() returns an error.

Syntax

```
const short NNFMgrGetFormat(
    NNFMgr *pNNFMgr,
    const char * const pFormatName,
    NNFMgrFormatInfo * const pFormatInfo,
    const NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of format for which to retrieve information.
pFormatInfo	NNFMgrFormat Info * const	Output	Pointer to a valid NNFMgrFormatInfo structure (see Format Management API Structures for details). pFormatInfo may not be NULL. Structure fields will be filled by in-database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlat FormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure. See Format Management API Structures for information about NNFMgrFormatInfo.

Remarks

If you are not interested in the contents of the NNFMgrFlatFormatInfo structure, pass a zero (0) pointer as the fourth argument. Input flat formats will be created with decomposition, length, termination, and delimiter defaulted to zero (0) if no NNFMgrFlatFormatInfo is provided.

Return Value

Returns a non-zero integer value if the format was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateFormat(), NNFMgrGetFirstFormat(), NNFMgrGetNextFormat()

NNFMgrGetFirstFormat

Overview

NNFMgrGetFirstFormat() reads information about the first input or output, flat or compound format. To iterate through all formats in the database a call NNFMgrGetFirstFormat() must be followed by calls to NNFMgrGetNextFormat() with the same session handle until NNFMgrGetNextFormat() returns an error.

Syntax

```
const short NNFMgrGetFirstFormat(
    NNFMgr * pNNFMgr,
    NNFMgrFormatInfo * const pFormatInfo,
    const NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatInfo	NNFMgrFormat Info * const	Output	Pointer to a valid NNFMgrFormatInfo structure (see Format Management API Structures for details). pFormatInfo may not be NULL. Structure fields will be filled by in-database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlat FormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure. See Format Management API Structures for information about NNFMgrFormatInfo.

Remarks

If you are not interested in the contents of the NNFMgrFlatFormat structure, pass a zero (0) pointer as the second argument. Input flat formats will be created with decomposition, length, termination, and delimiter defaulted to zero (0) if no NNFMgrFlatFormatInfo is provided.

Return Value

Returns a non-zero integer value if the format was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateFormat(), NNFMgrGetFormat(), NNFMgrGetNextFormat()

NNFMgrGetNextFormat

Overview

NNFMgrGetNextFormat() reads information about all but the first input or output, flat or compound format. To iterate through all formats in the database a call NNFMgrGetFirstFormat() must be followed by calls to NNFMgrGetNextFormat() with the same session handle until NNFMgrGetNextFormat() returns an error.

Syntax

```
const short NNFMgrGetNextFormat(
     NNFMgr * pNNFMgr,
     NNFMgrFormatInfo * const pFormatInfo,
     const NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatInfo	NNFMgrFormat Info * const	Output	Pointer to a valid NNFMgrFormatInfo structure. See Format Management API Structures for details. pFormatInfo may not be NULL. Structure fields will be filled by in-database values if the call is successful.
pFlatFormat Info	const NNFMgrFlat FormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure. See Format Management API Structures for information about NNFMgrFormatInfo.

Remarks

If you are not interested in the contents of the NNFMgrFlatFormat structure, pass a zero (0) pointer as the second argument. Input flat formats will be created with decomposition, length, termination, and delimiter defaulted to zero (0) if no NNFMgrFlatFormatInfo is provided.

Return Value

Returns a non-zero integer value if the format was read successfully; zero (0) on failure.

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

See Also

NNFMgrCreateFormat(), NNFMgrGetFirstFormat(), NNFMgrGetFormat()

NNFMgrGetFirstFieldFromInputFormat

Overview

NNFMgrGetFirstFieldFromInputFormat() retrieves child field information for the first field of a flat input format. To iterate through all child fields in the format, a call to NNFMgrGetFirstFieldFromInputFormat() must be followed by calls to NNFMgrGetNextFieldFromInputFormat() with the same NNFMgr session handle until NNFMgrGetNextFieldFromInputFormat() returns an error.

Syntax

```
const short NNFMgrGetFirstFieldFromInputFormat(
      NNFMgr *pNNFMgr,
      const char * const pFormatName,
      NNFMgrInFieldInfo * const pInFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of the parent format. NULL-terminated string length 1 to 32 (inclusive).
pInFieldInfo	NNFMgrInField Info * const	Output	Information about the field. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrAppendFieldToInputFormat(),
NNFMgrGetNextFieldFromInputFormat()

NNFMgrGetNextFieldFromInputFormat

Overview

NNFMgrGetNextFieldFromInputFormat() retrieves field information for all but the first child field of a flat input format. To iterate through all child fields in the format, a call to NNFMgrGetFirstFieldFromInputFormat() must be followed by calls to NNFMgrGetNextFieldFromInputFormat() with the same NNFMgr session handle until NNFMgrGetNextFieldFromInputFormat() returns an error.

Syntax

```
const short NNFMgrGetNextFieldFromInputFormat(
      NNFMgr *pNNFMgr,
      NNFMgrInFieldInfo * const pInFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pInFieldInfo	NNFMgrInField Info * const	Output	Information about the field. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrAppendFieldToInputFormat(), NNFMgrGetFirstFieldFromInputFormat()

NNFMgrGetFirstFieldFromOutputFormat

Overview

NNFMgrGetFirstFieldFromOutputFormat() retrieves field information about the first field of a flat output format. To iterate through all child fields in the format a call to NNFMgrGetFirstFieldFromOutputFormat() must be followed by calls to NNFMgrGetNextFieldFromOutputFormat() with the same NNFMgr session handle until NNFMgrGetNextFieldFromOutputFormat() returns an error.

Syntax

```
const short NNFMgrGetFirstFieldFromOutputFormat(
      NNFMgr *pNNFMgr,
      const char * const pFormatName,
      NNFMgrOutFieldInfo * const pOutFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of flat output format to add field to. Must be a NULL-terminated string between 1 and 31 characters in length inclusive.
pOutFieldInfo	NNFMgr OutField Info * const	Output	Information about the field. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrAppendFieldToOutputFormat(), NNFMgrGetNextFieldFromOutputFormat()

NNFMgrGetNextFieldFromOutputFormat

Overview

NNFMgrGetNextFieldFromOutputFormat() retrieves field information for all but the first field of a flat output format. To iterate through all child fields in the format a call to NNFMgrGetFirstFieldFromOutputFormat() must be followed by calls to NNFMgrGetNextFieldFromOutputFormat() with the same NNFMgr session handle until NNFMgrGetNextFieldFromOutputFormat() returns an error.

Syntax

```
const short NNFMgrGetNextFieldFromOutputFormat(
      NNFMgr *pNNFMgr,
      NNFMgrOutFieldInfo * const pOutFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pOutFieldInfo	NNFMgr OutField Info * const	Output	Information about the field. See Format Management API Structures for information about NNFMgrOutFieldInfo.

Return Value

Returns a non-zero integer value if the field information was read successfully; zero (0) on failure.

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

See Also

NNFMgrAppendFieldToOutputFormat(), NNFMgrGetFirstFieldFromOutputFormat()

NNFMgrGetFirstChildFormat

Overview

NNFMgrGetFirstChildFormat() fetches details about the first child format of a compound input or output parent format. To iterate through all child formats in the parent a call to NNFMgrGetFirstChildFormat() must be followed by calls to NNFMgrGetNextChildFormat() with the same NNFMgr session handle until NNFMgrGetNextChildFormat() returns an error.

Syntax

```
const short NNFMgrGetFirstChildFormat(
      NNFMgr *pNNFMgr,
      const char * const pParentName,
      NNFMgrRepeatFormatInfo * const pRepeatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParentName	const char * const	Input	Name of compound format. Must be a NULL-terminated string between 1 and 32 characters in length inclusive.
pRepeatFormatInfo	NNFMgr Repeat FormatInfo * const	Output	Repetition information structure. See Format Management API Structures for information about NNFMgrRepeatFormatInfo.

Return Value

Returns a non-zero integer value if the child format was read successfully; zero (0) on failure.

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

See Also

NNFMgrGetNextChildFormat()

NNFMgrGetNextChildFormat

Overview

NNFMgrGetNextChildFormat() fetches details about all but the first child format of a compound input or output parent format. To iterate through all child formats in the parent a call to NNFMgrGetFirstChildFormat() must be followed by calls to NNFMgrGetNextChildFormat() with the same NNFMgr session handle until NNFMgrGetNextChildFormat() returns an error.

Syntax

```
const short NNFMgrGetNextChildFormat(
      NNFMgr *pNNFMgr,
      NNFMgrRepeatFormatInfo * const pRepeatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pRepeatFormat Info	NNFMgr RepeatFor matInfo * const	Output	Repetition information structure. See Format Management API Structures for information about NNFMgrRepeatFormatInfo.

Return Value

Returns a non-zero integer value if the child format was read successfully; zero (0) on failure.

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

See Also

NNFMgrGetFirstChildFormat()

Format Management API Error Handling

GetErrorNo

Overview

GetErrorNo() returns the error number for the last function call error.

Syntax

```
const int NNFMgr::GetErrorNo();
```

Parameters

none

Return Value

Returns the error number for the last function call error.

See Also

GetErrorMessage()

GetErrorMessage

Overview

GetErrorMessage() returns the error message describing the cause of the last function call error.

Syntax

```
const char * const NNFMgr::GetErrorMessage();
```

Parameters

none

Return Value

Returns the error message describing the cause of the last function call error.

See Also

GetErrorNo()

Format Management Error Messages

Error Name	Error Description	Explanation	Response to Error
NNF_NO_ERROR	No error was encountered.	No error was encountered	None needed.
NNF_INCOMPLETE_ ARGUMENTS	Arguments to function do not contain enough data.	Arguments passed to a function do not contain enough data.	Check data passed to function.
NNF_INCONSISTENT _ARGUMENTS	Arguments to function contain conflicting data.	Arguments passed to a function contain conflicting data.	Check data passed to function.
NNF_DB_ERROR	SQL query failed.	A database error was detected.	Call NEON tech support.
NNF_NO_DATA_ FOUND	Database returned no rows.	The data requested was not found.	None needed.
NNF_DB_ INCONSISTENT	Corrupt database condition detected.	A database error was detected.	Call NEON tech support.
NNF_NOT_ IMPLEMENTED	This function not implemented.	User is attempting to use functionality not present in this release.	Stop using this function.
NNF_INTERNAL_ ERROR	Internal consistency checking detected an internal error.	An internal error within Format Management code was detected.	Describe what happened and how it happened to NEON tech support.
NNF_DATA_INVALID	Data value supplied to the function was invalid.	Data value supplied to the function was invalid.	Check the data passed to the function.
NNF_DATATYPE_ INVALID	Data type supplied to the function was invalid.	Data type supplied to the function was invalid.	Check the data passed to the function.
NNF_OPERATION_ INVALID	Operation supplied to the function was invalid.	Operation supplied to the function was invalid.	Check the data passed to the function.

Appendix A

Sample Programs

The following programs provide examples of how Formatter APIs and User Callbacks are used.

Sample Program 1: msgtest.cpp

Using Formatter APIs to Reformat a Message

```
static const char RCS_ID[] = "$Header:
/source/aig/formatter/test/msgtest.cpp,v 1.9.4.3 1997/02/24 20:36:05 aaron
Exp $";

// This program can be used either with or without user defined type
// input field validation.  It calls an external function,
// GetValidationCallbacks(), to get a collection of validation callback
// objects.
//
// A do-nothing version of GetValidationCallbacks() is in getval.cpp. It just
// returns nil.  If you want to supply actual validation callbacks, just mv
// that source file to another name, and replace it with your own.
//
// One flexible way to do this is to create your own file that defines
// GetValidationCallbacks(), e.g. minimalgetval.cpp, and create a soft link
// to it, named getval.cpp.
//
// Once you've built the program, use it exactly the same as you did before.


// This program is a test driver for the formatter engine.  It asks the
// caller for an input file name, an output file name, an input format
// name and an output format name.  It treats the contents of the input
// file as a single message with the input format specified, and then
// reformats it and writes the resultint output message to the output
// file.
//
// The driver exercises the following APIs:
//
//   Formatter::AddInputMessage
//   Formatter::AddOutputFormat
//   Formatter::Reformat
//   Formatter::GetOutMsgGroup
//   OutMsgGroup::GetMsgCount
//   OutMsgGroup::GetMsg
//   OutMsg::GetMsgBuffer
//   OutMsg::GetMsgLength

// System include files

extern "C" {
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <stdlib.h>
#include <memory.h>
}
```

```
#include <iostream.h>
#include <fstream.h>

// Include files for database access

#include "interface.h"
#if defined(_MS_SQL_NT)
#include "sqlses.h"
#include "sqlapi.h"
#elif defined(sybase)
#include "sybfront.h"
#include "sybdb.h"
#endif

// Neonet include files

#include "dbtypes.h"
#include "ses.h"
#include "sqlapi.h"

#include "formatter.h"
#include "msgs.h"


// aaron: ser1079 2/97
NNFunctionKeyPairCollection* GetValidationCallbacks();

// Handles error returned by formatter (outputs a message), and returns
// error code.
static int
handleError(char * func, Formatter * pFormatter ) {
        int     code;

        if (code = pFormatter->GetErrorCode()) {
                cerr << "\nERROR during " << func << ": " << "(" << code <<
") " << pFormatter->GetErrorMessage() << "." << endl << endl;
        }
        return code;
}

int
main()
{

        // Open the database session.
#ifdef oracle
        DbmsSession *Session = OpenDbmsSession("new_format_demo",ORACLE7);
#else
        DbmsSession *Session = OpenDbmsSession("new_format_demo",SYBASE49);
#endif

        // Handle error if database cannot be opened.
        if ( !Session || !Session->Ok()){
                cerr << "No session created" << endl;
                cin.get();
                exit(errno);
        }

        NNFunctionKeyPairCollection* valCallbacks = GetValidationCallbacks();

        // Construct a formatter instance.
        Formatter * formatter;
        if( valCallbacks )
                formatter = new Formatter(Session, valCallbacks);
        else
                formatter = new Formatter(Session);

        if (handleError("




structor", formatter)) {
                exit(1);
        }

        char inFormatName[33];
        char outFormatName[33];
        char inFileName[128];
        char outFileName[128];
        char *pInFile = new char[10000];
```

```
        int nFileLen;
        while (1) {
                if (!Session || !Session->Ok()) {
                        cerr << "Database session broken, program terminating." << endl;
                        cin.get();
                        break;
                }

                // Get the input file name.
                cerr << "Enter the input file name: " << endl;
                if(!gets(inFileName)) break;
                if (!*inFileName) break;

                // Try to open the input file.
#ifdef WIN32
                ifstream inFile(inFileName, ios::binary|ios::nocreate);
#else
                ifstream inFile(inFileName);
#endif
                if (!inFile) {
                        cerr << "Invalid input file." << endl;
                        continue;
                }
                // Get the output file name.
                cerr << "Enter the output file name: " << endl;
                gets(outFileName);
                if (!*outFileName) {
                        break;
                }
#ifdef WIN32
                ofstream outFile(outFileName, ios::binary);
#else
                ofstream outFile(outFileName);
#endif

                // Get the input and output format names.
                cerr << "Enter the input format name: " << endl;
                gets(inFormatName);
                cerr << "Enter the output format name: " << endl;
                gets(outFormatName);

                // Read the input file.
                char c;
                char* p = pInFile;
                while (inFile.get(c)) {
                        *p++ = c;
                }
                int dw_newFileLen = p - pInFile;

                // This API call adds the entire file as a single input
                // message
                formatter->AddInputMessage(inFormatName,pInFile,dw_newFileLen);
                if (handleError("Formatter::AddInputMessage", formatter)) {
                        exit(1);
                }

                // You can add more input messages here, if desired.

                // Add an output format to reformat to.
                formatter->AddOutputFormat(outFormatName);
                if (handleError("Formatter::AddOutputFormat", formatter)) {
                        exit(1);
                }

                // You can add more output formats here, if desired.

                // Reformat the input message(s).  (Internally, this calls
                // Formatter::Parse prior to generating the output message(s).
                if (!formatter->Reformat()) {
                        if (handleError("Formatter::Reformat", formatter)) {
                                cin.get();
                        }
                }
                else {
                        OutMsgGroup* pOutMsgGroup;
                        OutMsg* pOutMsg;

                        // Get the output message group corresponding to the output
                        // format name.
                        pOutMsgGroup = formatter->GetOutMsgGroup(outFormatName);
```

```
                        if (pOutMsgGroup) {
                                int i=0;
                                int msgcnt=pOutMsgGroup->GetMsgCount();
                                cout << "Message count: " << msgcnt << endl;
                                for (;i<msgcnt;i++){
                                        // Get the output messages in the group.  Currently, there
                                        // is only one.
                                        pOutMsg = pOutMsgGroup->GetMsg(i);
                                        if (pOutMsg) {
                                                if (*outFileName) {
                                                       outFile.write(pOutMsg->GetMsgBuffer(),
                                                                pOutMsg->GetMsgLength());
                                                }
                                                else {
                                                        cout << endl <<"OUTPUT MESSAGE: " << endl;
                                                       cout.write(pOutMsg->GetMsgBuffer(),
                                                           pOutMsg->GetMsgLength());
                                                }
                                                cerr << endl;
                                                cerr << "Success.  Hit return." << endl;
                                                cin.get();
                                        }
                                        else {
                                            cerr << "Could not get the requested output message"<< endl;
                                               cerr << "\nERROR: " << "(" << formatter->GetErrorCode()
                                                      << ") " << formatter->GetErrorMessage() << "." <<
                                                           endl << endl;
                                                cerr << "Hit return." << endl;
                                                cin.get();
                                        }
                                }
                        }
                }
                inFile.close();
        }
        delete [] pInFile;
        CloseDbmsSession(Session);
}
```

GetValidationCallbacks Function

getval.ccp #1

Use the following function stub to run msgtest.cpp without validation callbacks.

```
static const char RCS_ID[] = "$Header:
/source/aig/formatter/test/Attic/getval.cpp,v 1.1.2.1 1997/02/21 21:14:12
aaron Exp $";

#include "nnuserfunction.h"

NNFunctionKeyPairCollection* GetValidationCallbacks();

NNFunctionKeyPairCollection* GetValidationCallbacks()
{
        return 0;
}
```

getval.cpp #2

Use the following function stub to run msgtest.cpp with validation callbacks.

```
static const char RCS_ID[] = "$Header:
/source/aig/formatter/test/Attic/minimalgetval.cpp,v 1.1.2.1 1997/02/25
20:14:41 aaron Exp $";

// System include files

extern "C" {
#include <stdio.h>
```

```
#include <string.h>
#include <time.h>
#include <errno.h>
#include <stdlib.h>
#include <memory.h>
#include <sys/types.h>
}

#include <iostream.h>
#include <fstream.h>

// Include files for database access

#include "interface.h"
#if defined(_MS_SQL_NT)
#include "sqlses.h"
#include "sqlapi.h"
#elif defined(sybase)
#include "sybfront.h"
#include "sybdb.h"
#endif

// Neonet include files

#include "dbtypes.h"
#include "ses.h"
#include "sqlapi.h"

#include "formatter.h"
#include "msgs.h"
#include "nnuserfunction.h"


struct CharPair
{
        char name[64];
        char time[64];
};


class MyCallback: public NNDBFieldsUserFunction
{
private:
        CharPair myRuntimeData;
        time_t  timeData;
        const char* fieldName;
        const char* fieldData;

public:
    MyCallback( const char* objectName )
        { sprintf( myRuntimeData.name, "%s", objectName ); }

    virtual ~MyCallback(){}

    virtual int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields )
        {
                fieldName = parsedFields.GetCurrInFldName();
                fieldData = parsedFields.GetCurrInFldAsciiData();
                cout << "MyCallback::Callback( dbsess, fields )\n"
                        << "\tobject name: " << myRuntimeData.name << "\n"
                        << "\tfield name : " << fieldName << "\n"
                        << "\tfield data : " << fieldData << endl;
                return 1;
        }
    virtual int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
        NameValuePair* nameValuePairArray)
        {
                fieldName = parsedFields.GetCurrInFldName();
                fieldData = parsedFields.GetCurrInFldAsciiData();
                cout << "MyCallback::Callback( dbsess, fields, name/valPairs
)\n"
                        << "\tobject name: " << myRuntimeData.name << "\n"
                        << "\tfield name : " << fieldName << "\n"
                        << "\tfield data : " << fieldData << endl;
                return 1;
        }
    virtual int Callback (
```

```
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
        void* userRuntimeData)
        {
                CharPair* urd( (CharPair*)userRuntimeData );
                fieldName = parsedFields.GetCurrInFldName();
                fieldData = parsedFields.GetCurrInFldAsciiData();
                cout << "MyCallback::Callback( dbsess, fields, runtimeData
)\n"
                        << "\tobject name: " << myRuntimeData.name << "\n"
                        << "\tfield name : " << fieldName << "\n"
                        << "\tfield data : " << fieldData << "\n"
                        << "\tval time   : " << urd->time << endl;
                return 1;
        }
    virtual int Callback (
        const DbmsSession& dbSession,
        const NNParsedFields& parsedFields,
        NameValuePair* nameValuePairArray,
        void* userRuntimeData)
        {
                CharPair* urd( (CharPair*)userRuntimeData );
                fieldName = parsedFields.GetCurrInFldName();
                fieldData = parsedFields.GetCurrInFldAsciiData();
                cout << "MyCallback::Callback( dbsess, fields, name/valPairs,
runtimeData )\n"
                        << "\tobject name: " << myRuntimeData.name << "\n"
                        << "\tfield name : " << fieldName << "\n"
                        << "\tfield data : " << fieldData << "\n"
                        << "\tval time   : " << urd->time << endl;
                return 1;
        }

    inline virtual void* RuntimeDataLookup(const char* parmName)
        {
                const char* a1 = "TestName";
                timeData = time( 0 );
                if( ! strcmp( parmName, a1 ))
                        cftime( myRuntimeData.time, "%c", &timeData );
                else
                        return 0;
                cout << "MyCallback::RuntimeDataLookup( data name )\n"
                        << ": Data Name: " << (parmName ? parmName : "(nil)")
                        << endl;
                return (void*)&myRuntimeData;
        }

    inline virtual int Cleanup (){ return 1; }
};


// make an object to hold sets of callback objects
//
NNFunctionKeyPairCollection keyPairs;

// make a callback object
//
const char myObjectName1[] = "objectName1";
const char myObjectName2[] = "objectName2";
MyCallback myCalls1( myObjectName1 );
MyCallback myCalls2( myObjectName2 );

const char myCallName1[] = "ValFunc2";

NNFunctionKeyPairCollection* GetValidationCallbacks();

NNFunctionKeyPairCollection* GetValidationCallbacks()
{
        keyPairs.AddPair( &myCalls1, myCallName1 );
        return &keyPairs;
}
```

Sample Program 2: apitest.cpp

Traversing a Parsed Message

This example illustrates how to traverse the structure of a parsed message.

```
static const char RCS_ID[] = "$Header: /u/users/clc/solaris-2.4.SparcWorks3.0.1.oracle-7-1.6/formatter/
test/RCS/apitest.cpp,v 1.3 1996/06/13 21:46:54 clc Exp $";

extern "C" {
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <stdlib.h>
#include <memory.h>
}

#include <iostream.h>
#include <fstream.h>

#include "interface.h"
#if defined(_MS_SQL_NT)
#include "sqlses.h"
#include "sqlapi.h"
#elif defined(sybase)
#include "sybfront.h"
#include "sybdb.h"
#endif

#include "dbtypes.h"
#include "ses.h"
#include "sqlapi.h"

#include "formatter.h"
#include "pmsg.h"
#include "pfield.h"
#include "fmtdefs.h"


// This program is a test driver for the parsed message APIs in the
// formatter engine. It asks the caller for an input file name and an
// input format name. It treats the contents of the input file as a single
// message with the input format specified, and then parses it and outputs
// the structure of the parsed message to standard out.
//
// The driver exercises the following functions:
//
//  Formatter::AddInputMessage
//  Formatter::Parse
//  ParsedMessage::GetInfo
//  ParsedMessage::GetCompCount
//  ParsedMessage::GetFieldComp
//  ParsedMessage::GetMsgComp
//  ParsedField::GetInfo
//  ParsedField::GetValue


// Handles error returned by formatter (outputs a message), and returns
// error code.
static int
handleError(char * func, Formatter * pFormatter ) {
    int code;

    if (code = pFormatter->GetErrorCode()) {
        cerr << "\nERROR during " << func << ": " << "(" << code << ") " << pFormatter-
>GetErrorMessage() << "." << endl << endl;
    }
    return code;
}

// Returns a string for a data type code.
static char *
```

```
codeToString(int code) {
    switch (code) {
    case DATA_TYPE_Not_Applicable:
        return "DATA_TYPE_Not_Applicable";
    case DATA_TYPE_Ascii_String:
        return "DATA_TYPE_Ascii_String";
    case DATA_TYPE_Ascii_Numeric:
        return "DATA_TYPE_Ascii_Numeric";
    case DATA_TYPE_Binary_Data:
        return "DATA_TYPE_Binary_Data";
    case DATA_TYPE_EBCDIC_Data:
        return "DATA_TYPE_EBCDIC_Data";
    case DATA_TYPE_IBM_Packed_Integer:
        return "DATA_TYPE_IBM_Packed_Integer";
    case DATA_TYPE_IBM_Signed_Packed_Integer:
        return "DATA_TYPE_IBM_Signed_Packed_Integer";
    case DATA_TYPE_IBM_Zoned_Integer:
        return "DATA_TYPE_IBM_Zoned_Integer";
    case DATA_TYPE_IBM_Signed_Zoned_Integer:
        return "DATA_TYPE_IBM_Signed_Zoned_Integer";
    case DATA_TYPE_Little_Endian2:
        return "DATA_TYPE_Little_Endian2";
    case DATA_TYPE_Little_Swap_Endian2:
        return "DATA_TYPE_Little_Swap_Endian2";
    case DATA_TYPE_Little_Endian4:
        return "DATA_TYPE_Little_Endian4";
    case DATA_TYPE_Little_Swap_Endian4:
        return "DATA_TYPE_Little_Swap_Endian4";
    case DATA_TYPE_Big_Endian2:
        return "DATA_TYPE_Big_Endian2";
    case DATA_TYPE_Big_Swap_Endian2:
        return "DATA_TYPE_Big_Swap_Endian2";
    case DATA_TYPE_Big_Endian4:
        return "DATA_TYPE_Big_Endian4";
    case DATA_TYPE_Big_Swap_Endian4:
        return "DATA_TYPE_Big_Swap_Endian4";
    default:
        return "Unknown code";
    }
}

// Indents output so that nested parsed structure is visible.
static void
doIndent(int indent) {
    int i;

    for (i = 0; i < indent; i++) {
        cerr << "  ";
    }
    return;
}

// Traverses parsed message and field structure, printing out contents.
// Returns 0 if success; 1 otherwise
static int
traverse(Formatter * pFormatter, int indent, ParsedMessage * pParsedMessage) {
    char *pFormatName;
    int      msgType;
    int      compCount;
    int      index;
    ParsedMessage *pSubMessage;
    ParsedField *pField;
    int      dataLength;
    int      dataType;
    char*pFieldData;
    char*pFieldName;
    charbuffer[BUFSIZ];
    char*pBuffer;
    int      i;

    // Get the format name of the parsed message.
    pFormatName = pParsedMessage->GetInfo(&msgType);

    // Get the count of components (fields or messages) in the message
    compCount = pParsedMessage->GetCompCount();

    if (msgType == FLAT_FORMAT) {
        doIndent(indent);
        cerr << "--- Flat format: " << pFormatName << endl;
        // Flat format: print out field names, values and data lengths.
        for (index = 0; index < compCount; index++) {
```

```
            pField = pParsedMessage->GetFieldComp(index);
            pFieldName = pField->GetInfo();
            pFieldData = pField->GetValue(&dataType, &dataLength);
            doIndent(indent);
            cerr << "Field[" << index << "] (" << pFieldName << "):" << endl;
            doIndent(indent);
            cerr << "Data type: " << codeToString(dataType) << endl;
            doIndent(indent);
            cerr << "Data: '";
            pBuffer = buffer;
            for (i = 0; i < dataLength; i++, pFieldData++) {
                pBuffer += sprintf(pBuffer, "%c", *pFieldData);
            }
            *pBuffer = '\0';
            cerr << buffer << "'" << endl;
        }
    } else {
        doIndent(indent);
        // Compound format: traverse each of component formats
        cerr << "+++ Compound format: " << pFormatName << endl;
        for (index = 0; index < compCount; index++) {
            pSubMessage = pParsedMessage->GetMsgComp(index);
            if (traverse(pFormatter, indent+1, pSubMessage)) {
                return 1;
            }
        }
    }
    return 0;
}

int
main()
{
    int msgCount, msgIndex;
    ParsedMessage *pParsedMessage;

#ifdef oracle
    DbmsSession *Session = OpenDbmsSession("new_format_demo",ORACLE7);
#else
    DbmsSession *Session = OpenDbmsSession("new_format_demo",SYBASE49);
#endif

    if ( !Session || !Session->Ok()){
        cerr << "No session created" << endl;
        cin.get();
        exit(errno);
    }
    Formatter formatter(Session);
    if (handleError("formatter constructor", &formatter)) {
        exit(1);
    }

    char inFormatName[33];
    char inFileName[128];
    char *pInFile = new char[10000];

    int nFileLen;
    while (1) {
        if (!Session || !Session->Ok()) {
            cerr << "Database session broken, program terminating." << endl;
            cin.get();
            break;
        }

        cerr << "Enter the input file name: " << endl;
        gets(inFileName);
        if (!*inFileName) break;

        ifstream inFile(inFileName);
        if (!inFile) {
            cerr << "Invalid input file." << endl;
            continue;
        }
        cerr << "Enter the input format name: " << endl;
        gets(inFormatName);

        // read the file
        char c;
        char* p = pInFile;
        while (inFile.get(c)) {
            *p++ = c;
```

```
        }
        int dw_newFileLen = p - pInFile;

        // this API call adds the entire file as a single input
        // message
        formatter.AddInputMessage(inFormatName,pInFile,dw_newFileLen);
        if (handleError("Formatter::AddInputMessage", &formatter)) {
            exit(1);
        }
        // add more input messages here if desired.

        // Parse the message
        formatter.Parse();
        if (!handleError("Formatter::Parse", &formatter)) {
            // Get parsed message count (same number as number of
            // AddInputMessage functions called).
            msgCount = formatter.GetParsedInMsgCount();
            if (handleError("Formatter::GetParsedInMsgCount", &formatter)) {
                exit(1);
            }
            for (msgIndex = 0; msgIndex < msgCount; msgIndex++) {
                // Get and traverse each parsed message.
                pParsedMessage = formatter.GetParsedInMsg(msgIndex);
                if (handleError("Formatter::GetParsedInMsg", &formatter)) {
                    exit(1);
                }
                if (traverse(&formatter, 0, pParsedMessage)) {
                    exit(1);
                }
            }
        } else {
            exit(1);
        }
        inFile.close();
    }
    delete [] pInFile;
}
```

Appendix B

Access Mode Examples

Access Mode Types

Access Mode	Description
Not Applicable	Accesses no field in the input message.
Normal Access	Access the instance in the same repeating component as the current controlling field instance. If there is no controlling field, access the first instance. This access mode behaves just like “Access sibling instance”.
Access with Increment	When the last child of a parent is accessed, increment the parent index. A field with this access mode is the controlling field for the repeating component (See “Controlling field”).
Access the nth Instance of Field	Always access the nth instance of a field in the input message. (When n=0, then get the first instance.)
Access within Compound	Accesses child with the same index as accessed in the previous format.
Cycling Access, stay in Compound	When the last field in a compound is accessed, go back to the first field.
Access using relative index	The first field in a repeating component that Formatter encounters with this access mode is the controlling field for the repeating component (see “Controlling field”). Any other field in the repeating component with this access mode behaves as if it has access mode “Access sibling instance” or “Normal access” (access the sibling of the controlling field).
Controlling Field	Marks this field as the controlling field of the repeating component. On each repetition, access the next field instance that is still a child of the current controlling field instance of the parent format. If there is no parent controlling field, the repetitions end with the last field instance from the input message.
Access next instance	Access the next field instance relative to the previous access.
Access parent instance	Access the instance in the hierarchy above the controlling field.
Access sibling instance	Access the instance in the same repeating component as the current controlling field instance. If there is no controlling field, access the first instance.
Access current instance	Access the same field instance as on the previous access. (The first access will get the first instance of the field.)

In the examples provided, the following notation represents format definitions. Indents indicate different "levels" of a particular format definition.

- Fn: Represents a field with the name "Fn," such as F1 or F2.
- (): Items contained within parentheses represent field definitions within the same flat format. (F1 F2) indicates a flat format with two fields, F1 and F2.
- {}: Items contained within braces indicate a repeating component.
- / and \: Items contained within forward and backward slashes indicate an alternative format. In alternative formats, only one alternative applies for each single or repeating component.
- []: Items contained within brackets indicate an optional component.

Formatting Nonrepeating Messages into Nonrepeating Messages--
Use "Normal Access" when formatting an input message with no repeating components into an output message with no repeating components, so the input and output messages have the same basic structure. In this instance, you could also use "Access Current Instance" or "Access Nth Instance" where N=0.
For example, if you have an input format describing three levels of nesting in an input message (F0 F1 F2 F3), then you could use "Normal Access" to describe the output format as:

(F0 - Normal Access

F1 - Normal Access

F2 - Normal Access

F3 - Normal Access)

Formatting Nested Messages into Nested Messages with a Similar Structure--
Use combinations of "Controlling Field" and "Access Sibling Instance" when you want to format an input message with nested repeating components into an output message with similar structure and contents.

For example, if you have an input format describing three levels of nesting in an input message--

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
```

```
}
```

--you could use "Controlling Field" and "Access Sibling Instance" access modes as follows:

```
{
    (F0 - Controlling Field)
    {
        (F1 - Controlling Field)
        {
            (F2  - Controlling Field
            F3 - Access Sibling Instance
            F4 - Access Sibling Instance)
        }
    }
}
```

Outputting the Same Field Twice in a Repeating Component--
Use combinations of "Controlling Field," "Access Sibling Instance," and "Access Current Instance" when you want to you want to format an input message with repeating components into an output message where a particular field is output more than once in each repetition.

For example, if you have an input format describing three levels of nesting in an input message--

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

--you could use "Controlling Field," "Access Sibling Instance," and "Access Current Instance" access modes, if F2 is the field to be repeated:

```
{
    (F0 - Controlling Field)
    {
```

```
            (F1 - Controlling Field)
            {
                (F2  - Controlling Field
                F3 - Access Sibling Instance
                F4 - Access Sibling Instance
                F2 - Access Current Instance)
            }
        }
}
```

Formatting a Nested Format into a Nested Format with a Missing Intermediate Level of Nesting--

Use combinations of "Controlling Field" and "Access Sibling Instance" when you want to format an input message with nested repeating components into an output message missing one of the intermediate nesting levels in the input format.

For example, if you have an input format describing three levels of nesting in an input message--

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

--you could use "Controlling Field" and "Access Sibling Instance" access modes when you want to eliminate the nesting level containing field F1:

```
{
    (F0 - Controlling Field)
    {
        (F2  - Controlling Field
        F3 - Access Sibling Instance
        F4 - Access Sibling Instance)
    }
}
```

Formatting a Nested Format into a Flattened Format-- Use combinations of "Controlling Field," "Access Parent Instance" and "Access Sibling Instance" whenº you want to flatten a nested input message so the parent fields are output with each child field instance.

For example, if you have an input format describing three levels of nesting in an input message--

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

--you could eliminate the nesting level containing field F1, by using "Controlling Field" and "Access Sibling Instance" access modes as follows:

```
{
    (F0 - Access Parent Instance)
    (F1 - Access Parent Instance)
    (F2 - Controlling Field
    F3 - Access Sibling Instance
    F4 - Access Sibling Instance)
}
```

Formatting an Alternative with Overlapping Field Names-- Use combinations of "Controlling Field" and "Access Sibling Instance" when you have an input format definition for a format with alternative components and want to output it using the same alternative structure.

If your input format definition look like this--

```
{
    (F0)
    {
        (
            /
                (F1 F2)
                (
                    /
```

```
                    (F3)
                \
                /
                    (F4 F5)
                \
            )
            {
                (F6)
            }
        \
        /
            (F1)
            (
                /
                    (F8)
                \
                /
                    (F9)
                \
            )
            {
                (F6)
            }
        \
    )
}
}
```

--your output format definition should be constructed like this. Since each alternative includes F1, you can use it as the Controlling Field for the alternative.

```
{
    (F0 - Normal Access)
    {
        (
            /
                (F1 - Controlling Field
```

```
                    F2 - Access Sibling Instance)

(
                    /
                        (F3 - Access Sibling Instance)
                    \
                    /
                        (F4 - Access Sibling Instance
                            F5 - Access Sibling Instance)

\
                )
                {
                    (F6 - Controlling Field)
                }
            \
            /
                (F1 - Controlling Field)
                (
                    /
                        (F8 - Access Sibling Instance)
                    \
                    /
                        (F9 - Access Sibling Instance)
                    \
                )
                {
                    (F6 - Controlling Field)
                }
            \
        )
    }
}
```

Formatting a Floating Alternative with Overlapping Field Names-- Use combinations of "Controlling Field" and "Access Sibling Instance" when you have an input format definition for a format with floating alternative

components and want to output it using the same alternative structure. The alternative components are considered "floating," because for each repetition of the alternative, there is no field to be coupled with the alternative structure at output.

If your input format definition look like this:

```
{
    /
        (F1 F2 F3)
    \
    /
        (F1 F2)
    \
    /
        (F2 F3)
    \
}
```

Your output format definition should be constructed like this. Because there is no anchoring field for each alternative, each one needs its own controlling field.

```
{
    /
        (F1 - Controlling Field
        F2 - Access Sibling Instance
        F3 - Access Sibling Instance)
    \
    /
        (F1 - Controlling Field
        F2 - Access Sibling Instance)
    \
    /
        (F2 - Controlling Field
        F3 - Access Sibling Instance)
    \
}
```

Formatting an Output Message with Optional Repeating Components-- Use combinations of "Controlling Field" and "Access Sibling Instance" when you have an input format definition for a repeating format with optional components.

If your input format definition looks like this:

{

[(F1 F2)]

(F3 F4 F5)

}

If you want our output message to be output with a similar structure and contents, the output format would have the following structure:

{

[(F1 - Access Sibling Instance

F2 - Access Sibling Instance)]

(F3 - Controlling Field

F4 - Access Sibling Instance

F5 - Access Sibling Instance)

}

F3 always appears in each repetition of the format, so you can use it as the controlling field. You could also have selected F4 or F5.

Formatting an Output Message with Boolean Fields in Repeating Components-- Use combinations of "Controlling Field" and "Access Sibling Instance" when you have an input format definition for a repeating format that uses Boolean controls. The "Input Field =" and "Field Exists" output control types are considered Boolean - included in the output message if they meet the specified condition.

If your input format definition looks like this--

{

(F1 F2 F3)

}

--then the output format definition (assuming you use a Boolean control for F1 where "Field = 'X'") would be structured as follows:

{

(F1 (Boolean) - Access Sibling Instance

F1 - Controlling Field

F2 - Access sibling instance

F3 - Access sibling instance)

}

In order to conduct an evaluation on the instance of F1 in that repetition, you should choose "Access Sibling Instance." The sibling of F1 is itself, and you

could also have chosen "Access Current Instance." You would not want to use "Controlling Field" for the Boolean condition because F1 would be incremented twice for the same repetition.

Appendix C

Code Example for Substitute Controls

```
 NNFMgrSubstituteCntlInfo info;
int totalEntries = 10;
// This number is up to you, I just used 10 as an example
int numRemainingEntries;
short ret;
 // Code to get an NNFMgr object named pNNFMgr ...
 // Initialize the info structure
 NNF_CLEAR(&info);
 // Set the structure values
 strcpy(info.cntlName, “My Substitute”);
memcpy(info.inputValue, “abc”);
// or use inputLitrlName instead
info.inputValueLen = 3;
memcpy(info.outputValue, “xyz”);
// or use outputLitrlName instead
info.outputValueLen = 3;
info.outputValueType = DATA_TYPE_Ascii_String;
 // Create a new substitute control and the first substitute entry
 if (!NNFMgrCreateSubstituteCntl(pNNFMgr, &info))
{
  // error
}
// Create the remaining entries
 for (int i=1; i < totalEntries; i++)
{
  // I’m not setting cntlName because I want to add this to the end of the
control
  // we created in NNFMgrCreateSubstituteCntl.
  sprintf(info.inputValue, “inval%02d”, i);
  info.inputValueLen = 7;
```

```
  sprintf(info.outputValue, “outval%02d”, i);
  info.outputValueLen = 8;
  / / I’m not setting outputValueType because I want all entries to act the
same
  / / in a single control
  / / Append the entry to the substitute control
   if (!NNFMgrAppendEntryToSubstituteCntl(pNNFMgr, &info)
  {
    / / error
  }
 }
  / / Now let’s get the entries back
 / / Just to be clear, I will set the cntlName again.  All that matters is that
 / / the cntlName you specify is an existing control.
 NNF_CLEAR(&info);
  strcpy(info.cntlName, “My Substitute”);
  / / Get the first entry from the control
  if (!NNFMgrGetSubstituteCntl(pNNFMgr, GET, &info,
&numRemainingEntries))
 {
   / / error
 }


 / / We got the first entry.  We can print it out, or store it.  If you want to store
 / / all the entries for this control, a good way to do it is to allocate an array of
 / / NNFMgrSubstituteCntlInfo structures.  The size of this array will be
 / / numRemainingEntries + 1 (for the entry we already got).
 NNFMgrSubstituteCntlInfo* entries = new
NNFMgrSubstituteCntlInfo[numRemainingEntries + 1];
 / / Store the first entry
 entries[0] = info; / / this is equivalent to a memcpy from one struct to the
other
  / / Get and store the remaining entries
  for (int i=1; i < numRemainingEntries + 1; i++)
 {
  if (!NNFMgrGetNextEntryFromSubstituteCntl(pNNFMgr, &info))
  {
```

```
    // error
  }
  entries[i] = info;
}
```

Appendix D
OpCode

- If OpCode is ‘GET’ the user provides the control name in the input structure, and this function fills in the rest of the structure with information from the control record with the given name.
- If OpCode is ‘GET_FIRST’, this function returns the first row of the control table (no input data from the input structure is used).
- If OpCode is ‘GET_NEXT’, this function returns the next row from the control table (user must have called this function previously with an OpCode equal to ‘GET_FIRST’).

Appendix E

Data Type Descriptions

Supported Data Types

Data Type Field Values	Data Type #Define	Description
Not Applicable	DATA_TYPE_Not_Applicable	No data type is assumed.
ASCII String	DATA_TYPE_Ascii_String	A string of standard ASCII characters. Note that non-printable characters are valid as long as they are in the ASCII character set. (EBCDIC characters outside the valid AsciiString range are not valid AsciiString characters. During a reformat from Ascii to EBCDIC if a character being converted is not in the EBCDIC character set the conversion results in a EBCDIC space (hexadecimal 40)).
ASCII Numeric	DATA_TYPE_Ascii_Numeric	A string of standard ASCII numeric characters. Note that the '-' and '.' characters are not valid ASCII numeric characters.

Data Type Field Values	Data Type #Define	Description
Binary Data	DATA_TYPE_ Binary_Data	The Binary data type is used to parse any value and transform that value to an ASCII representation of the value internally in the Formatter. The internal representation takes each byte of the input value and converts it to a readable form. An example of this is parsing a byte whose value is (hexadecimal) 0x9C and transforming that to the internal ASCII representation of 9C, which is the hexadecimal value 0x3943. If this value is used in an output format with the output control's data type set to String, the value placed in the message is ASCII 0x9C. If this value is again placed in an output message with the data type Binary, the ASCII value is not printable and occupies one byte with the value of (hexadecimal) 0x9C. Conversely, an input value of ASCII 3B7A parsed with the String data type can be output using the Binary data type. The output value is (hexadecimal) 0x37BA and occupies 2 bytes in the output message. Valid characters that can be converted to Binary from the String data type are 0 through 9 and A through F. All other characters are invalid.
EBCDIC Data	DATA_TYPE_ EBCDIC_Data	A string of characters encoded using the EBCDIC (Extended Binary Coded Decimal Interexchange Code) encoding used on larger IBM computers. During a reformat from EBCDIC to Ascii if a character being converted is not in the EBCDIC character set the conversion results in a space (hexadecimal) 0x20.

Data Type Field Values	Data Type #Define	Description
IBM Packed Integer	DATA_TYPE_IBM _Packed_Integer	Data type on larger IBM computers used to represent integers in compact form. Each byte represents two decimal digits, one in each nibble of the byte. The final nibble is always a hexadecimal 'F'. For example, the number "1234" is stored as a 3 byte value: "01 23 4F" (the number pairs show the hexadecimal values of the nibbles of each byte). The number "12345" is stored as a 3 byte value: "12 34 5F". There is no accounting for the sign of a number, so all numbers are assumed to be positive.
IBM Signed Packed Integer	DATA_TYPE_IBM _Signed_Packed_ Integer	Data type on larger IBM computers used to represent integers in compact form. This data type takes into account the sign (positive or negative) of a number. Each byte represents two decimal digits, one in each nibble of the byte. The final nibble is a hexadecimal 'C' if the number is positive, and a hexadecimal 'D' if the number is negative. For example, the number "1234" is stored as a 3 byte value: "01 23 4C" (the number pairs show the hexadecimal values of the nibbles of each byte). The number "-12345" is stored as a 3 byte value: "12 34 5D".
IBM Zoned Integer	DATA_TYPE_IBM _Zoned_Integer	Data type on larger IBM computers used to represent integers. Each decimal digit is represented by a byte. The left nibble of the byte is a hexadecimal 'F'. The right nibble is the hexadecimal value of the digit. For example, "1234" is represented as "F1 F2 F3 F4" (the number pairs show the hexadecimal values of the nibbles of each byte).

Data Type Field Values	Data Type #Define	Description
IBM Signed Zoned Integer	DATA_TYPE_IBM _Signed_Zoned_ Integer	Data type on larger IBM computers used to represent integers. Each decimal digit is represented by a byte. The left nibble of each byte, EXCEPT THE LAST BYTE, is a hexadecimal 'F'. The left nibble of the last byte is a hexadecimal 'C' if the number is positive, and a hexadecimal 'D' if the number is negative. The right nibble of each byte is the hexadecimal value of the digit. For example, "1234" is represented as "F1 F2 F3 C4" (the number pairs show the hexadecimal values of the nibbles of each byte). "-1234" is represented as "F1 F2 F3 D4".
Little Endian 2	DATA_TYPE_ Little_Endian2	Two-byte integer where the bytes are ordered with the rightmost byte being the high order or most significant byte. For example, the hexadecimal number "0x0102" is stored as "02 01" (where the number pairs show the hexadecimal values of the nibbles of a byte).
Little Swap Endian 2	DATA_TYPE_ Little_Swap_ Endian2	Two-byte integer where the two bytes are swapped with respect to a "Little Endian 2" value. For example, the hexadecimal number "0x0102" is stored as "01 02".
Little Endian 4	DATA_TYPE_ Little_Endian4	Four-byte integer where the bytes are ordered with the rightmost byte being the high order or most significant byte. For example, the hexadecimal number "0x01020304" is stored as "04 03 02 01" (where the number pairs show the hexadecimal values of the nibbles of a byte).
Little Swap Endian 4	DATA_TYPE_ Little_Swap_ Endian4	Four-byte integer where the two bytes of each word are swapped with respect to a "Little Endian 4" value. For example, the hexadecimal number "0x01020304" is stored as "03 04 01 02".

Data Type Field Values	Data Type #Define	Description
Big Endian 2	DATA_TYPE_Big_Endian2	Two-byte integer where the bytes are ordered with the leftmost byte being the high order or most significant byte. For example, the hexadecimal number “0x0102” is stored as “01 02” (where the number pairs show the hexadecimal values of the nibbles of a byte).
Big Swap Endian 2	DATA_TYPE_Big_Swap_Endian2	Two-byte integer where the two bytes are swapped with respect to a “Big Endian 2” value. For example, the hexadecimal number “0x0102” is stored as “02 01”.
Big Endian 4	DATA_TYPE_Big_Endian4	Four-byte integer where the bytes are ordered with the leftmost byte being the high order or most significant byte. For example, the hexadecimal number “0x01020304” is stored as “01 02 03 04” (where the number pairs show the hexadecimal values of the nibbles of a byte).
Big Swap Endian 4	DATA_TYPE_Big_Swap_Endian4	Four-byte integer where the two bytes of each word are swapped with respect to a “Big Endian 4” value. For example, the hexadecimal number “0x01020304” is stored as “02 01 04 03”.
Decimal, International	DATA_TYPE_Decimal_International	Data type where every third number left of the decimal point is preceded by a period. The decimal point is represented by a comma. Numbers right of the decimal point represent a fraction of one unit. For example, the number “12345.678” is represented as “12.345,678”. Decimal international datatypes can contain negative values.

Data Type Field Values	Data Type #Define	Description
Decimal, U.S.	DATA_TYPE_ Decimal_US	Data type where every third number left of the decimal point is preceded by a comma. The decimal point is represented by a period. Numbers right of the decimal point represent a fraction of one unit. For example, the number “12345.678” is represented as “12,345.678”. Decimal US datatypes can contain negative values.
Unsigned Little Endian 2	DATA_TYPE_Unsigned_LittleEndian 2	Like “Little Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Little Swap Endian 2	DATA_TYPE_ Unsigned_LittleSwapEndian2	Like “Little Swap Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Little Endian 4	DATA_TYPE_ Unsigned_ LittleEndian4	Like “Little Endian 4”, except that the value is interpreted as an unsigned value.
Unsigned Little Swap Endian 4	DATA_TYPE_ Unsigned_ LittleSwapEndian4	Like “Little Swap Endian 4”, except that the value is interpreted as an unsigned value.
Unsigned Big Endian 2	DATA_TYPE_ Unsigned_ BigEndian2	Like “Big Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Big Swap Endian 2	DATA_TYPE_ Unsigned_ BigSwapEndian2	Like “Big Swap Endian 2”, except that the value is interpreted as an unsigned value.
Unsigned Big Endian 4	DATA_TYPE_ Unsigned_ BigEndian4	Like “Big Endian 4”, except that the value is interpreted as an unsigned value
Unsigned Big Swap Endian 4	DATA_TYPE_ Unsigned_ BigSwapEndian4	Like “Big Swap Endian 4”, except that the value is interpreted as an unsigned value.

Data Type Field Values	Data Type #Define	Description
Date and Time		Based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDhhmmss. See the first paragraph of each of the Date and Time type descriptions for details on representing Date and Time components. Combined dates and times may be represented in any of the following list of data types. The list includes String, Numeric, and EBCDIC.
Time		Based on the international ISO-8601:1988 standard time notation: HHmmss where HH represents the number of complete hours that have passed since midnight (between 00 and 23), mm is the number of minutes passed since the start of the hour (between 00 and 59), and ss is the number of seconds since the start of the minute (between 00 and 59). Times are represented in 24-hour format. Times may be represented in any of the following list of data types. For some data types, a minimum of 4 bytes is required. The list includes: EBCDIC, String, and Numeric.
Date		Based on the international ISO-8601:1988 standard date notation: YYYYMNDD where YYYY represents the year in the usual Gregorian calendar, MM is the month between 01 (January) and 12 (December), and DD is the day of the month with a value between 01 and 31. Dates may be represented in any of the following list of data types. For some data types, a minimum of 4 bytes is required. The list includes: EBCDIC, String, and Numeric.

Data Type Field Values	Data Type #Define	Description
Custom Date and Time		Custom Date and Time enables users to specify different formats of dates, times, and combined dates and times. Date / Time formats may include: 1) Variations in year (2 or 4 digit year representation: YY or YYYY). 2) Variations in month –use of a month number (01-12) or three letter abbreviation (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC). The format string for month numbers is “MN”. The format string of three letter abbreviations is “MON”. 3) Variations in the day of the month – use of a day of the month number (01-31). “DD” is the format string. 4) Variations in hour – 12-hour or 24-hour representation, with or without a meridian indicator (AM or PM) Hours / minutes / seconds are represented as HHMMSS. 5) Valid data types include EBCDIC, String, and Numeric. For information on how to set the Year Cutoff value once Custom Date and Time has been selected refer to the section *Specifying a Year Cutoff Value* .

Notes for Data Conversion

Formatter can convert data between any two supported types via an intermediary representation. The data conversion occurs when Formatter, during a reformat(), encounters an input field with one data type and a different data type for the output field.

Certain pairs of data conversions don’t make much sense. For example, if you have a String in the input message with non-numeric data and the output format specifies that the data type for the field should be “IBM packed integer,” that conversion cannot happen correctly. Formatter will generate an error message indicating invalid data.

Formatter does not have a conversion function for every pair of native data types. Instead, Formatter converts data internally from the input data type to an String representation, and then from the String representation to the

output data type. So instead of (K**2 - K) conversion functions, Formatter has (K*2 - 2) functions, where K is the number of native data types.

For example, let's say you're converting from IBM signed packed integer to IBM packed integer. Your input is:

12 34 56 7C (where each pair of numbers are the 2 nibbles of a byte of data)

The data length is 4 bytes and the data represents the number "+1234567." The "C" is a sign nibble indicating the number is positive.

Formatter converts this to the String "+1234567," then converts the String to IBM packed data:

12 34 56 7F

When binary data (DATA_TYPE_Binary_Data) is involved, it means that the bytes of data can have any value without restriction or interpretation. If you have a field in the input format that's in binary and the corresponding field in the output format is also binary, what does Formatter do?

For example, if you have:

12 34 56 78 90 ab cd ef

where each pair of numbers are the 2 nibbles (in hexidecimal encoding) of a byte of data.

Formatter first converts this data to an ASCII string representation of the binary data:

"1234567890abcdef"

and then converts this ASCII string back to binary data:

12 34 56 78 90 ab cd ef

What if you're converting between an ASCII string and binary? Formatter will expect the ASCII string to be a proper representation of a binary value. Let's say you have the input:

"Hello, world!"

and you want Formatter to generate a binary value on output.

Formatter will issue an error, since the ASCII string isn't a proper string representation of a binary value. The string needs to be composed of the characters 0-9 and A-F.

So how do you get Formatter to take the binary values of the string "Hello, world!" and generate the proper binary output for it?

The actual binary encoding of the ASCII string "Hello, world!" is:

48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21

You should specify that the data type of the input is binary, not ASCII string. Formatter will generate the internal ASCII string:

"48656c6c6f2c20776f726c6421"

and then convert this back to binary:

48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21

The point here is that if you want Formatter to construct a binary value that equates to the acutal byte values of your input, you will want to specify that the input data type is also Binary. If you have an input data type other than Binary, Formatter will attempt to interpret your input as the string representation of a binary value.

Appendix F

ASCII Extended Character Set

Decimal Value from 000 to 051

Decimal Value	Hex Value	Extended Character Set		Decimal Value	Hex Value	Extended Character Set
000	00	NUL		026	1A	SUB
001	01	SCH		027	1B	ESCAPE
002	02	STX		028	1C	FS
003	03	ETX		029	1D	GS
004	04	EOT		030	1E	RS
005	05	ENO		031	1F	US
006	06	ACK		032	20	SPACE
007	07	BEL		033	21	!
008	08	BS		034	22	“
009	09	HT		035	23	#
010	0A	LF		036	24	$
011	0B	VT		037	25	%
012	0C	FF		038	26	&
013	0D	CR		039	27	‘
014	0E	SO		040	28	(
015	0F	SI		041	29	)
016	10	DLE		042	2A	*
017	11	DC1		043	2B	+
018	12	DC2		044	2C	,
019	13	DC3		045	2D	-
020	14	DC4		046	2E	.
021	15	NAK		047	2F	/
022	16	SYN		048	30	0
023	17	ETB		049	31	1
024	18	CAN		050	32	2
025	19	EM		051	33	3

Decimal Value from 052 to 115

Decimal Value	Hex Value	Extended Character Set		Decimal Value	Hex Value	Extended Character Set
052	34	4		084	54	T
053	35	5		085	55	U
054	36	6		086	56	V
055	37	7		087	57	W
056	38	8		088	58	X
057	39	9		089	59	Y
058	3A	:		090	5A	Z
059	3B	;		091	5B	[
060	3C	<		092	5C	\
061	3D	=		093	5D	]
062	3E	>		094	5E	^
063	3F	?		095	5F	_
064	40	@		096	60	‘
065	41	A		097	61	a
066	42	B		098	62	b
067	43	C		099	63	c
068	44	D		100	64	d
069	45	E		101	65	e
070	46	F		102	66	f
071	47	G		103	67	g
072	48	H		104	68	h
073	49	I		105	69	i
074	4A	J		106	6A	j
075	4B	K		107	6B	k
076	4C	L		108	6C	l
077	4D	M		109	6D	m
078	4E	N		110	6E	n
079	4F	O		111	6F	o
080	50	P		112	70	p
081	51	Q		113	71	q
082	52	R		114	72	r
083	53	S		115	73	s

Decimal Value from 116 to 179

Decimal Value	Hex Value	Extended Character Set		Decimal Value	Hex Value	Extended Character Set
116	74	t		148	94	”
117	75	u		149	95	•
118	76	v		150	96	–
119	77	w		151	97	—
120	78	x		152	98	˜
121	79	y		153	99	™
122	7A	z		154	9A	š
123	7B	{		155	9B	›
124	7C	|		156	9C	œ
125	7D	}		157	9D	unused
126	7E	~		158	9E	unused
127	7F	DEL		159	9F	Ÿ
128	80	unused		160	A0	nonbreaking space
129	81	unused		161	A1	¡
130	82	,		162	A2	¢
131	83	*f*		163	A3	£
132	84	„		164	A4	¤
133	85	…		165	A5	¥
134	86	†		166	A6	¦
135	87	‡		167	A7	§
136	88	ˆ		168	A8	¨
137	89	‰		169	A9	©
138	8A	Š		170	AA	ª
139	8B	‹		171	AB	«
140	8C	Œ		172	AC	¬
141	8D	unused		173	AD	-
142	8E	unused		174	AE	®
143	8F	unused		175	AF	¯
144	90	unused		176	B0	°
145	91	‘		177	B1	±
146	92	‘		178	B2	²
147	93	“		179	B3	³

Decimal Value from 180 to 243

Decimal Value	Hex Value	Extended Character Set		Decimal Value	Hex Value	Extended Character Set
180	B4	´		212	D4	Ô
181	B5	µ		213	D5	Õ
182	B6	¶		214	D6	Ö
183	B7	·		215	D7	×
184	B8	¸		216	D8	Ø
185	B9	¹		217	D9	Ù
186	BA	º		218	DA	Ú
187	BB	»		219	DB	Û
188	BC	¼		220	DC	Ü
189	BD	½		221	DD	Ý
190	BE	¾		222	DE	Þ
191	BF	¿		223	DF	ß
192	C0	À		224	E0	à
193	C1	Á		225	E1	á
194	C2	Â		226	E2	â
195	C3	Ã		227	E3	ã
196	C4	Ä		228	E4	ä
197	C5	Å		229	E5	å
198	C6	Æ		230	E6	æ
199	C7	Ç		231	E7	ç
200	C8	È		232	E8	è
201	C9	É		233	E9	é
202	CA	Ê		234	EA	ê
203	CB	Ë		235	EB	ë
204	CC	Ì		236	EC	ì
205	CD	Í		237	ED	í
206	CE	Î		238	EE	î
207	CF	Ï		239	EF	ï
208	D0	Ð		240	F0	ð
209	D1	Ñ		241	F1	ñ
210	D2	Ò		242	F2	ò
211	D3	Ó		243	F3	ó

Decimal Value 244 to 255

Decimal Value	Hex Value	Extended Character Set		Decimal Value	Hex Value	Extended Character Set
244	F4	ô		250	FA	ú
245	F5	õ		251	FB	û
246	F6	ö		252	FC	ü
247	F7	÷		253	FD	ý
248	F8	ø		254	FE	þ
249	F9	ù		255	FF	ÿ

Appendix G

EBCDIC Character Set

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
000	00	NUL	Null	0000 0000
001	01	SOH	Start of Heading	0000 0001
002	02	STX	Start of Text	0000 0010
003	03	ETX	End of Text	0000 0011
004	04	SEL	Select	0000 0100
005	05	HT	Horizontal Tab	0000 0101
006	06	RNL	Required New Line	0000 0110
007	07	DEL	Delete	0000 0111
008	08	GE	Graphic Escape	0000 1000
009	09	SPS	Superscript	0000 1001
010	0A	RPT	Repeat	0000 1010
011	0B	VT	Vertical Tab	0000 1011
012	0C	FF	Form Feed	0000 1100
013	0D	CR	Carriage Return	0000 1101
014	0E	SO	Shift Out	0000 1110
015	0F	SI	Shift In	0000 1111
016	10	DLE	Data Link Escape	0001 0000
017	11	DC1	Device Control 1	0001 0001
018	12	DC2	Device Control 2	0001 0010
019	13	DC3	Device Control 3	0001 0011
020	14	RES/ENP	Restore/Enable Presentation	0001 0100
021	15	NL	New Line	0001 0101
022	16	BS	Backspace	0001 0110
023	17	POC	Program-Operator Communication	0001 0111
024	18	CAN	Cancel	0001 1000
025	19	EM	End of Medium	0001 1001

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
026	1A	UBS	Unit Backspace	0001 1010
027	1B	CU1	Customer Use 1	0001 1011
028	1C	IFS	Interchange File Separator	0001 1100
029	1D	IGS	Interchange Group Separator	0001 1101
030	1E	IRS	Interchange Record Separator	0001 1110
031	1F	IBT/IUS	Intermediate Transmission Block/Interchange Unit Separator	0001 1111
032	20	DS	Digit Select	0010 0000
033	21	SOS	Start of Significance	0010 0001
034	22	FS	Field Separator	0010 0010
035	23	WUS	Word Underscore	0010 0011
036	24	BYP/INP	Bypass/Inhibit Presentation	0010 0100
037	25	LF	Line Feed	0010 0101
038	26	ETB	End of Transmission Block	0010 0110
039	27	ESC	Escape	0010 0111
040	28	SA	Set Attribute	0010 1000
041	29	SFE	Start Field Extended	0010 1001
042	2A	SM/SW	Set Mode/Switch	0010 1010
043	2B	CSP	Control Sequence Prefix	0010 1011
044	2C	MFA	Modify Field Attribute	0010 1100
045	2D	ENQ	Enquiry	0010 1101
046	2E	ACK	Acknowledge	0010 1110
047	2F	BEL	Bell	0010 1111
048	30			0011 0000
049	31			0011 0001
050	32	SYN	Synchronous Idle	0011 0010
051	33	IR	Index Return	0011 0011
052	34	PP	Presentation Position	0011 0100
053	35	TRN	Transparent	0011 0101
054	36	NBS	Numeric Backspace	0011 0110
055	37	EOT	End of Transmission	0011 0111

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
056	38	SBS	Subscript	0011 1000
057	39	IT	Indent Tab	0011 1001
058	3A	RFF	Required Form Feed	0011 1010
059	3B	CU3	Customer Use 3	0011 1011
060	3C	DC4	Device Control 4	0011 1100
061	3D	NAK	Negative Acknowledge	0011 1101
062	3E			0011 1110
063	3F	SUB	Substitute	0011 1111
064	40	SP	Space	0100 0000
065	41	RSP		0100 0001
066	42			0100 0010
067	43			0100 0011
068	44			0100 0100
069	45			0100 0101
070	46			0100 0110
071	47			0100 0111
072	48			0100 1000
073	49			0100 1001
074	4A	¢		0100 1010
075	4B	.		0100 1011
076	4C	<		0100 1100
077	4D	(		0100 1101
078	4E	+		0100 1110
079	4F	|		0100 1111
080	50	&		0101 0000
081	51			0101 0001
082	52			0101 0010
083	53			0101 0011
084	54			0101 0100
085	55			0101 0101
086	56			0101 0110
087	57			0101 0111

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
088	58			0101 1000
089	59			0101 1001
090	5A	!		0101 1010
091	5B	$		0101 1011
092	5C	*		0101 1100
093	5D	)		0101 1101
094	5E	;		0101 1110
095	5F	¬		0110 1111
096	60	-		0110 0000
097	61	/		0110 0001
098	62			0110 0010
099	63			0110 0011
100	64			0110 0100
101	65			0110 0101
102	66			0110 0110
103	67			0110 0111
104	68			0110 1000
105	69			0110 1001
106	6A	|		0110 1010
107	6B	,		0110 1011
108	6C	%		0110 1100
109	6D	_		0110 1101
110	6E	>		0110 1110
111	6F	?		0110 1111
112	70			0111 0000
113	71			0111 0001
114	72			0111 0010
115	73			0111 0011
116	74			0111 0100
117	75			0111 0101
118	76			0111 0110
119	77			0111 0111

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
120	78			0111 1000
121	79			0111 1001
122	7A	:		0111 1010
123	7B	#		0111 1011
124	7C	@		0111 1100
125	7D	’		0111 1101
126	7E	=		0111 1110
127	7F	"		0111 1111
128	80			1000 0000
129	81	a		1000 0001
130	82	b		1000 0010
131	83	c		1000 0011
132	84	d		1000 0100
133	85	e		1000 0101
134	86	f		1000 0110
135	87	g		1000 0111
136	88	h		1000 1000
137	89	i		1000 1001
138	8A			1000 1010
139	8B			1000 1011
140	8C			1000 1100
141	8D			1000 1101
142	8E			1000 1110
143	8F			1000 1111
144	90			1001 0000
145	91	j		1001 0001
146	92	k		1001 0010
147	93	l		1001 0011
148	94	m		1001 0100
149	95	n		1001 0101
150	96	o		1001 0110
151	97	p		1001 0111

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
152	98	q		1001 1000
153	99	r		1001 1001
154	9A			1001 1010
155	9B			1001 1011
156	9C			1001 1100
157	9D			1001 1101
158	9E			1001 1110
159	9F			1001 1111
160	A0			1010 0000
161	A1	~		1010 0001
162	A2	s		1010 0010
163	A3	t		1010 0011
164	A4	u		1010 0100
165	A5	v		1010 0101
166	A6	w		1010 0110
167	A7	x		1010 0111
168	A8	y		1010 1000
169	A9	z		1010 1001
170	AA			1010 1010
171	AB			1010 1011
172	AC			1010 1100
173	AD			1010 1101
174	AE			1010 1110
175	AF			1010 1111
176	B0			1011 0000
177	B1			1011 0001
178	B2			1011 0010
179	B3			1011 0011
180	B4			1011 0100
181	B5			1011 0101
182	B6			1011 0110

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
183	B7			1011 0111
184	B8			1011 1000
185	B9			1011 1001
186	BA			1011 1010
187	BB			1011 1011
188	BC			1011 1100
189	BD			1011 1101
190	BE			1011 1110
191	BF			1011 1111
192	C0	{		1100 0000
193	C1	A		1100 0001
194	C2	B		1100 0010
195	C3	C		1100 0011
196	C4	D		1100 0100
197	C5	E		1100 0101
198	C6	F		1100 0110
199	C7	G		1100 0111
200	C8	H		1100 1000
201	C9	I		1100 1001
202	CA	SHY		1100 1010
203	CB			1100 1011
204	CC			1100 1100
205	CD			1100 1101
206	CE			1100 1110
207	CF			1100 1111
208	D0	}		1101 0000
209	D1	J		1101 0001
210	D2	K		1101 0010
211	D3	L		1101 0011
212	D4	M		1101 0100
213	D5	N		1101 0101

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
214	D6	O		1101 0110
215	D7	P		1101 0111
216	D8	Q		1101 1000
217	D9	R		1101 1001
218	DA			1101 1010
219	DB			1101 1011
220	DC			1101 1100
221	DD			1101 1101
222	DE			1101 1110
223	DF			1101 1111
224	E0	\		1110 0000
225	E1			1110 0001
226	E2	S		1110 0010
227	E3	T		1110 0011
228	E4	U		1110 0100
229	E5	V		1110 0101
230	E6	W		1110 0110
231	E7	X		1110 0111
232	E8	Y		1110 1000
233	E9	Z		1110 1001
234	EA			1110 1010
235	EB			1110 1011
236	EC			1110 1100
237	ED			1110 1101
238	EE			1110 1110
239	EF			1110 1111
240	F0	0		1111 0000
241	F1	1		1111 0001
242	F2	2		1111 0010
243	F3	3		1111 0011
244	F4	4		1111 0100
245	F5	5		1111 0101

Decimal Value	Hex Value	EBCDIC Value*	Description	Binary
246	F6	6		1111 0110
247	F7	7		1111 0111
248	F8	8		1111 1000
249	F9	9		1111 1001
250	FA			1111 1010
251	FB			1111 1011
252	FC			1111 1100
253	FD			1111 1101
254	FE			1111 1110
255	FF	EO	Eight Ones	1111 1111

* In the IBM-DOS Character Set, the nonprinting characters may be displayed as figures, for example, (x03) ETX is shown as a heart, and (x0D) CR is shown as a musical note.

Appendix H
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.

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and Service Marks

The following, which appear in this book or other MQSeries Integrator books, are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

MQSeries
AIX
DB2
IBM

NEONFormatter and NEONRules are trademarks of New Era of Networks, Inc. in the United States and / or other countries.

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

UNIX is a registered trademark in the United States and / or other countries licensed exclusively through X / Open Company Limited.

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

Index

G

H

I

J

L

M

N

O

P

R

S

T

U

V

Y

Sending your comments to IBM

MQSeries Integrator

Programming Reference for NEONFormatter

SC34-5507-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book only and the way in which the information is presented.

To request additional publications or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

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

Electronically, use the appropriate network ID:

- IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
- IBMLink: HURSLEY(IDRCF)
- Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic number to which your comment applies
- Your name / address / telephone number / fax number / network ID

SC34-5507-00