



MQSeries® Integrator

# **Programming Reference for NEONFormatter**

Version 1.1

**Note: Before using this information, and the product it supports, be sure to read the general information under *Notices* on page 539.**

**Second edition (June 1999)**

This edition applies to IBM® MQSeries Integrator, Version 1.1 and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

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

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

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

© Copyright New Era of Networks, Inc., 1998, 1999. All rights reserved.

© Copyright International Business Machines Corporation, 1999. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

# Contents

---

<b>Chapter 1: Introduction .....</b>	<b>1</b>
Product Documentation Set .....	1
Summary of Changes .....	2
Supported Platforms and Compilers .....	3
Year 2000 Readiness Disclosure.....	4
<b>Chapter 2: Formatter Overview.....</b>	<b>5</b>
Format Structure .....	7
Input Format Structure .....	8
Compound Input Formats .....	19
Output Format Structure.....	20
Output Operations .....	29
Compound Output Formats .....	35
Converting Data Types.....	36
Automatic Format Conversion .....	58
Using the Formatter Engine .....	59
API and Header Files .....	63
Libraries.....	76
<b>Chapter 3: Formatter APIs .....</b>	<b>77</b>
Formatter Member Functions .....	77
OutMsg Class Member Functions .....	113
OutMsgGroup Class Member Functions .....	115
ParsedField Class Member Functions.....	119
ParsedMessage Class Member Functions .....	125
User Exit API Functions.....	131
User Exit Lookup Interface .....	131
User Exit Function Developer Interface .....	135
User Callback API Functions .....	157
User Callback API Structures .....	159
User Callback Class Definition.....	168
User Callback Lookup Interface .....	192
User-Defined Type Input Field Validation .....	197
Description .....	197

User Callback API Functions .....	199
Rough Sketch of Required Code .....	200
Using the Thread Safe Release of the Formatter Engine.....	202
Thread Safety Feature Impacts to Formatter APIs .....	203
Data Cleanup.....	205
Linking .....	205
Formatter Error Handling .....	206
Formatter Error Messages .....	208
Parsing Errors.....	226

## **Chapter 4: Formatter Management APIs .....229**

General Formatter Management APIs.....	231
Field Management APIs .....	234
Field Management API Structure.....	234
Field Management APIs .....	236
Output Format Control Management APIs.....	242
Output Control Management API Structures .....	243
Output Control Management APIs.....	265
Output Operation Controls.....	274
Literal Management APIs.....	371
Literal Management API Structure.....	372
Literal Management APIs.....	373
User-Defined Data Type Management APIs .....	382
User-Defined Data Type Management API Structures.....	383
User-Defined Data Type Management APIs.....	385
Parse Control Management APIs .....	398
Parse Control Management API Structures.....	399
Parse Control Management APIs.....	404
Format Management APIs.....	417
Format Management API Structures .....	418
Format Management APIs .....	425
Format Management API Error Handling.....	452
Format Management Error Messages.....	454

<b>Appendix A: Sample Programs .....</b>	<b>457</b>
Using Formatter APIs to Reformat a Message: msgtest.cpp .....	457
GetValidationCallbacks Function: getval.cpp #1 .....	463
GetValidationCallbacks Function: getval.cpp #2 .....	463
Traversing a Parsed Message:	
apitest.cpp .....	468
Using Format Management APIs:	
fmgr.cpp .....	475
<b>Appendix B: Access Mode Examples .....</b>	<b>493</b>
<b>Appendix C: Code Example for Substitute</b>	
<b>Controls.....</b>	<b>507</b>
<b>Appendix D: Data Type Descriptions .....</b>	<b>509</b>
Notes for Data Conversion.....	517
<b>Appendix E: ASCII Extended Character Set ..</b>	<b>519</b>
<b>Appendix F: EBCDIC Character Set .....</b>	<b>525</b>
<b>Appendix G: Notices .....</b>	<b>539</b>
Trademarks and Service Marks .....	541



---

## Chapter 1

# Introduction

---

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

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

---

## Summary of Changes

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

The *Formatter Management APIs* on page 229 has been updated to include new functionality for Update APIs and Delete APIs in the following areas:

- Field Management
- Output Control Management
- Output Operation Controls
  - Substitute Controls
  - User Exit Controls
  - Math Expression Controls
  - Pre/PostFix Controls
  - Default Controls
  - Length Controls
  - Substring Controls
  - Trim Controls
  - Collection Controls
- Literal Management
- User-Defined Data Type Management



- Parse Control Management
- Format Management

## Supported Platforms and Compilers

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

---

## Year 2000 Readiness Disclosure

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

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

IBM reserves the right to update the information shown here. For the latest information regarding levels of supported software, refer to:

<http://www.software.ibm.com/ts/mqseries/platforms/supported.html>

For the latest IBM statement regarding Year 2000 readiness, refer to:

<http://www.ibm.com/ibm/year2000/>

---

## Chapter 2

# Formatter Overview

---

NEONFormatter has two main functions: parsing and reformatting.

- Parse separates an input message into individual fields.
- Reformat transforms 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 NEONFormatter graphical user interface (GUI) tool or the NEONFormatter Management API functions.

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

Consistency Checker is another NEONFormatter tool. This 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 the data. For more information on the Consistency Checker, see the ***System Management*** guide.

NNFie is a NEONFormatter command line tool that allows you to export format definitions from a database to an export file, and to import from the export file into a database. 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 must 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.

## Format Components

Component	Description
Literals	See <i>Literals</i> in Volume II.
Input Controls	See <i>Input Controls</i> on page 8.
Output Controls	See <i>Output Controls</i> on page 22.
Output Operations Output Operation Collections	See <i>Output Operations</i> on page 29. See <i>Output Operation Collections</i> on page 33.
Fields	See <i>Input Format Structure</i> on page 8.
Flat Input Formats Compound Input Formats Flat Output Formats Compound Output Formats	See <i>Input Format Structure</i> on page 8.
User-Defined Types	See <i>User-Defined Data Type Management APIs</i> .

## 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 your message 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 to parse or format them (input controls 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.

### Input Format Component Fields

When you insert a field into a flat input format, he 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.

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

The data can have a tag, which is an alternate identifier for a field. Formatter can retrieve the data for a field using its name or its tag.

Field data can have a length value, which specifies the length of the field data in bytes.

## Input Control Types

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.
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.

Type	Description
Regular Expression	Field data in the input message conforms to a string pattern defined by a regular expression.

## ***Regular Expression Syntax***

Regular expression (RE) input controls are strings defining 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 will not get a match for the field because no space follows 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]`.

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 can 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.

- An RE enclosed within parentheses (`(“,”)`) is an RE that matches whatever the non-parenthetical RE matches. There is no difference between parenthetical and non-parenthetical 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 (`|`) symbol indicates or. For example, `(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 may 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 will continue 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 must specify information to parse the tag and length portions of the field.

## Data Portion of an Input Control

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

Type	Description
Type	<p>Specifies the data type of the field. Formatter supports a set of native data types and can interpret data values and convert data values from one native type to another. See <i>Data Type Descriptions</i> on page 509.</p> <p>Data types are either fixed length data: (Endian types and date/time types) or variable length. If the length of the data is variable, Formatter must determine the length of the data. To specify data length, choose one of the following:</p> <ol style="list-style-type: none"> <li>1) Use 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.</li> <li>2) Use a termination for the data (see <i>Termination</i> in this table).</li> </ol>
Base Type	<p>This applies only to date/time data types. You must 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 <i>Data Type Descriptions</i> on page 509.</p>
Format	<p>This applies only to date/time data types. You must specify the date/time format string to which the data must conform. See <i>Data Type Descriptions</i> on page 509.</p>
Year cutoff	<p>The year cutoff value indicates how to convert a two-digit year to a four-digit year. For more information, see <i>Specifying a Year Cutoff Value</i> on page 15.</p>

<b>Type</b>	<b>Description</b>
Termination	<p>This specifies how to find the end of the data. The choices are:</p> <ul style="list-style-type: none"> <li>■ Delimiter (data is terminated by a literal delimiter)</li> <li>■ Exact length (data has a fixed length in bytes)</li> <li>■ Minimum Length + Delimiter (data is of at least a minimum length in bytes followed by a literal delimiter)</li> <li>■ Minimum Length + White Space (data is of at least a minimum length in bytes followed by a single white space)</li> <li>■ Not Applicable (data ends at the end of the message, or just before the format terminator of the flat format that contains the field)</li> <li>■ White Space Delimited (data is terminated by a single white space)</li> </ul>
Length	<p>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.</p>
Delimiter	<p>Specifies the delimiter if the termination of the data is Delimiter or Minimum Length + Delimiter.</p>
Decimal Loc	<p>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 the value must not be greater 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.</p>
Literal	<p>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). You must choose an existing literal.</p>

## Specifying a Year Cutoff Value

The internal application functions of MQSeries Integrator use date-time information for archiving, time stamping, logging, etc. These functions are made using the standard C++ class libraries and use 4-digit notation or Universal Coordinated Time (UCT) for time stamps; thus, these functions are Y2K Compliant, given that the underlying hardware is compliant. The function and libraries used with MQSeries Integrator include the logic for correct processing of leap year before, during, and after 1/1/2000.

Within the message handling and processing functionality, date information can be embedded and reformatted. MQSeries Integrator provides date and date-time comparison, parsing, and reformatting functions for this. Date/date-time parsing and reformatting 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 4-character years and are Y2K compliant. MQSeries Integrator also supports 2-character years as custom field definitions. These custom formats are Y2K compliant if used as described in the following paragraphs.

NEON's products provide the facility to resolve the century ambiguity through a Year Cutoff Number for Input field data definitions (Input Controls) using Custom Date and Time and Custom Date definitions, which include 2-digit year notation (such as MM/DD/Y HH:MM:SS or MM/DD/YY). You must specify a Year Cutoff that must be from 0 to 100 (inclusive). Using this cutoff number, NEONFormatter converts a 2-digit year (YY) to a 4-digit year (YYYY).

The Year Cutoff algorithm is as follows:

- year value  $\geq$  cutoff value -> 19XX
- year value < cutoff value -> 20XX

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

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

- If you specify the Year Cutoff number as 50, then all 2-digit input dates from 50 to 99 are designated as 1950 to 1999 output dates; all 2-digit input dates from 00 to 49 are designated as 2000 to 2049 output dates.

- If you specify the cutoff date as 75, then all 2-digit input dates from 75 to 99 are designated as 1975 to 1999 output dates; all 2-digit input dates from 00 to 74 are designated as 2000 to 2074 output dates.

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

### ***NEONFormatter API***

For an input control that specifies a data type of custom date or date-time with a 2-digit year format string, you must specify a Year Cutoff value (regardless of the output date/date-time string). `NEONFormatter` uses this value to convert the 2-digit year date value to a 4-digit year date value. When `NEONFormatter` does the conversion, it compares the year value of the input data to the specified year Cutoff value and assigns the century designation as required. For example, based on the comparison, `NEONFormatter` converts the year value "XX" to "20XX" (21st century year) or "19XX" (20th century year) as appropriate.

### ***NEONFormatter GUI***

In the `NEONFormatter` GUI, you must specify a Year Cutoff value for all input with a 2-digit year date string. The GUI provides a field for this and defaults the field to a Year Cutoff of '101', which is an invalid number. You are required to enter a valid Year Cutoff value to continue.

## **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.

<b>Type</b>	<b>Description</b>
Type	Specifies the data type of the tag portion of the field. Formatter supports a set of native data types and can 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.
Tag Value	You can specify an existing literal, or NONE. If you specify an existing literal, Formatter attempts to parse the exact literal value specified. If you specify NONE, the tag can take on different values, and Formatter determines the value by parsing it.
Termination	<p>This applies only to a Tag Value = NONE, where Formatter may have a variable tag value to parse. You can specify the following terminations:</p> <ul style="list-style-type: none"> <li>Delimiter (tag is terminated by a literal delimiter)</li> <li>Exact length (tag has a fixed length in bytes)</li> <li>Minimum Length + Delimiter (tag is of at least a minimum length in bytes followed by a literal delimiter)</li> <li>Minimum Length + White Space (tag is of at least a minimum length in bytes followed by a single white space)</li> <li>Not Applicable (no termination specified because data type of tag is of fixed length)</li> <li>White Space Delimited (tag is terminated by a single white space).</li> </ul>
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.
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 has four fields that 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 and can interpret and convert data values from one native type to another. A length can have only a data type that contains an integer number: Endian types, IBM types, EBCDIC, Numeric, and String.
Termination	If the data type of the length value indicates a variable length value, then you 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 can be repeating. Types of compound input formats are described in the following table.

<b>Type</b>	<b>Description</b>
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	<p>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 can contain component formats B, C, and D. A message of format A can actually be of variation B, C, or D.</p> <p>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 be careful to construct an alternative format so that a general alternative does not supersede a more specific alternative.</p> <p>If none of the alternatives applies, and the format is a mandatory component of the message, the parse fails.</p>

## Inserting a Component Format into a Compound Input Format

You can insert either a flat input format 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 have 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, choose 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 that 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 can 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 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.
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. (zero indicates the first instance.)
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, for example, search by name or tag or use a literal, how to convert data types, how to format the data (formatting operations are defined by subordinate output operations associated with the output control), and whether 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 math 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 type uses access modes to determine which instance of the input field to access, and uses input field mapping if you want a different named input field to map to the output field.
Data Field Tag Search	Maps the output field to an input field by tag value. Access modes and input field mapping are not used for this 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	<p>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.</p> <p><b>Note:</b> It is recommended that you use the Math expression operation rather than Calculated Field, which will become obsolete in a future version of Formatter.</p>
Conditional Field	Marks field as output only if the Existence check field exists.

Type	Description
Rules Field	<p>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.</p> <p>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 will be formatted according to the other settings in the Rules Field output control. Otherwise, the data will be formatted according to the output control specified by the rule that hits.</p> <p>If multiple conflicting rules hit (multiple output controls are returned by Rules for the same output field), Formatter generates an error.</p> <p>See <i>Conditional Branching</i> in the MQSeries Integrator User Guide for details.</p> <p><b>Note:</b> 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.</p> <p><b>Note:</b> The Output Control type determines which fields require information. Fields containing Not Applicable or NONE do not require a value.</p>

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	<p>Mark field as left operand. (Used for Calculated Fields)</p> <p><b>Note:</b> Use the Math expression operation rather than Left Operand Field, which will become obsolete in a future release of Formatter.</p>

<b>Type</b>	<b>Description</b>
Right Operand Field	Mark field as a right operand. (Used for Calculated Fields)  <b>Note:</b> Use the Math expression operation rather than Right Operand Field, which will become obsolete in a future release of Formatter.
Existence Check Field	Mark field as an existence check field. (Used for Conditional Fields)
Input Field Exists	The format in which the associated field appears should be output only if the associated input field exists.
Input Field Value =	The format in which the associated field appears 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.

### Access Mode Definitions

<b>Access Mode</b>	<b>Description</b>
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 the same as an access sibling instance.
Access with Increment	A field with this access mode is the controlling field for the repeating component.

<b>Access Mode</b>	<b>Description</b>
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. Any other field in the repeating component responds the same as access sibling instance or normal access.
Access nth instance of field	Access the nth 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 the previous access. The first access gets 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 Control Properties

An output control usually specifies formatting operations, for example, justification, trimming, and default values, 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 either a collection of output operations of various types or can a single output operation.

Property	Description
Optional/Not optional	<p>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.</p> <p>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 will be used.</p> <p>By default, the output control is mandatory. Mandatory means if the output field value has zero length, for example, no mapped input field or input field value has zero length, no default, no math expression, or no user exit, the entire formatting operation fails.</p>
Data type, base data type, and format	<p>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.</p> <p>If the specified output data type is a date/time data type, you must specify a Base Data Type (underlying native data type of String, EBCDIC, or Numeric) and a Format date/time format string.</p>

<b>Property</b>	<b>Description</b>
Length type, tag type, and tag before length	<p>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.</p> <p>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.</p> <p>Normally, the output field components are output in the following order: length, followed by tag, followed by data. If you choose Tag before length, then the output field components are output in the following order: tag followed by length followed by data.</p>
Input Tag Value	<p>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 specified by Tag Value (a literal).</p>
Calculation	<p>This field applies only if the output control type is Calculated Field. The available calculation choices are Add, Subtract, Multiply, and Divide.</p> <p>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. Use the Math Expression output operation instead.</p>
Field Value	<p>This field applies only if the output control type is either Literal or Input Field Value =. For Literal, the Field Value field is set to the particular literal that should be output for the field. For Input Field Value =, the Field Value field is set to the value, allowing a comparison with the input field value.</p>

# Output Operations

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

By using output operation collections, you can collect operations to perform sequentially. For example, you can 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.

## Output Operation Types

Output operation types are described in the following table.

---

### 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	Affects 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.
Default	Provides 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. <b>Note:</b> A default operation negates all operations that precede it, because it generates a new value for the field.

<b>Output Operation Type</b>	<b>Description</b>
Prefix/Suffix	<p>Specifies 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 symbol (&lt;) before and a greater-than symbol (&gt;) after the output data. This would require a Prefix of &lt; and a suffix of &gt;. Any defined literal can be used as a prefix or suffix.</p> <p>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 is not present in the input message, or it has zero length.</p>
Justify	<p>Justify operations justify field data to the left, center, or right within the length of the field. A justify operation has no effect on the data, unless the field width specified is greater than the length of data. This field width adjustment is accomplished by including a Length operation after the Justify operation in the collection.</p>
Length	<p>Length operations adjust the width of an output field. For example, if you define a length of 12, and the data is longer, the data is truncated on the right. If it is shorter, you can define a pad character to fill the empty space. If no pad character is specified, a space character is used.</p> <p>If there is no preceding Justify operation, numeric data is right justified and padded on the left. Other data is left justified and padded on the right.</p> <p>If there is a preceding Justify operation, data is justified and padded according to the parameters of the Justify operation.</p>
Math Expression	<p>Outputs a value resulting from an arithmetic expression. The expression can be built using arithmetic operators, constants, and input field values.</p> <p><b>Note:</b> A math expression operation negates all operations that precede it because it generates a new value for the field.</p>

<b>Output Operation Type</b>	<b>Description</b>
Substitute	<p>Defines 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 will be left unchanged.</p> <p><b>Note:</b> A substitute operation negates all operations that precede it, because it generates a new value for the field.</p>
Substring	<p>Extracts 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.</p> <p>Only String, Numeric, EBCDIC, and Binary data types can be used for substrings.</p> <p><b>Note:</b> If you extract a substring longer than the data, you can 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.</p>
Trim	<p>Removes a defined trim character to the right or left of the output data.</p>
User Exit	<p>Computes the value of an output field. If a function you must perform is outside what the Formatter can currently do, you can write a C function to perform the task. See the <b><i>System Management</i></b> guide for details.</p> <p><b>Note:</b> A user exit operation negates all operations that precede it because it generates a new value for the field.</p>

## Building Math Expressions

Using Math 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 containing spaces or underscores must be surrounded by either single or double quotes. A field with the name Field\_1 could be used as "Field\_1" or 'Field\_1', but not or 'Field\_1".

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

$\text{InF1} + \text{InF2} * -\text{InF3}$

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

Other expression examples include:

- $\text{InF1} + -\text{InF2}$
- $\text{InF1} * 8$
- $\text{InF1} * 9.3$
- $\text{InF1} * -8$
- $\text{InF1} * -9.3$
- $(\text{InF1} + \text{InF2}) * 3 / \text{InF3}$
- $(\text{InF1} * (\text{InF2} + \text{InF3})) * 4$

## Output Operation Collections

Output operation collections enable you to group and sequence a set of output operations 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 must sequence LEFT\_JUSTIFY, LENGTH, RIGHT\_TRIM, and SUBSTRING.

### Building a Collection

Some 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:

<b>Output Operations</b>	<b>Description</b>
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 a succeeding operation any number of times:

<b>Output Operations</b>	<b>Description</b>
Case	Converts data to uppercase or lowercase.
Prefix/Suffix	Prepends a prefix or appends a suffix to the data.
[Justify] Length	<p>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 together and separately:</p> <p><b>Rule 1:</b> A Justify operation without a succeeding Length operation has no effect.</p> <p><b>Rule 2:</b> 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.</p> <p><b>Rule 3:</b> 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.</p> <p><b>Rule 4:</b> A Length operation with a succeeding Justify operation operates just like Rule 2. (Justify has no effect; Length works as though there were no corresponding Justify operation.)</p> <p><b>Rule 5:</b> 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.</p>
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 can be repeating.

### Types of Compound Output Formats

There are two types of compound output formats:

<b>Compound Output Formats</b>	<b>Description</b>
Ordinal	The component formats of the compound output format always appear in the order in which they are defined.
Alternative	<p>Alternative formats are special compound formats where only one format in a set of alternatives applies 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.</p> <p>Exactly one of the alternatives must apply, or the entire alternative format does not apply.</p> <p>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.</p>

## Inserting a Component Format Into a Compound Output Format

You can 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:

<b>Compound Output Format</b>	<b>Description</b>
Optional/not optional	If the component is optional, the component format does not have 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, choose 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.

## Converting Data Types

When `NEONFormatter` 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 and is then converted to its final output data type.

## Value Ranges and Intermediate Representation

The following table describes the valid values allowed for data of each type and intermediate String representation.

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

<b>Data Type</b>	<b>Source Data Value Range</b>	<b>Intermediate String Representation</b>
EBCDIC	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) can contain the hexadecimal value 0 to 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.
IBM Signed Packed Integer	Maximum 16-byte value. Each nibble (except for the last nibble) can 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".

<b>Data Type</b>	<b>Source Data Value Range</b>	<b>Intermediate String Representation</b>
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 2-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 2-byte integer in the range 0 to 65535 (0 to $(2^{16}) - 1$ )	String that represents the integer value ( $\geq 0$ ).

<b>Data Type</b>	<b>Source Data Value Range</b>	<b>Intermediate String Representation</b>
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. For example, 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 - can 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. For example, 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.

<b>Data Type</b>	<b>Source Data Value Range</b>	<b>Intermediate String Representation</b>
Date and Time	A 14-byte numeric string. Each character is in the range 0 to 9 that represents a valid date in the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS. The base data type is 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.
Time	A 6-byte numeric string. Each character is in the range 0 to 9 that 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 numeric string. Each character is in the range 0 to 9 that represents a valid date in the international ISO-8601:1988 standard datetime notation: YYYYMNDD. The base data type is 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.

<b>Data Type</b>	<b>Source Data Value Range</b>	<b>Intermediate String Representation</b>
Custom Date and Time	<p>A string in one of the custom date/time formats. Users cannot currently specify date/time formats. The base data type is String, EBCDIC, or Numeric. The following is a list of the custom formats supplied with Formatter:</p> <p>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/YYHH:MM:SS  PMDD/MN/YY  HH:MM:SS PMMN/  DD/YY HH:MM:SSDD/  MN/  YYHH:MM:SSHH:MM  PMHH:MMHH:MM:SS  PMHH:MM:SSMNDY  YMNDDYYYYDDMON  YYDDMONYYYYMON  YYMONYYYYMONDD  YYYYMNDDYYHHMM  MNDDYYHHMMSSHH  MMSS</p>	<p>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.</p>



## Data Type Conversion Constraints

There are some pairs of data type conversions that are not sensible. For example, converting the string *good morning* to a number. This section discusses the constraints that exist for data conversion pairs.

### ***Not Applicable***

A data type of Not applicable means that you do 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 on which that Formatter executes.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	The data remains unchanged between the input message and the output message.
String	The data remains unchanged between the input message and the output message.
Numeric	Valid only if each character of the field value is 0 to 9. In this case, the data remains unchanged between the input message and the output message.
Binary Data	Valid only if the string contains only the characters 0 to 9 and A to F, and the string contains an even number of characters. This is because the string is interpreted as the string encoded hexadecimal form of a binary field. For example, the string0123AC 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.

<b>Output Data Type</b>	<b>Constraints</b>
EBCDIC Data	This is a valid conversion. Values in the ASCII character set that do not 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 greater than or equal to zero 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 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 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 an integer and has a value that is within the range allowed for 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 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.

<b>Output Data Type</b>	<b>Constraints</b>
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: YYYYMMDDHHMMSS.

## ***Numeric***

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

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	The data remains unchanged between the input message and the output message.
String	The data remains unchanged between the input message and the output message.
Numeric	The data remains unchanged between the input message and the output message.
Binary Data	Valid only if the numeric string contains an even number of characters. This 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 within the range allowed for the IBM types.

<b>Output Data Type</b>	<b>Constraints</b>
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 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 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 within the range allowed for 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 a number with a value 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: YYYYMMDDHHMMSS.

## ***Binary***

Binary indicates a sequence of binary characters.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
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. 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 to 9.
Binary Data	The data remains unchanged between the input message and the output message.
EBCDIC Data	Same behavior as String, except each character is an EBCDIC encoded character instead of 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 to 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 to 9, and the number is within the range allowed for the various Endian types.
Decimal, International Decimal, U.S.	This is a valid conversion only if the string-encoded hexadecimal form of the binary string contains only the numbers 0 to 9.

<b>Output Data Type</b>	<b>Constraints</b>
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***

A string of data encoded using the Extended Binary Coded Decimal Interchange Code (EBCDIC) used on larger IBM machines.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
String	This is a valid conversion (each EBCDIC character is converted to its ASCII equivalent). Values in the EBCDIC character set that do not 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 to 9.
Binary Data	Valid only if the string contains the EBCDIC characters 0 to 9 and A to F, and the string contains an even number of characters. This is because the string is interpreted as the string-encoded hexadecimal form of a binary field. For example, the string 0123AC is converted to 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 remains unchanged between the input message and the output message.

<b>Output Data Type</b>	<b>Constraints</b>
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 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 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.
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, and IBM Signed Zoned.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
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 greater than or equal to zero.
Binary Data	Valid only if the number is an integer greater than or equal to zero 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.



<b>Output Data Type</b>	<b>Constraints</b>
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.
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.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
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 greater than or equal to zero.
Binary Data	Valid only if the value of the number is an integer with a value greater than or equal to zero 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 greater than or equal to zero.
Decimal, International Decimal, U.S.	This is a valid conversion.

<b>Output Data Type</b>	<b>Constraints</b>
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.
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 $\geq 0$ .
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.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
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 greater than or equal to zero.
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 greater than or equal to zero in the correct range are valid. For unsigned numbers being converted to signed numbers, only values in the correct range are valid.

<b>Output Data Type</b>	<b>Constraints</b>
Endian 2 types	<p>Valid for conversions from signed to signed only if the Endian 4 number is in the range allowed for signed Endian 2 types.</p> <p>Valid for conversions from unsigned to unsigned only if the Endian 4 number is in the range allowed for unsigned Endian 2 types.</p> <p>Valid for conversions from signed to unsigned only if the Endian 4 number in the range allowed for unsigned Endian 2 types.</p> <p>Valid for conversions from unsigned to signed only if the Endian 4 number is in the range allowed for signed Endian 2 types.</p>
Endian 4 types	<p>Valid for conversions from signed to signed.</p> <p>Valid for conversions from unsigned to unsigned.</p> <p>Valid for conversions from unsigned to signed.</p> <p>Valid for conversions from signed to unsigned, only if the number is greater than or equal to zero.</p>
Date and Time Custom Date and Time Time Date	<p>Valid only if the number converts to a string in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.</p>

## ***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 - can 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.

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	The data remains unchanged between the input message and the output message.
String	The data remains 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 remains 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.

<b>Output Data Type</b>	<b>Constraints</b>
Decimal, International Decimal, U.S.	This is a valid conversion. If going between International and US, . is changed to , and , is changed to ..

## ***Date and Time***

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

<b>Output Data Type</b>	<b>Constraints</b>
Not Applicable	Same behavior as String.
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 default format, so there is no change in data. Data in Custom Date and Time format is changed as described.
Numeric	Same behavior as String.
Binary Data	This is a valid conversion. For example, the date/time value 19560601190000 (June 1, 1956 at 7:00 PM) is 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 String, 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 integer format. 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.

<b>Output Data Type</b>	<b>Constraints</b>
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 can be converted between each other, and to Time (drops the date portion) or to Date (drops the time portion).
Time	A Time can 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 can 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

NEONFormatter contains higher-level APIs that can request Formatter 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) need to format the new message.

The message path through M&Q and Formatter is the same as the message path through M&Q alone. The NNHPutMsg() command uses Formatter by means of a function call, as does the sending process, receiving process, or NNHGetMsg() command. Sending and receiving applications remain uninvolved with transaction details.



---

## Using the Formatter Engine

The `NEONFormatter` engine can be compared to 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 sent 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.

If 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.

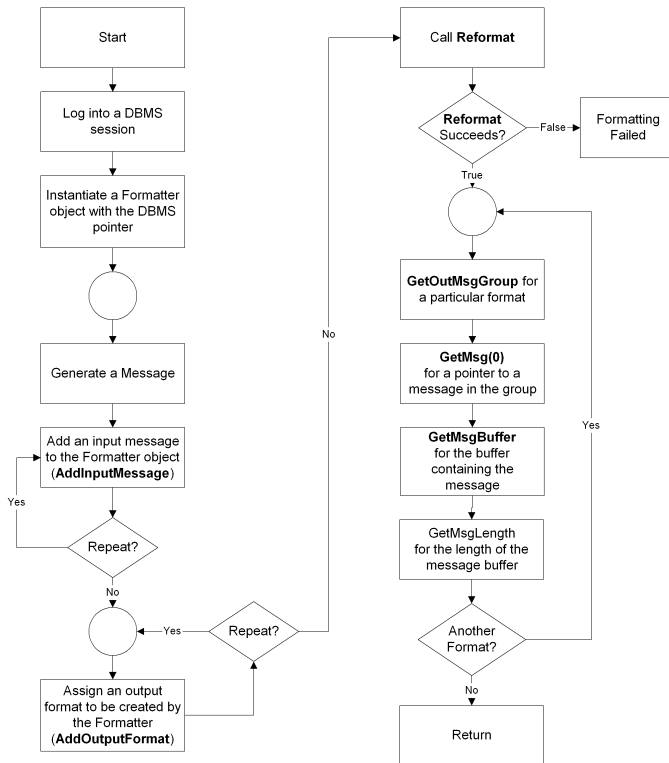
[Formatter formats one output message for each output format specified using AddOutputFormat().]

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 is produced. For example, if one input message is added using `AddInputMessage()` and three output formats are specified using `AddOutputFormat()`, `Formatter` generates three formatted messages. The input message is formatted to fit each of the three output formats.

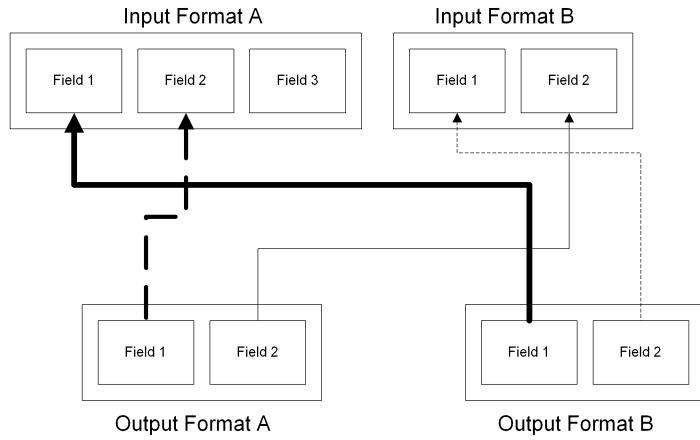
Each field of an input message (parsed according to the specified input format) you want to appear in the output message must be mapped to a field in an output format. This mapping can be implicit, based on the name of each field in the output format, or it can be explicit.

Field mapping provides flexibility, enabling the combination of different input message/output format field-level mappings. The following examples illustrate some ways mappings can be combined. 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 can 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 will be 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.)

Refer to the following diagram for an illustration of this example.



*Forming Multiple Output Messages from Multiple Input Messages*

# API and Header Files

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

## Header Files

<b>Object Class</b>	<b>Header File</b>	<b>Description</b>
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	nfmgr.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
ParsedMessage*	GetParsedInMsg	(int index)
void	SetUserTypeValidationOn	none
void	SetUserTypeValidationOff	none
int	UserTypeValidationIsOn	none
int	GetErrorCode	none
char*	GetErrorMessage	none

**OutMsg Class Functions**

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
char*	GetMsgBuffer	none
int	GetMsgLength	none

**OutMsgGroup Class Functions**

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
OutMsg*	GetMsg	(int index)
int	GetMsgCount	none

**ParsedField Class Functions**

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

**ParsedMessage Class Functions**

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

## Formatter Management API Functions

Return Type	Function	Arguments
NNFMgr *	NNFMgrInit	(DbmsSession *session)
void	NNFMgrClose	(NNFMgr *pNNFMgr)
N/A	NNF_CLEAR	(_p)
const short	NNFMgrCreateField	(NNFMgr *pNNFMgr, const NNFMgrFieldInfo * const pFieldInfo)
const short	NNFMgrGetFirstField	(NNFMgr *pNNFMgr, NNFMgr * const pFieldInfo)
const short	NNFMgrGetNextField	(NNFMgr *pNNFMgr, NNFMgr * const pFieldInfo)
const short	NNFMgrUpdateField	(NNFMgr *pNNFMgr, const char * const fldName, const NNFMgrFieldInfo * const pInfo)
const short	NNFMgrDeleteField	(NNFMgr *pNNFMgr, const char * const fldName)
const short	NNFMgrCreateOut MstrCntl	(NNFMgrOutMstrCntlInfo* const pInfo)
const short	NNFMgrGetOutMstr Cntl	(NNGetOp OpCode, NNFMgrOutMstrCntlInfo*)
const short	NNFMgrUpdateOut MstrCntl	(NNFMgr *pNNFMgr, const char * const cntlName, const NNFOutputControlInfo * const pInfo)
const short	NNFMgrDeleteOut MstrCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreate SubstituteCntl	(NNFMgr *pNNFMgr, NNFMgrSubstituteCntlInfo* const pInfo)



<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgAppendEntry ToSubstituteCntl	(NNFMgr *pNNFMgr, const NFMgrSubstituteCntlInfo* const pInfo)
const short	NNFMgrGet SubstituteCntl	(NNFMgr *pNNFMgr, NNGetOp OpCode, NNFMgrSubstituteCntlInfo* const pInfo, int* const NumRemainingEntries)
const short	NNFMgrGetNextEntry FromSubstituteCntl	(NNFMgr *pNNFMgr, NNFMgr SubstituteCntlInfo* const)
const short	NNFMgrUpdate SubstituteCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrSubstituteCntlInfo* const pInfo)
const short	NNFMgrDelete SubstituteCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreateUser ExitCntl	(NNFMgr *pNNFMgr, const NNFMgrUser ExitCntlInfo* const pInfo)
const short	NNFMgrGetUser ExitCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrUserExitCntlInfo * const pInfo)
const short	NNFMgrUpdateUser ExitCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgruserExitCntlInfo * const pInfo)
const short	NNFMgrDeleteUser ExitCntl	(NNFMgr *pNNFMgr, const char * const cntlName)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrCreateMathExpCntl	(NNFMgr *pNNFMgr, NFMgrMathExpCntlInfo* const pInfo)
const short	NNFMgrGetMathExpCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrMathExpCntlInfo * const pInfo)
const short	NNFMgrAppendSegToMathExpCntl	(NNFMgr * pNNFMgr, const char* const CntlName, const NNFMgrMathExpCntl Segment Info * const pInfo)
const short	NNFMgrGetSegFromMathExpCntl	(NNFMgr * pNNFMgr, const char* const CntlName, NNGetOp OpCode, const NNFMgrMathExpCntl Segment Info * const pInfo)
const short	NNFMgrUpdateMathExpCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrMathExpCntlInfo* const pInfo)
const short	NNFMgrDeleteMathExpCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreatePrePostFixCntl	(NNFMgr *pNNFMgr, NFMgrPrePostFixCntlInfo* const pInfo)
const short	NNFMgrGetPrePostFixCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrPrePostFixCntlInfo * const pInfo)
const short	NNFMgrUpdatePrePostFixCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrPrePostFixCntlInfo* const pInfo)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrDeletePrePost FixCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreateDefault Cntl	(NNFMgr *pNNFMgr, NFMgrDefaultCntlInfo* const pInfo)
const short	NNFMgrGetDefault Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrDefaultCntlInfo * const pInfo)
const short	NNFMgrGetDefault CntlName	(NNCntlType Type, char CntlName)
const short	NNFMgrUpdate DefaultCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrDefaultCntlInfo* const pInfo)
const short	NNFMgrDeleteDefault Cntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreateLength Cntl	(NNFMgr *pNNFMgr, NFMgrLengthCntlInfo* const pInfo)
const short	NNFMgrGetLength Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrLengthCntlInfo * const pInfo)
const short	NNFMgrUpdateLength Cntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrLengthCntlInfo* const pInfo)
const short	NNFMgrDeleteLength Cntl	(NNFMgr *pNNFMgr, const char * const cntlName)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrCreateSubStringCntl	(NNFMgr *pNNFMgr, NFMgrSubStringCntlInfo* const pInfo)
const short	NNFMgrGetSubStringCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrSubStringCntlInfo * const pInfo)
const short	NNFMgrUpdateSubstringCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrSubStringCntlInfo* const pInfo)
const short	NNFMgrDeleteSubStringCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrGetCaseCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrCaseCntlInfo * const pInfo)
const short	NNFMgrGetJustifyCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrJustifyCntlInfo * const pInfo)
const short	NNFMgrCreateTrimCntl	(NNFMgr *pNNFMgr, NFMgrTrimCntlInfo* const pInfo)
const short	NNFMgrGetTrimCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrTrimCntlInfo * const pInfo)
const short	NNFMgrUpdateTrimCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrTrimCntlInfo* const pInfo)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrDeleteTrim Cntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreate CollectionCntl	(NNFMgr *pNNFMgr, NFMgrCollectionCntlInfo* const pInfo)
const short	NNFMgrGetCollection Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrCollectionCntlInfo * const pInfo)
const short	NNFMgrAddCntlTo Collection	(NNFMgr * pNNFMgr, const char* const CollName, int SeqNum, NNFMgrCntlInfo * const pInfo)
const short	NNFMgrGetCntlFrom Collection	(NNFMgr *pNNFMgr, const char * const cntlName, NNGetOp OpCode, NNFMgrCntlInfo* const pInfo)
const short	NNFMgrUpdate CollectionCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrCollectionCntlInfo* const pInfo)
const short	NNFMgrDelete CollectionCntl	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrGetDateTime FormatString	(NNFMgr * pNNFMgr, NNGetOp OpCode, short customFlag, char* const pFormatStr)
const short	NNFMgrIsRecursive Format	(NNFMgr *pNNFMgr, const char * const FirmatName, short * const IsRecursive)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrIsRecursive Collection	(NNFMgr *pNNFMgr, const char * const CollectionName, short * const IsRecursive)
const short	NNFMgrUpdateOutput Control	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrOutputControlInfo * const pInfo)
const short	NNFMgrDeleteOutput Control	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreate Literal	(NNFMgr *pNNFMgr, NFMgrLiteralInfo* const pInfo)
const short	NNFMgrGetLiteral	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrLiteralInfo * const pInfo)
const short	NNFMgrUpdateLiteral	(NNFMgr *pNNFMgr, const char * const literalName, NNFMgrLiteralInfo* const pInfo)
const short	NNFMgrDeleteLiteral	(NNFMgr *pNNFMgr, const char * const literalName)
const short	NNFMgrDelete Delimiter	(NNFMgr *pNNFMgr, const char * const delimiterName)
const short	NNFMgrCreateUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pInfo)
const short	NNFMgrAddName ValuePairs	(NNFMgr *pNNFMgr, const NFMgrNameValuePairInfo * const pInfo)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrGetUser DefinedType	(NNFMgr *pNNFMgr, const char* const pTypeName, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrGetFirstUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrGetNextUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrUpdateUser DefinedType	(NNFMgr *pNNFMgr, const char * const pTypeName, NNFMgrUserDefTypeInfo* const pInfo)
const short	NNFMgrDeleteUser DefinedType	(NNFMgr *pNNFMgr, const char * const pTypeName)
const short	NNFMgrCreateParse Control	(NNFMgr * pNNFMgr, const NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrGetParse Control	(NNFMgr *pNNFMgr, har * pParseName, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrGetFirstParse Control	(NNFMgr * pNNFMgr, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrGetNextParse Control	(NNFMgr * pNNFMgr, NNFMgrParseControlInfo * const pParseControlInfo)
const short	NNFMgrUpdateParse Control	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrParseControlInfo * const pInfo)

<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrDeleteParse Control	(NNFMgr *pNNFMgr, const char * const cntlName)
const short	NNFMgrCreateFormat	(NNFMgr * pNNFMgr, const NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)
const short	NNFMgrAppendField ToInputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, const NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrAppendField ToOutputFormat	(NNFMgr * pNNFMgr, const char * const pFormatName, const NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrAppend FormatToFormat	(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	NNFMgrGetFirst Format	(NNFMgr * pNNFMgr, NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)



<b>Return Type</b>	<b>Function</b>	<b>Arguments</b>
const short	NNFMgrGetNextFormat	(NNFMgr * pNNFMgr, NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)
const short	NNFMgrGetFirstFieldFromInputFormat	(NNFMgr * pNNFMgr, const char * const pFormatName, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetNextFieldFromInputFormat	(NNFMgr * pNNFMgr, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetFirstFieldFromOutputFormat	(NNFMgr * pNNFMgr, const char * const pFormatName, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetNextFieldFromOutputFormat	(NNFMgr * pNNFMgr, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetFirstChildFormat	(NNFMgr * pNNFMgr, const char * const pParentName, NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)
const short	NNFMgrGetNextChildFormat	(NNFMgr * pNNFMgr, NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)
const short	NNFMgrUpdateFormat	(NNFMgr * pNNFMgr, const char * const fmtName, const NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatInfo)
const short	NNFMgrDeleteFormat	(NNFMgr * pNNFMgr, const char * const fmtName)

---

# Libraries

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

## Library Files

UNIX Library	Description
libnnrulesfmt.so	NEONRules and NEONFormatter Library
libnntools.so	Generic Tool Set Library
libnnfmgr.so	Format Management Library
libnnses.so	Session-Specific Library
libnnsesdbold.so	MQSeries Integrator Session-Specific Libraries
libnnNetExits.so	User Exit Library
libnnaim.so	High-Level MQSeries Integrator APIs
libnnsql.so	MQSeries Integrator SQL Object Interface Library
—	System/Compiler Specific Libraries
—	Database-Dependent Libraries

### Notes:

Library file extensions are .so or .sl for UNIX, .DLL for NT, and .a for AIX.

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

### WARNING!

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

---

---

## Chapter 3

# Formatter APIs

---

---

## Formatter Member Functions

### Formatter Constructor

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 uses a session pointer to the input configuration database object.

#### Parameters #1

Name	Type	Input/ Output	Description
DatabaseSession Object	DbmsSession*	Input	Name of the current open database session.

#### 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 157. For information on user-defined type input field validation, see *User-Defined Type Input Field Validation* on page 197.

## Parameters

Name	Type	Input/Output	Description
DatabaseSession Object	DbmsSession*	Input	Name of the currently open database session.
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

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

## See Also

OpenDbmsSession

## Formatter Destructor

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 `NEONFormatter` 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

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

---

### 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
DatabaseSession Object	DbmsSession*	Input	Null pointer or pointer to a new and currently open database session.

### Remarks

You can use this function to do one or both of the following:

- Let Formatter know the database session is closed:

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

 then close the database session.
- 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.
Formatter formatter(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

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

---

### Note:

THREAD SAFETY NOTE: All input messages added using AddInputMessage() will be processed entirely within the thread from which they were added. See *Using the Thread Safe Release of the Formatter Engine* on page 202 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 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:**

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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

**See Also**

[parse](#)

[reformat](#)

[PreloadInFormat](#)

## AddInputMessage

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](#)

[PreloadInFormat](#)

## AddOutputFormat

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 *Using the Thread Safe Release of the Formatter Engine on page 202* for more information.

---

### Syntax

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

### Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Identifier to retrieve a format 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

## **See Also**

[PreloadOutFormat](#)

## RemoveOutputFormat

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 *Using the Thread Safe Release of the Formatter Engine on page 202* for more information.

---

### Syntax

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

### Parameters

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

### Remarks

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

### Return Value

Void

## PreloadInFormat

PreloadInFormat() preloads an input format into memory. If you do not 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.

---

### WARNING!

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

---



---

### Note:

THREAD SAFETY NOTE: PreloadInFormat() only loads input formats for the thread from which it is called. See *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

## 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

You built input formats `IFFormat1`, `IFFormat2`, and `IFFormat3` using the Formatter GUI tool or the Format Management API functions, and your Formatter application uses these formats. You can preload these format definitions prior to calling the other Formatter functions by adding the following calls to your application program:

```
// Construct Formatter instance
Formatter myFormatter;

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

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

## See Also

[PreloadOutFormat](#)

## PreloadOutFormat

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.

---

### WARNING!

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

---



---

### Note:

THREAD SAFETY NOTE: PreloadOutFormat() only loads output formats for the thread from which it is called. See *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

## 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

You built output formats `OFFFormat1`, `OFFFormat2`, and `OFFFormat3` using the Formatter GUI tool or the Format Management API functions, and your Formatter application uses these formats. You can preload these format definitions prior to calling the other Formatter functions by adding the following calls to your application program:

```
// Construct Formatter instance
Formatter    myFormatter;

// Preload output formats.
myFormatter.PreLoadOutFormat("OFFFormat1");
myFormatter.PreLoadOutFormat("OFFFormat2");
myFormatter.PreLoadOutFormat("OFFFormat3");

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

## See Also

[PreloadInFormat](#)

## StartDebug

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 the detail of the debug information. Currently, only BASIC applies.
pOutputBuffer	ostream&	Input	User-provided buffer. All debug information is 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

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

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Remarks

parse() can be called without reformatting the input messages into output messages. Formatter attempts 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() calls this function if it has not already been called for the current set of input messages.

If no input messages were added using AddInputMessage(), parse() fails. When the error-handling routines are used to return the error message, a “no input message” error is 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 end up with six 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

## **See Also**

[AddInputMessage](#)

[GetFieldAscii](#)

[reformat](#)



## reformat

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 is 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 is returned.

---

### WARNING!

It is not recommended that you 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 formatted using reformat(), and a substitute string is used in the output control, the input field value must be found in the set of substitute string entries or the output field is not output. If the input field value is not

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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### **See Also**

[reformat](#)

[AddOutputFormat](#)

[parse](#)

## GetFieldAscii/GetFieldString

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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; NULL 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`. You use `Formatter` to parse a message with this input format, and you want to get the value of the input field `F1`. An example sequence of `Formatter` function calls, including a call to `GetFieldAscii()` is:

```
// Construct Formatter instance
Formatter    myFormatter;

// 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

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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; NULL 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`. 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` by its tag value, not its name. Here is an example sequence of `Formatter` function calls, including a call to `GetFieldAsciiByTag()`:

```
// Construct Formatter instance.
Formatter myFormatter;

// 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

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

Returns the number of output message groups in the Formatter object. There is 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[AddOutputFormat](#)

## GetOutMsgGroup

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 one 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

## **See Also**

[GetMsg](#)

## GetParsedInMsgCount

GetParsedInMsgCount() returns the number of input messages parsed by 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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

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

### Example

See *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[AddInputMessage](#)

## GetParsedInMsg

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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, the index would be 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

[AddInputMessage](#)

## SetUserTypeValidationOn

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 *UUsing the Thread Safe Release of the Formatter Engine* on page 202 or more information.

---

### Syntax

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

### Parameters

None

### Return Value

None.

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

### Example

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[SetUserTypeValidationOff](#)

[UserTypeValidationIsOn](#)

## SetUserTypeValidationOff

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

None.

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

### Example

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[SetUserTypeValidationOn](#)

[UserTypeValidationIsOn](#)

## UserTypeValidationIsOn

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[SetUserTypeValidationOn](#)

[SetUserTypeValidationOff](#)

---

# OutMsg Class Member Functions

## GetMsgBuffer

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 not be allocated by the Formatter object. If you need to maintain 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

## See Also

[AddInputMessage](#)

[GetMsgLength](#)

## GetMsgLength

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[GetMsgBuffer](#)



---

# OutMsgGroup Class Member Functions

## GetMsg

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 outMsg 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

## GetMsgCount

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 in an output message group.

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

### Example

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[GetMsg](#)

## GetParsedOutMsgCount

GetParsedOutMsgCount() 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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

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

### Example

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[GetMsg](#)

## GetParsedOutMsg

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

Name	Type	Input/Output	Description
index	int	Input	Integer that represents a given subcomponent of an output format. For example, an index value of two (2) returns the second subcomponent of the flat or compound output format.

### Return Value

ParsedMessage \*. This returns a pointer to the ParsedMessage object from which ParsedMessage API calls can be made. If the ParsedMessage object represents a flat format, use ParsedMessage::GetFieldComp() to extract the final reformatted values of each message field. If the ParsedMessage object represents a compound format, use ParsedMessage::GetMsgComp() to extract another level of subcomponents.

### Example

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[GetMsg](#)

---

# ParsedField Class Member Functions

## GetInfo

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

## See Also

[parse](#)

## GetAsciiValue/GetStringValue

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

## GetValue

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.

See Appendix E, *Supported Data Types* for more information.

---

### Note:

THREAD SAFETY NOTE: GetValue() returns the value of the specified field in a parsed message within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 202 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.
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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

## GetFmtValue

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

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

### Return Value

None



## GetFmtValueLen

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

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

## GetByteOffset

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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

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 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

## GetMsgComp

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 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

[GetFieldComp](#)

## GetInfo

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 *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

## GetFieldComp

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

---

### Note:

THREAD SAFETY NOTE: GetFieldComp() returns the component field within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 202 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 *Traversing a Parsed Message: apitest.cpp* on page 468.

### See Also

[parse](#)

[GetMsgComp](#)

## GetFmtValLen

Returns length in bytes of formatted submessage.

---

**Note:**

THREAD SAFETY NOTE: GetFmtValLen() returns the component field within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 202 for more information.

---

### Syntax

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

### Parameters

None

### Return Value

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

## GetFmtVal

Returns buffer containing the Formatter submessage.

---

### Note:

THREAD SAFETY NOTE: GetFmtVal() returns the component field within the current thread. See *Using the Thread Safe Release of the Formatter Engine* on page 202 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. The API defines how the user is expected to construct functions callable by Formatter rather than to define functions called by the user application.

You must link 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 include 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.
- 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, Formatter tries to resolve the exit name into a callable function address. Since the function name and address are developed outside the scope of Formatter, Formatter has to ask (call) the user application to provide the function address. This means 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 in `nnuserexit.dll`. To override this stub function you must create the

replacement function, compile it, and then link it along with the `nnrulesfmt.lib` and `nnsesdbold.lib` libraries into your own dll named `nnuserexit.dll`. When you build the dll, the User Exit Lookup function must be exported with the ordinal value of 1000. This is done using the following linker option:

```
/EXPORT:NNGetUserExitFuncPtrs,@1000
```

The following is a sample makefile rule to build your `nnuserexit.dll`:

```
MY_OBJECTS=my_user_exit.obj
DBLIB=C:\MSSQL-6.5\DBLIB\NTWDBLIB.LIB
NNLIB=C:\NEONet\lib\nnrulesfmt.lib c:\NEONet\lib\nses.bold.lib

nnuserexit.dll: $(OBJECTS)
    del nnuserexit.lib
    del nnuserexit.dll
    cl.exe /nologo /link /dll \
        $(MY_OBJECTS) \
        $(DBLIB) \
        $(NNLIB) \
        /EXPORT:NNGetUserExitFuncPtrs,@1000 \
        /out:"nnuserexit.dll"
```

Make sure that the `PATH` environment variable has the name of the directory containing your `nnuserexit.dll` ahead of the library directory, so that the replacement User Exit Lookup function is used instead of the stub function.

## NNGetUserExitFuncPtrs

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

```
extern "C" 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
<code>acFuncName</code>	<code>char*</code>	Input	Exit function name associated with an output format control encountered during the reformat process.
<code>rUEptr</code>	<code>NN_EXIT_FUNC_t &amp;</code>	Output	Exit function pointer. This parameter is returned as <code>NULL</code> or a valid exit function address of type <code>NN_EXIT_FUNC_t</code> .
<code>rUEClUpPtr</code>	<code>NN_EXIT_CLEANUP_FUNC_t &amp;</code>	Output	Exit Cleanup function pointer. If <code>rUEptr</code> is not <code>NULL</code> , this parameter can be <code>NULL</code> . A Cleanup function is used to clean up any memory allocations not already cleaned up as part of the user exit.

## Remarks

The body of `NNGetUserExitFuncPtrs()` is user-defined. Note that the type modifier `extern "C"` must be used to make sure that the function is exported by that name in the User Exit dll.

## Example

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"
.....

extern "C" 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.

---

# User Exit Function Developer Interface

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

## User Exit Function Specification

The user exit 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 is generated if it is used to initialize the user exit return value, rUEptr, in lookup function NNGetUserExitFuncPtrs(). The typedef is in the nnextit.h header file.

### Syntax

```
typedef NNextitRet (*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 uxit 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().

<b>Name</b>	<b>Type</b>	<b>Input/ Output</b>	<b>Description</b>
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:

## Session Handle Types

<b>Database Type</b>	<b>SessionHandle Type</b>
Sybase (CTLIB)	CS_CONNECTION*
Sybase (dblib)	DBPROCESS*
MS SQL Server (dblib)	DBPROCESS*
Oracle	Lda_Def*
ODBC (NT only) or DB2	HDBC*

## Example

If the MQSeries Integrator portion of the user application has an open connection to a Sybase RDBMS, and the user exit function must 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

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 which 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

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**

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

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

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

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

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

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

Here's an example of how to use 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 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 Formatter checks to see if the return status is equal to NN\_ERSTATUS\_OK. If any other value is returned, Formatter assumes an error occurred and fails the reformat process. If possible, the user application developer should treat these 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 NEONet 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 NEONet 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 NEONet 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)

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

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 lLen); // <in> length of array of bytes
```

### Parameters

Name	Type	Input/Output	Description
pabVal	const char*	Input	Pointer to byte array.
lLen	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

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 is 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

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 is generated if it is used to initialize the user exit return value, `rUEClUpPtr`, in lookup function `NN GetUserExitFuncPtrs()`. The typedef is in the `nnexit.h` header file.

### Syntax

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

### Parameters

N/A

### Remarks

Formatter checks to see if the return value is equal to `NN_ERSTATUS_OK`. If not, 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 the `nnexit.h` header file.
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. Build a DLL called `nnuserexit.dll`, exporting `NNGetUserExitFuncPtrs` with the ordinal value 1000.
4. Set the `PATH` environment variable so that `nnuserexit.dll` will be found before the stub version in the library.
5. In the Formatter GUI, specify the name of the exit routine in the Exit Routine field in the Field Format Output Control Tool window.

### Example

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 non fatal 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 reformat process to fail.

---



---

## 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 are defined in `nnuserfunction.h` header file.

You define user callbacks as methods in a callback class derived from a MQSeries Integrator-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 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 `ses.h` and `nnparsedflds.h`. Use the following database session handles to access the current connection to the database.

## Session Handle Types

Database Type	SessionHandle Type
Sybase (CTLIB)	CS_CONNECTION*
Sybase (dblib)	DBPROCESS*
MS SQL Server (dblib)	DBPROCESS*
Oracle	Lda_Def*
ODBC (NT only) or DB2	HDBC*

See *Using Formatter APIs to Reformat a Message: msgtest.cpp* 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 because 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

NameValuePair is the basic element of the array type passed into the callback methods described. In general, a database object, like a parse control, has 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

<b>Name</b>	<b>Type</b>	<b>Description</b>
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

<b>Name</b>	<b>Prototype</b>	<b>Description</b>
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)**

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

#### **Syntax**

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

#### **Parameters**

None

## NameValuePair (Alternate Constructor)

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)

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)

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)**

Destructor.

### **Syntax**

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

### **Parameters**

None

## **MakeNull**

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

### **Syntax**

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

### **Parameters**

None

## Set

Sets name and value to the input values.

### Syntax

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

### Parameters

<b>Name</b>	<b>Type</b>	<b>Input/ Output</b>	<b>Description</b>
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 MQSeries Integrator-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 derive their own callback classes. NNUserFunction is used when MQSeries Integrator DbmsSession and MQSeries Integrator NNParsedFields are not needed.

## **class NNDBUserFunction: public NNUserFunction**

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

## **class NNDBFieldsUserFunction: public NNUserFunction**

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

<b>Class Name</b>	<b>Dbms Session?</b>	<b>Parsed Fields?</b>	<b>User Parameters?</b>
NNGenericUserFunction	No	No	Yes
NNDBUserFunction	Yes	No	Yes
NNDBFieldsUserFunction	Yes	Yes	Yes

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

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

## NNUserFunction

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 described on page 168.



## NNGenericUserFunction

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.

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

## **NNGenericUserFunction Member Functions**

### **Callback (No Parameters)**

A Formatter feature that uses objects derived from this class 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)

A Formatter feature that uses objects derived from this class 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	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

## Callback (userRuntimeData)

A Formatter feature that uses objects derived from this class call this method if there are both name/value pairs and user runtime-allocated 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
nameValuePairArray	NameValuePair*	Input	Array of name/value pairs retrieved from the database.
userRuntimeData	void*	Input	User-allocated runtime data obtained by the RuntimeDataLookup() method.

## Callback (userRuntimeData)

A Formatter feature that uses objects derived from this class call this method if there is user runtime-allocated 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 runtime data obtained by the RuntimeDataLookup() method.

## RuntimeDataLookup

Formatter calls `RuntimeDataLookup()` after looking up a callback object of this type to obtain a pointer to user-allocated runtime 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 in some way, depending on the feature being supported, and passes it to <code>Lookup()</code> . <code>Lookup()</code> may or may not use it to determine the data address to pass back.

## NNDBUserFunction

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.



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

## NNDBUserFunction Member Functions

### Callback (dbSession)

A Formatter feature that uses objects derived from this class calls 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)

A Formatter feature that uses objects derived from this class calls 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	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

## Callback (dbSession, nameValuePairArray userRuntimeData)

A Formatter feature that uses objects derived from this class calls 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	constDbmsSession&	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 runtime data obtained by the RuntimeDataLookup() method.

## Callback (dbSession, userRuntimeData)

A Formatter feature that uses objects derived from this class calls this method if there is user-allocated runtime 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	constDbmsSession&	Input	Handle to the current database session.
userRuntimeData	void*	Input	User-allocated runtime data obtained by the RuntimeDataLookup() method.

## RuntimeDataLookup

Formatter calls `RuntimeDataLookup()` after looking up a callback object of this type to obtain a pointer to user-allocated runtime 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 <code>Lookup()</code> . <code>Lookup()</code> may or may not use it to determine the data address to pass back.

## NNDBFieldsUserFunction

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

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

A Formatter feature that uses objects derived from this class calls 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 NNParsedFields&	Input	The set of all parsed fields.

## Callback (dbSession, parsedFields, nameValuePairArray)

A Formatter feature that uses objects derived from this class calls 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	constDbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsedFields&	Input	The set of all parsed fields.
nameValuePairArray	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

## Callback (dbSession, parsedFields, nameValuePairArray, userRuntimeData)

A Formatter feature that uses objects derived from this class calls 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	constDbmsSession&	Input	Handle to the current database session.
parsedFields	const NNParsedFields&	Input	The set of all parsed fields.
nameValuePairArray	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)

A Formatter feature that uses objects derived from this class calls this method if there is user-allocated runtime 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 NNParsedFields&	Input	The set of all parsed fields.
userRuntimeData	void*	Input	User-allocated runtime data obtained by the RuntimeDataLookup() method.

## RuntimeDataLookup

Formatter calls `RuntimeDataLookup()` after looking up a callback object of this type to obtain a pointer to user-allocated runtime 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 <code>Lookup()</code> . <code>Lookup()</code> may or may not use it to determine the data address to pass back.

## User Callback Lookup Interface

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

## NNFunctionKeyPairCollection

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:
```

### 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

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 MQSeries Integrator 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

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**

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 either the Formatter GUI or Format Management APIs.

---

## Notes:

- 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.
- 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.
- 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

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

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

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 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

```
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. 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. This means 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; that is, one thread cannot call `GetOutMsgGroup` to obtain the output that another thread has created by calling `reformat`. The template defined earlier for formatting changes slightly, as shown in the following example:

### Example

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 desired 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 desired output message, call `AddOutputFormat` to add the output format.

Call `Reformat` on the `Formatter` instance.

[Formatter formats 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 is affected by Thread Safety. There is no impact on the `OutMsg`, `OutMsgGroup`, `ParsedField`, and `ParsedMsg` APIs. Formatter Class API functions impacted by Thread Safety are listed in the following table:

Function	Description
<code>AddInputMessage</code>	All input messages added using <code>AddInputMessage()</code> are processed entirely within the thread from which they were added.
<code>AddOutputFormat</code>	The output formats added using <code>AddOutputFormat()</code> are used to specify formatting rules for all input messages added within the same thread.
<code>RemoveOutputFormat</code>	<code>RemoveOutputFormat()</code> removes all output formats only for the thread from which it is called.
<code>PreloadInFormat</code>	<code>PreloadInFormat()</code> loads input formats only for the thread from which it is called. If <code>PreloadInFormat()</code> is called from the main thread, its results are not available to any other threads.

<b>Function</b>	<b>Description</b>
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.

<b>Function</b>	<b>Description</b>
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.
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() or AddOutputFormat(), reformat() or parse(), GetOutMsgGroup() or GetParsedInMsg(), internal buffers are stored for input messages, output messages, and so on. After you retrieve 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 provided by POSIX pthreads and UI threads with the thread-specific data interface.

---

### **Note:**

In a Windows NT environment, this procedure does not perform data cleanup.

---

## Linking

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

---

# Formatter Error Handling

## GetErrorCode

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 [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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### See Also

[GetErrorMessage](#)

## GetErrorMessage

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 Using the Thread Safe Release of the Formatter Engine 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 *Using Formatter APIs to Reformat a Message: msgtest.cpp* on page 457.

### 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	Mandatory 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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.



<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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::GetFieldAscii().	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::GetFieldAscii().	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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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().
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.



<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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 <code>apitest -d</code> to help isolate the field missing the termination delimiter.
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.
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

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
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.



<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
-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.
-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.

<b>Code</b>	<b>Error Name</b>	<b>Error Message</b>	<b>Error Explanation</b>	<b>Response to Error</b>
-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 expression 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

# Formatter Management APIs

---

When adding formats, 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 database where case sensitivity is irrelevant, 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`.

If you import components into a case-insensitive database that were exported from a case-sensitive database, these differences cause NNFie to fail during

import because a conflict arises when two components are named the same with only case differences.

See ***System Management*** for information on using NNFie and changing a case-insensitive installation to case sensitive.

---

# General Formatter Management APIs

## NNFMgrInit

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 open database session.

### 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](#)

[OpenDbmsSession](#)

## NNFMgrClose

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

---

### Note:

NNFMgrClose() cleans up resources claimed by NNFMgrInit() but 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

When using Format Management APIs, the user must 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
<code>_p</code>	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 use 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 you import components exported from a case-sensitive database into a case-insensitive database, the case differences cause NNFIe to fail during import if a conflict arises between two components named the same with only case differences.

See *System Management* for information on using NNFIe and changing a case-insensitive installation to case sensitive.

---

## Field Management API Structure

### **NNFMgrFieldInfo**

NNFMgrFieldInfo is a structure containing field information.

#### **Syntax**

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

    long initFlag;
};
```



## Parameters

<b>Name</b>	<b>Type</b>	<b>Description</b>
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

NNFMgrCreateField() adds a field to the database.

### Syntax

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

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr *	Input/Output	Valid FMgr session previously returned by NNFMgrInit().
pFieldInfo	const NNFMgrFieldInfo * const	Input	Information about the field to add.

### 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

NNFMgrGetFirstField() retrieves field information for the first field from the database. To iterate through 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.

### Return Value

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

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

### See Also

[NNFMgrFieldInfo](#)

[NNFMgrCreateField](#)

[NNFMgrGetNextField](#)

## NNFMgrGetNextField

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 field.

### Return Value

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

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

### See Also

[NNFMgrFieldInfo](#)

[NNFMgrCreateField](#)

## NNFMgrUpdateField

Updates an existing field in the database. Before calling this function, the NNFMgrFieldInfo data structure must be initialized with the new values. The NNFMgrGetField function can be used to populate this data structure with the current values. The fldName parameter must be the current name of the Field. The pInfo structure should contain the new field name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateField(
    NNFMgr *pNNFMgr,
    const char * const fldName,
    const NNFMgrFieldInfo * const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
fldName	const char * const	Input	The name of the field.
pInfo	NNFMgrSubstituteCntlInfo* const	Input	Pointer to a valid NNFMgrSubstituteCntlInfo structure.

### Remarks

A call to NNF\_CLEAR for pInfo should be made prior to populating the structure or calling this API.

### Return Value

Returns a non-zero integer value on success and zero on failure. Use GetLastError() to retrieve the number for the error that occurred, then use

`GetErrorMessage()` to retrieve the error message associated with that error number.

### **Example**

```
NNFMgrFieldInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetLiteral(pNNFMgr, "FirstName", &Info );
strcpy(Info.fieldName, "First");
// change field name
strcpy(Info.fieldDescription, "Customer's First Name");
// change description
NNFMgrUpdateField(pNNFMgr, "FirstName", &Info);
```

### **See Also**

[NNFMgrUpdateParseControl](#)

[NNFMgrUpdateOutputControl](#)

## NNFMgrDeleteField

Deletes a single field from the database.

### Syntax

```
const short NNFMgrDeleteField(
    NNFMgr *pNNFMgr,
    const char * const fldName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
fldName	const char * const	Input	The name of the field.

### Remarks

This function does not perform any referential integrity checks on the database. If the field you are deleting is used in one or more formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

### See Also

[NNFMgrDeleteParseControl](#)

[NNFMgrDeleteOutputControl](#)

---

# Output Format Control Management APIs

The output control API structures are used to create or get controls. This section details the following output control API structures:

- NNFMgrOutMstrCntlInfo
- NNFMgrSubstituteCntlInfo
- NNFMgrUserExitCntlInfo
- NNFMgrMathExpCntlInfo
- NNFMgrMathExpCntlSegmentInfo
- NNFMgrPrePostFixCntlInfo
- NNFMgrDefaultCntlInfo
- NNFMgrLengthCntlInfo
- NNFMgrSubStringCntlInfo
- NNFMgrCaseCntlInfo
- NNFMgrJustifyCntlInfo
- NNFMgrCollectionCntlInfo

---

## **WARNING!**

If you use 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.

If you import components exported from a case-sensitive database into a case-insensitive database, the case differences will cause NNFIe to fail during import if a conflict arises between two components named the same with only case differences.

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

---



# Output Control Management API Structures

## NNFMgrOutMstrCntlInfo

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 type
    char 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
    short 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 data type.

<b>Name</b>	<b>Type</b>	<b>Description</b>
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. shortbaseDataType is used 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 User Defined data types
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 data types.
tagLitrName	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 tagLitrName is specified, this field is ignored.

<b>Name</b>	<b>Type</b>	<b>Description</b>
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 data types.
operationType	short	Used only for fieldType = OUTFIELD_FORMAT_Calculated_Field. Valid values are: OPERATION_Add OPERATION_Subtract OPERATION_Multiply OPERATION_Divide
fldLitrName	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 fldLitrName 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.

<b>Name</b>	<b>Type</b>	<b>Description</b>
childCntlType	NNCntl Type	<p>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:</p> <p>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</p>
initFlag	short	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

## NNFMgrSubstituteCntlInfo

Houses information used to create a new substitute control or gets an existing substitute control.

### Syntax

```
typedef struct NNFMgrSubstituteCntlInfo {
    char cntlName[33];
    char inputLitrName[33];
    unsigned char inputValue[128];
    unsigned short inputValueLen;
    char outputLitrName[33];
    unsigned char outputValue[128];
    unsigned short outputValueLen;
    short outputValueType; // Data type of outputvalue
    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 request a default name, the cntlName field is populated with the default name upon return from the API call.
inputLitrName	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 inputLitrName is specified, this field is ignored.
inputValueLen	unsigned short	The length (in bytes) of the valid data stored in inputValue.

<b>Name</b>	<b>Type</b>	<b>Description</b>
outputLitrName	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 outputLitrName is specified, this field is ignored.
outputValueLen	unsigned short	The length (in bytes) of the valid data stored in outputValue.
outputValueType	short	The data type of the output field data. Must be a valid NEONet data type.
initFlag	short	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

## NNFMgrUserExitCntlInfo

Houses information used 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 request 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

Houses information used 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 request 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

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 request 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

Houses information used to create a new prefix or postfix control or gets an existing prefix or postfix control.

### Syntax

```
typedef struct NNFMgrPrePostFixCntlInfo {
    char cntlName[33];
    char litrlName[33];
    unsigned char value[128];
    unsigned short valueLen;
    NNFPrefPostFix 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 request 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 postfix (suffix).
value	unsigned char[128]	The value data to be added to the output field is a prefix or postfix (suffix). If litrlName is specified, this field is ignored.
valueLen	unsigned char	The length (in bytes) of the valid data in value.
place	NNFPrefPostFix	PREFIX or POSTFIX

<b>Name</b>	<b>Type</b>	<b>Description</b>
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

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 request 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 (in bytes) of the valid data in value.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

## NNFMgrLengthCntlInfo

Houses information used to create a new length control or gets an existing length control.

### Syntax

```
typedef struct NNFMgrLengthCntlInfo {
    char cntlName[33];
    char padLitrName[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 request a default name, the cntlName field is populated with the default name upon return from the API call.
padLitrName	char[33]	The name of the literal that contains the data to be used to pad to the length specified in dataLen.
padValue	unsigned char[128]	The value to be used to pad to the length specified in dataLen. If padLitrName is specified, this field is ignored.
padValueLen	unsigned short	The length (in bytes) of the valid data in padValue.
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

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, 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 request 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.
padValue	unsigned char[128]	The value to be used to pad to the length specified in len. If padLitrlName is specified, this field is ignored.

<b>Name</b>	<b>Type</b>	<b>Description</b>
padValueLen	unsigned short	The length (inbytes) of the valid data in padValue.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.



## NNFMgrCaseCntlInfo

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. This field should contain the name of the new control or a zero-length string (“”) to generate a default name. If you request a default name, the cntlName field is populated with the default name upon return from the API call.
caseId	NNFCase	Indicates whether this case control is an uppercase or lowercase 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

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 request a default name, the cntlName field is populated with the default name upon return from the API call.
justify	NNFJustify	Indicates the type of justification to perform. 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.

## NNFMgrTrimCntlInfo

NNFMgrTrimCntlInfo is a structure that houses information used to create a trim control.

### Syntax

```
NNFMgrTrimCntlInfo {
    char cntlName[NAME_LENGTH+1];
    char trimCharLitrName[LITRL_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. To create APIs, this field should contain the name of the new control, or a zero-length string (“”) to generate a default name. For a default name, the cntlName field is populated with the default name upon return from the API call.
trimCharLitrName	char[NAME_LENGTH+1]	The name of the Literal formatter component defining the pad character for the trim operation. If no name is entered, a new literal component is 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.

<b>Name</b>	<b>Type</b>	<b>Description</b>
trimCharLen	unsigned short	The length of the pad character literal. NOTE: Although the pad literal value can be greater than 1 byte, only the first byte of the value is used.
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.

## NNFMgrCollectionCntlInfo

Houses information used to create a new collection control or get 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 request 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.

## NNFMgrCntlInfo

Generic control info structure. Houses information used to add an existing control to a collection, or to retrieve a control from a collection. This structure is used to return alternate key information for a control inside a collection. The alternate key can then be used with a GET API call for the specific type of control.

### Syntax

```
typedef struct NNFMgrCntlInfo {
    char cntlName[NAME_LENGTH+1];
    NNCntlType cntlType;
    long initFlag;
} NNFMgrCntlInfo;
```

### Parameters

Name	Type	Description
cntlName	char	Name of control.
cntlType	NNcntlType	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.

## Output Control Management APIs

The Output Control Management APIs are used to retrieve output controls from the database. These APIs have names such as `NNFMgrGetxxxCntl`, where `xxx` is the type of control to be retrieved. The `NNFMgrGetxxxCntl` APIs can be used to return a specific control or to iterate through the list of controls of a specified type in the database. One control is returned for each `NNFMgrGetxxxCntl` call. You choose the behavior by setting an argument in the API call.

### OpCode

Each `NNFMgrGetxxxCntl` API takes an operation code, or `OpCode`, as its first argument. The `OpCode` argument selects the behavior of an API call and designates the control returned by the API call. `OpCode` determines the location of the returned control within a the list of all controls of the specified type.

Argument	Description
GET	The user provides the control name in the input structure. The rest of the structure is filled in with information from the control record.
GET_FIRST	The function returns the first row of the control table. No input data from the input structure is used.
GET_NEXT	The function returns the next row from the control table. The user must have called this function previously using the <code>GET_FIRST</code> argument.

## ***Getting a specific control***

1. Pass the OpCode argument of the NNFMgrGetxxxCntl API call with the value GET.

---

### **Note:**

By using the GET value, the NNFMgrGetxxxCntl API looks at the CntlName field in the structure, indicated by the pInfo pointer.

---

2. Enter the name of the control you want to retrieve.

Following the API call, the pInfo structure contains the information for the requested control.

## ***Listing Controls***

To iterate through the list of controls, make at least one call to NNFMgrGetxxxCntl for the control type.

1. In the first call, pass the OpCode argument of the NNFMgrGetxxxCntl API call with the value GET\_FIRST.

Following the API call, the pInfo structure contains the information for the first control.

2. To retrieve the subsequent controls on the list, make a call to the NNFMgrGetxxxCntl API with the OpCode argument set to the value GET\_NEXT for each control.

---

### **Note:**

When the end of the list is reached, the NNFMgrGexxxCntl API call returns NO\_DATA\_FOUND. Call the NNFMgrGetxxxCntl APIs with an OpCode of GET\_FIRST before attempting to call the API with an OpCode of GET\_NEXT. If the GET\_NEXT OpCode value is used prior to the GET\_FIRST value, an error is returned.

---



## NNFMgrCreateOutMstrCntl

Creates a new output master control and associates it to a single child control. The child control can 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 NNFMgrCreateOutMstrCntl(
    NNFMgrOutMstrCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/Output	Description
pInfo	NNFMgrOutMstrCntlInfo*	Input/Output	Pointer to structure that provides data about CreateOutMstrCnt.

### Remarks

You can specify the literal name to use for tag by populating the tagLitrName field of the pInfo structure, and specify field value (used with Input Value Equals and Literal field types) by populating the fldLitrName field 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 GetLastError() to retrieve the number for the error that occurred; then use

`GetErrorMessage()` to retrieve the error message associated with that error number.

**See Also**

[NNFMgrOutMstrCntlInfo](#)

[NNFMgrGetOutMstrCntl](#)

## NNFMgrGetOutMstrCntl

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 NNFMgrGetOutMstrCntl(
    NNGetOp OpCode,
    NNFMgrOutMstrCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
OpCode	NNGetOp	Input/ Output	Pointer to structure that contains data about NNGetOpCode.
pInfo	NNFMgrOutMstrCntlInfo	Input/ Output	Pointer to structure that contains data about NNFMgrGetOutMstrCntl.

### Return Value

Returns a non-zero integer value. Use GetLastError() to retrieve the number for the error that occurred; then use GetErrorMessage() to retrieve the error message associated with that error.

### See Also

[NNFMgrCreateOutMstrCntl](#)

[NNFMgrGetOutMstrCntl](#)

## NNFMgrUpdateOutMstrCntl

Updates an existing version 4.x Output Control in the database. Before calling this function, the NNFMgrOutMstrCntlInfo data structure must be initialized with the new values. The API function NNFMgrGetOutMstrCntl may be used to populate this data structure with the current values. The cntlName parameter must be the current name of the Output Control. The pInfo structure should contain the new name if different from the current name.

### Syntax

```
const short NNFMgrUpdateOutMstrCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrOutMstrCntlInfo * const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.
pInfo	NNFMgrOutput ControlInfo * const	Input	Pointer to a valid NNFMgrOutMstrCntlInfo structure.

### Remarks

A call to NNF\_CLEAR for pInfo should be made prior to populating the structure or calling this API.

This function first deletes the named output control using the NNFMgrDeleteOutputControl API call; then calls NNFMgrCreateOutputControl to create it with the new values. All references

from parent components to this output control are maintained, even if the name of the control is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrOutputControlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetOutputControl(pNNFMgr, "StringWithNewline", &Info );
strcpy(Info.controlName, "StringWithSemicolon");// change name
strcpy(Info.suffix, "Semicolon");// change suffix literal
NNFMgrUpdateOutputControl(pNNFMgr, "StringWithNewline", &Info);
```

## See Also

[NNFMgrCreateOutMstrCntl](#)

[NNFMgrGetOutMstrCntl](#)

## NNFMgrDeleteOutMstrCntl

Deletes a single output master control from the database.

### Syntax

```
const short NNFMgrDeleteOutMstrCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrCreateOutMstrCntl](#)

[NNFMgrGetOutMstrCntl](#)

## Output Operation Controls

Output Operation Control APIs are used to define reformatting operations that can be performed on the data in output fields. The output operation types are described in the *NEONFormatter Overview* chapter.

This section details the following output operation control APIs:

- Substitute controls
- User Exit controls
- Math Expression controls
- PrePostFix controls
- Default controls
- Length controls
- SubString controls
- Case control
- Justify control
- Trim controls
- Collection controls

### 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

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.

### 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 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

See *Code Example for Substitute Controls* on page 507.

## See Also

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrAppendEntryToSubstituteControl](#)

[NNFMgrGetSubstituteCntl](#)

[NNFMgrGetNextEntryFromSubstituteCntl](#)

[NNFMgrUpdateSubstituteCntl](#)

[NNFMgrDeleteSubstituteCntl](#)

## NNFMgrAppendEntryToSubstituteControl

Appends a substitute entry to an existing substitute control named by the `cntlName` of the `pInfo` structure. Call `NNFMgrCreateSubstituteCntl()` 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
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input/ Output	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>pInfo</code>	<code>NNFMgrSubstituteCntlInfo*</code>	Input/ Output	Pointer to structure that provides data about <code>NNFMgrCreateSubstituteCntl</code> .

### 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

See *Code Example for Substitute Controls* on page 507.

## **See Also**

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrCreateSubstituteCntl](#)

[NNFMgrGetSubstituteCntl](#)

[NNFMgrGetNextEntryFromSubstituteCntl](#)

[NNFMgrUpdateSubstituteCntl](#)

[NNFMgrDeleteSubstituteCntl](#)

## NNFMgrGetSubstituteCntl

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. See *OpCode* on page 265.

### 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.
pInfo	NNFMgr SubstituteCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSubstituteCntl.
NumRemaining Entries	int* const	Input/ Output	Returns the number of remaining entries in this control.

## Remarks

The number of remaining entries in this control is returned in the `NumRemainingEntries` argument. You must use `NNFMgrGetNextEntryFromSubstituteCntl()` to get the remaining substitute entries for 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

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrCreateSubstituteCntl](#)

[NNFMgrAppendEntryToSubstituteControl](#)

[NNFMgrGetNextEntryFromSubstituteCntl](#)

[NNFMgrUpdateSubstituteCntl](#)

[NNFMgrDeleteSubstituteCntl](#)

## NNFMgrGetNextEntryFromSubstituteCntl

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.

### Remarks

Call NNFMgrGetSubstituteCntl prior to calling this routine. 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 zero (0) on failure. Use GetLastError() 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 507.

## **See Also**

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrCreateSubstituteCntl](#)

[NNFMgrAppendEntryToSubstituteControl](#)

[NNFMgrGetSubstituteCntl](#)

[NNFMgrUpdateSubstituteCntl](#)

[NNFMgrDeleteSubstituteCntl](#)



## NNFMgrUpdateSubstituteCntl

Updates an existing substitute control in the database. Before calling this function, the `NNFMgrSubstituteCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetSubstituteCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateSubstituteCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrSubstituteCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrOutput ControlInfo *</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrOutputControlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

This API function first truncates the list of substitution strings that may have been in the substitute control; then inserts the single substitution entry in the list. To add more substitution strings to the list, call the `NNFMgrAppendEntryToSubstituteCntl()` function.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrSubstituteCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetSubstituteCntl(pNNFMgr, "ReplaceWithBlanks", &Info );
strcpy(Info.inputLiteralName, "comma");
strcpy(Info.outputLiteralName, "blank");
NNFMgrUpdateSubstituteCntl(pNNFMgr,
                           "ReplaceWithBlanks", &Info);
strcpy(Info.inputLiteralName, "semicolon");
strcpy(Info.outputLiteralName, "blank");
NNFMgrAppendEntryToSubstituteCntl(pNNFMgr, &Info);
```

## See Also

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrCreateSubstituteCntl](#)

[NNFMgrAppendEntryToSubstituteControl](#)

[NNFMgrGetSubstituteCntl](#)

[NNFMgrDeleteSubstituteCntl](#)

## NNFMgrDeleteSubstituteCntl

Deletes a single substitute control from the database.

### Syntax

```
const short NNFMgrDeleteSubstituteCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrSubstituteCntlInfo](#)

[NNFMgrCreateSubstituteCntl](#)

[NNFMgrAppendEntryToSubstituteControl](#)

[NNFMgrGetSubstituteCntl](#)

[NNFMgrUpdateSubstituteCntl](#)

## User Exit Controls

### NNFMgrCreateUserExitCntl

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	NNFMgrUserExitCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateUserExitCntl.

#### Return Value

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

#### See Also

[NNFMgrUserExitCntlInfo](#)

[NNFMgrGetUserExitCntl](#)

## NNFMgrGetUserExitCntl

Gets a single User Exit control from the NNF\_YYY table. The location of the returned control within this table is determined by the OpCode argument. See *OpCode* on page 265.

### 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 is determined by the OpCode argument.
pInfo	NNFMgrUserExitCntlInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetUserExitCntl.

### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrUserExitCntlInfo](#)

[NNFMgrCreateUserExitCntl](#)

[NNFMgrUpdateUserExitCntl](#)

[NNFMgrDeleteUserExitCntl](#)

## NNFMgrUpdateUserExitCntl

Updates an existing User Exit control in the database. Before calling this function, the `NNFMgrUserExitCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetUserExitCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateUserExitCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrUserExitCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrUserExitCntlInfo*</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrUserExitCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the name of the control is changed.



## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrUserExitCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetUserExitCntl(pNNFMgr, "validateField", &Info );
Strcpy(Info.exitRoutine, "UE_ValidateField");
NNFMgrUpdateUserExitCntl(pNNFMgr, "validateField", &Info);
```

## See Also

[NNFMgrUserExitCntlInfo](#)

[NNFMgrCreateUserExitCntl](#)

[NNFMgrGetUserExitCntl](#)

[NNFMgrDeleteUserExitCntl](#)

## NNFMgrDeleteUserExitCntl

Deletes a single User Exit control from the database.

### Syntax

```
const short NNFMgrDeleteUserExitCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrUserExitCntlInfo](#)

[NNFMgrCreateUserExitCntl](#)

[NNFMgrGetUserExitCntl](#)

[NNFMgrUpdateUserExitCntl](#)

## Math Expression Controls

Math expression controls can contain any length of math expression. This is possible because the actual data is stored in a set of ordered segments in a separate table. Math expression controls are a form of collection. However, users can only append segments to a math expression, and can only access those segments sequentially from the first to the last segment. The parent math expression control is managed via the standard Create and Get APIs.

## NNFMgrCreateMathExpCntl

Creates a new math expression 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	NNFMgrMathExpCntlInfo* const	Input/Output	Pointer to structure that provides data about NNFMgrCreateMathExpCntl.

### 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

[NNFMgrMathExpCntlInfo](#)

[NNFMgrGetMathExpCntl](#)

[NNFMgrAppendSegToMathExpCntl](#)

[NNFMgrGetSegFromMathExpCntl](#)

[NNFMgrUpdateMathExpCntl](#)

[NNFMgrDeleteMathExpCntl](#)

## NNFMgrGetMathExpCntl

Gets a single math expression control from the database. The location of the returned control within the list of all math expressions is determined by the `OpCode` argument. See *OpCode* on page 265.

### Syntax

```
const short NNFMgrGetMathExpCntl(
    NNFMgr* pNNFMgr,
    NNGetOp OpCode,
    NNFMgrMathExpCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input/ Output	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>OpCode</code>	<code>NNGetOp</code>	Input/ Output	The location of the returned control within the list of all math expressions is determined by the <code>OpCode</code> argument.
<code>pInfo</code>	<code>NNFMgrMathExpCntlInfo*</code> <code>const</code>	Input/ Output	Pointer to structure that contains data about <code>NNFMgrGetMathExpCntl</code> .

### 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**

[NNFMgrMathExpCntlInfo](#)

[NNFMgrAppendSegToMathExpCntl](#)

[NNFMgrGetSegFromMathExpCntl](#)

[NNFMgrUpdateMathExpCntl](#)

[NNFMgrDeleteMathExpCntl](#)

## NNFMgrAppendSegToMathExpCntl

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.
CntlName	const char* const	Input/ Output	The name of the control
pInfo	const NNFMgrMath ExpCntl SegmentInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSegFrom MathExpCntl.

### Return Value

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



## NNFMgrGetSegFromMathExpCntl

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.
CntlName	const char* const	Input/ Output	The name of the control.
OpCode	NNGetOp	Input/ Output	The position of the returned segment is determined by the OpCode argument.
pInfo	NNFMgrMath ExpCntl SegmentInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetSegFromMathExpCntl.

### Remarks

The position of the returned segment is determined by the OpCode argument. See *OpCode* on page 265.

## 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

[NNFMgrMathExpCntlSegmentInfo](#)

[NNFMgrCreateMathExpCntl](#)

[NNFMgrGetMathExpCntl](#)

[NNFMgrAppendSegToMathExpCntl](#)

[NNFMgrUpdateMathExpCntl](#)

[NNFMgrDeleteMathExpCntl](#)

## NNFMgrUpdateMathExpCntl

Updates an existing math expression control in the database. Before calling this function, the `NNFMgrMathExpCntlInfo` data structure must be initialized with the new values. The `NNFMgrGetMathExpCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateMathExpCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrMathExpCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrMathExpCntlInfo*</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrMathExpCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

This API function truncates the list of math expressions that may have been in the Control, it does not insert math expression entries in the list. To add math expressions to the list, call the API function `NNFMgrAppendMathExpression`.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrMathExpCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetMathExpCntl(pNNFMgr, "timesThree", &Info );
Strcpy(Info.cntlName, "timesPi");// change name
Info.decimal_precision = 3;// change decimal precision
NNFMgrUpdateMathExpCntl(pNNFMgr, "timesThree", &Info);

NNFMgrMathExpressionInfo Expr;// build math expression
NNF_CLEAR(&Expr);
Strcpy(Expr.expression, "f1 * 3.14159");
Strcpy(Expr.outputControlName, "timesPi");
NNFMgrAppendMathExpression(pNNFMgr, &Expr);// append it
```

## See Also

[NNFMgrMathExpCntlSegmentInfo](#)

[NNFMgrCreateMathExpCntl](#)

[NNFMgrGetMathExpCntl](#)

[NNFMgrAppendSegToMathExpCntl](#)

[NNFMgrDeleteMathExpCntl](#)

## NNFMgrDeleteMathExpCntl

Deletes a single Math Expression control from the database.

### Syntax

```
const short NNFMgrDeleteMathExpCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrMathExpCntlSegmentInfo](#)

[NNFMgrCreateMathExpCntl](#)

[NNFMgrGetMathExpCntl](#)

[NNFMgrAppendSegToMathExpCntl](#)

[NNFMgrUpdateMathExpCntl](#)

## Pre/PostFix Controls

Pre/PostFix controls are used to add user-defined 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. Any of these controls can have `nullActionInd = 1`. If the input data for a field is NULL, the controls flagged with `nullActionInd = 1` are applied to the field data in the order they appear in the collection.

## NNFMgrCreatePrePostFixCntl

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.

### Return Value

Return a non-zero integer value on success and zero (0) 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

[NNFMgrPrePostFixCntlInfo](#)

[NNFMgrGetPrePostFixCntl](#)

[NNFMgrUpdatePrePostFixCntl](#)

[NNFMgrDeletePrePostFixCntl](#)



## NNFMgrGetPrePostFixCntl

Gets a single PrePostFix control from the database. The position of the returned segment is determined by the OpCode argument. See *OpCode* on page 265.

### 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.
pInfo	NNFMgrPrePostFixCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetPrePostFixCntl.

### 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

[NNFMgrPrePostFixCntlInfo](#)

[NNFMgrCreatePrePostFixCntl](#)

[NNFMgrGetPrePostFixCntl](#)

[NNFMgrUpdatePrePostFixCntl](#)

[NNFMgrDeletePrePostFixCntl](#)

## NNFMgrUpdatePrePostFixCntl

Updates an existing Pre/Postfix Control in the database. Before calling this function, the `NNFMgrPrePostFixCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetPrePostFixCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdatePrePostFixCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrPrePostFixCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrPrePostFixCntlInfo*</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrPrePostFixCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrPrePostFixCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetPrePostFixCntl(pNNFMgr, "NewlineSuffix", &Info );
Strcpy(Info.cntlName, "NewlinePrefix");// change name
Info.place = PREFIX;// change to prefix
NNFMgrUpdatePrePostFixCntl(pNNFMgr, "NewlineSuffix", &Info);
```

## See Also

[NNFMgrPrePostFixCntlInfo](#)

[NNFMgrCreatePrePostFixCntl](#)

[NNFMgrGetPrePostFixCntl](#)

[NNFMgrDeletePrePostFixCntl](#)

## NNFMgrDeletePrePostFixCntl

Deletes a single Pre/PostFix control from the database.

### Syntax

```
const short NNFMgrDeletePrePostFixCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrPrePostFixCntlInfo](#)

[NNFMgrCreatePrePostFixCntl](#)

[NNFMgrGetPrePostFixCntl](#)

[NNFMgrUpdatePrePostFixCntl](#)

## Default Controls

### NNFMgrCreateDefaultCntl

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.

#### 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 literalName 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 zero (0) on failure. Use GetLastError() to retrieve the number for the error that occurred; then use

`GetErrorMessage()` to retrieve the error message associated with that error number.

**See Also**

[NNFMgrDefaultCntlInfo](#)

[NNFMgrGetDefaultCntl](#)

[NNFMgrUpdateDefaultCntl](#)

[NNFMgrDeleteDefaultCntl](#)



## NNFMgrGetDefaultCntl

Gets a single default control from the database. The position of the returned segment is determined by the *OpCode* argument. See *OpCode* on page 265.

### 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.
pInfo	NNFMgrDefaultCntlInfo* const	Input/Output	Pointer to structure that provides data for NNFMgrGetDefaultCntl.

### Return Value

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

### See Also

[NNFMgrDefaultCntlInfo](#)

[NNFMgrGetDefaultCntl](#)

[NNFMgrUpdateDefaultCntl](#)

[NNFMgrDeleteDefaultCntl](#)

## NNFMgrGetDefaultCntlName

Naming individual controls can be time-consuming. In some cases, the user may not care what name is given to an individual control.

### Syntax

```
const short NNFMgrGetDefaultCntlName(
    NNCntlType Type,
    char* CntlName)
```

### Parameters

Name	Type	Input/Output	Description
Type	NNCntlType	Input/Output	Indicates the type of control. The valid values for Type are described in the following table.
CntlName	char*	Input/Output	Name of the control.

Type	Root Value	Reusable control
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

Type	Root Value	Reusable control
TRIM_CNTL	Trim	Yes
COLLECTION_CNTL	Collection	No

## Remarks

Default names can be generated in two ways:

1. Each of the `NNFMgrxxxCntlInfo` structures contains a `cntlName` member that names the control. If the user sets this name to an empty string (“”), the corresponding `NNFMgrCreatexxxCntl` API function detects the fact that no name has been provided, and automatically calls `GetDefaultCntlName()`. The generated name is stored in the `cntlName` field of the structure. This is the simplest way to get a default control name, and is recommended method. The user can retrieve the name of the newly created control from the `cntlName` field of the structure passed into the `NNFMgrCreatexxxCntl` API function.
2. The user can 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 to create the new control. If this happens, a duplicate key error occurs. If the user elects to use the second method, 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 locks out other transactions and prevents duplicate key errors.

## 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

Assume the user has created 10 substring controls. The following code fragments illustrate how to create a new substring control with a default name. In each example, 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);
```

See *Using Format Management APIs: fmgr.cpp* on page 475.

## NNFMgrUpdateDefaultCntl

Updates an existing default control in the database. Before calling this function, the `NNFMgrDefaultCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetDefaultCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateDefaultCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrDefaultCntlInfo* const pInfo )
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrDefaultCntlInfo* const</code>	Input	Pointer to a valid <code>NNFMgrDefaultCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrDefaultCntlInfo Info;  
NNF_CLEAR(&Info);  
NNFMgrGetDefaultCntl(pNNFMgr, "DefaultColor", &Info );  
Strcpy(Info.litrlName, "blue");// change literal name  
NNFMgrUpdateDefaultCntl(pNNFMgr, "DefaultColor", &Info);
```

## See Also

[NNFMgrDefaultCntlInfo](#)

[NNFMgrCreateDefaultCntl](#)

[NNFMgrGetDefaultCntl](#)

[NNFMgrDeleteDefaultCntl](#)

## NNFMgrDeleteDefaultCntl

Deletes a single default control from the database.

### Syntax

```
const short NNFMgrDeleteDefaultCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrDefaultCntlInfo](#)

[NNFMgrCreateDefaultCntl](#)

[NNFMgrGetDefaultCntl](#)

[NNFMgrUpdateDefaultCntl](#)



## Length Controls

### NNFMgrCreateLengthCntl

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	NNFMgr LengthCntl Info* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateLengthCntl.

#### 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 padLiteralName 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 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

[NNFMgrLengthCntlInfo](#)

[NNFMgrGetLengthCntlInfo](#)

[NNFMgrUpdateLengthCntlInfo](#)

[NNFMgrDeleteLengthCntlInfo](#)

## NNFMgrGetLengthCntl

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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrLengthCntlInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetLengthCnt.

### Remarks

The location of the returned control within the list of all Length controls is determined by the OpCode argument. The OpCode argument is on enumerated type. See *OpCode* on page 265.

### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrLengthCntlInfo](#)

[NNFMgrCreateLengthCntlInfo](#)

[NNFMgrUpdateLengthCntlInfo](#)

[NNFMgrDeleteLengthCntlInfo](#)

## NNFMgrUpdateLengthCntl

Updates an existing length control in the database. Before calling this function, the `NNFMgrLengthCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetLengthCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateLengthCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrLengthCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrLengthCntlInfo* const</code>	Input	Pointer to a valid <code>NNFMgrLengthCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrLengthCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetLengthCntl(pNNFMgr, "length7", &Info );
strcpy(Info.padLitrlName, "space");// change pad literal
Info.dataLen = 10;// change length
Strcpy(Info.cntlName, "length10");// change cntl name
NNFMgrUpdateLengthCntl(pNNFMgr, "length7", &Info);
```

## See Also

[NNFMgrLengthCntlInfo](#)

[NNFMgrCreateLengthCntlInfo](#)

[NNFMgrGetLengthCntlInfo](#)

[NNFMgrDeleteLengthCntlInfo](#)

## NNFMgrDeleteLengthCntl

Deletes a single length control from the database.

### Syntax

```
const short NNFMgrDeleteLengthCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName)
```

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrLengthCntlInfo](#)

[NNFMgrCreateLengthCntlInfo](#)

[NNFMgrGetLengthCntlInfo](#)

[NNFMgrUpdateLengthCntlInfo](#)



## SubString Controls

### NNFMgrCreateSubStringCntl

This function creates a substring control.

#### 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	NNFMgrSubStringCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateSubStringCntl.

#### Remarks

The substring control allows you to replace an incoming field value that 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.

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 padLiteralName 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 zero (0) 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**

[NNFMgrSubStringCntlInfo](#)

[NNFMgrGetSubStringCntl](#)

[NNFMgrUpdateSubStringCntl](#)

[NNFMgrDeleteSubStringCntl](#)

## NNFMgrGetSubStringCntl

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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrSubStringCntlInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetSubStringCntl.

### 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 265.

### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrSubStringCntlInfo](#)

[NNFMgrCreateSubStringCntl](#)

[NNFMgrUpdateSubStringCntl](#)

[NNFMgrGetSubStringCntl](#)

## NNFMgrUpdateSubStringCntl

Updates an existing substring control in the database. Before calling this function, the `NNFMgrSubStringCntlInfo()` data structure must be initialized with the new values. The `NNFmgrGetSubStringCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateSubStringCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrSubStringCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrSubStringCntlInfo*</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrSubStringCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrSubStringCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetSubStringCntl(pNNFMgr, "first8", &Info );
Strcpy(Info.cntlName, "first10");// change control name
Info.start = 1;
Info.len = 10;
NNFMgrUpdateSubStringCntl(pNNFMgr, "first8", &Info);
```

## See Also

[NNFMgrSubStringCntlInfo](#)

[NNFMgrCreateSubStringCntl](#)

[NNFMgrUpdateSubStringCntl](#)

[NNFMgrGetSubStringCntl](#)

## NNFMgrDeleteSubStringCntl

Deletes a single substring control from the database.

### Syntax

```
const short NNFMgrDeleteSubStringCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is used in other formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrSubStringCntlInfo](#)

[NNFMgrCreateSubStringCntl](#)

[NNFMgrGetSubstringCntl](#)

[NNFMgrUpdateSubStringCntl](#)



## Case Control

### NNFMgrGetCaseCntl

Gets a single Case control from the database.

#### Syntax

```
const short NNFMgrGetCaseCntl(
    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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrCaseCntlInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetCaseCntl.

#### 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 265.

#### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrCaseCntlInfo](#)

## Justify Control

### NNFMgrGetJustifyCntl

Gets a single Justify control from the database. The location of the returned control is determined by the OpCode argument. See *OpCode* on page 265.

#### 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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrJustifyCntlInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetJustifyCntl.

#### Return Value

Return a non-zero integer value on success and zero (0) 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

[NNFMgrJustifyCntlInfo](#)

## Trim Controls

### NNFMgrCreateTrimCntl

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.

#### 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 trimCharLitrName 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 zero (0) 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

[NNFMgrTrimCntlInfo](#)

[NNFMgrGetTrimCntl](#)

[NNFMgrUpdateTrimCntl](#)

[NNFMgrDeleteTrimCntl](#)

## NNFMgrGetTrimCntl

Gets a single Trim control from the database. The location of the returned control is determined by the OpCode argument. See *OpCode* on page 265.

### 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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgr TrimCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetTrimCntl.

### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrTrimCntlInfo](#)

[NNFMgrCreateTrimCntl](#)

[NNFMgrUpdateTrimCntl](#)

[NNFMgrDeleteTrimCntl](#)

## NNFMgrUpdateTrimCntl

Updates an existing Trim control in the database. Before calling this function, the `NNFMgrTrimCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetTrimCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateTrimCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrTrimCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrTrim CntlInfo* const</code>	Input	Pointer to a valid <code>NNFMgrTrimCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.



## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrTrimCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetTrimCntl(pNNFMgr, "trimLeadingBlanks", &Info );
Strcpy(Info.cntlName, "trimBlanks");// change name
Info.trim = BOTH_TRIM;
NNFMgrUpdateTrimCntl(pNNFMgr, "trimLeadingBlanks", &Info);
```

## See Also

[NNFMgrTrimCntlInfo](#)

[NNFMgrCreateTrimCntl](#)

[NNFMgrGetTrimCntl](#)

[NNFMgrDeleteTrimCntl](#)

## NNFMgrDeleteTrimCntl

Deletes a single Trim control from the database.

### Syntax

```
const short NNFMgrDeleteTrimCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrTrimCntlInfo](#)

[NNFMgrCreateTrimCntl](#)

[NNFMgrGetTrimCntl](#)

[NNFMgrDeleteTrimCntl](#)

## Collection Controls

Collection controls can contain zero (0) or more individual controls or collections of controls. The parent collection control is created using `Create()`. Use the `Get` API to get collections. The set of child controls is maintained using the `AddCntlToCollection()` and `GetCntlFromCollection()` APIs.

## NNFMgrCreateCollectionCntl

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
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input/Output	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>pInfo</code>	<code>NNFMgrCollectionCntlInfo* const</code>	Input/Output	Pointer to structure that provides data about <code>NNFMgrCreateCollectionCntl</code> .

### 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 zero (0) 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

[NNFMgrCollectionCntlInfo](#)

[NNFMgrGetCollectionCntl](#)

[NNFMgrAddCntlToCollection](#)

[NNFMgrGetCntlFromCollection](#)

[NNFMgrUpdateCollectionCntl](#)

[NNFMgrDeleteCollectionCntl](#)

## NNFMgrGetCollectionCntl

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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgr Collection CntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetCollectionCntl.

### 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 265.

### Return Value

Return a non-zero integer value on success and zero (0) 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**

[NNFMgrCollectionCntlInfo](#)

[NNFMgrCreateCollectionCntl](#)

[NNFMgrAddCntlToCollection](#)

[NNFMgrGetCntlFromCollection](#)

[NNFMgrUpdateCollectionCntl](#)

[NNFMgrDeleteCollectionCntl](#)

## NNFMgrAddCntlToCollection

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.
SeqNum	int	Input/ Output	Indicates the position where the control is added.
pInfo	const NNFMgrCntl Info* const	Input/ Output	Describes the information for the control.

### Remarks

If SeqNum is less than or equal to zero (0) or greater than the 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 zero (0) 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**

[NNFMgrCollectionCntlInfo](#)

[NNFMgrCreateCollectionCntl](#)

[NNFMgrGetCollectionCntl](#)

[NNFMgrGetCntlFromCollection](#)

[NNFMgrUpdateCollectionCntl](#)

[NNFMgrDeleteCollectionCntl](#)

## NNFMgrGetCntlFromCollection

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.
CollName	const char* const	Input	The name of the collection.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetCntlFromCollection.

### 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 265.

## Return Value

Return a non-zero integer value on success and zero (0) 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

[NNFMgrCollectionCntlInfo](#)

[NNFMgrCreateCollectionCntl](#)

[NNFMgrGetCollectionCntl](#)

[NNFMgrAddCntlToCollection](#)

[NNFMgrUpdateCollectionCntl](#)

[NNFMgrDeleteCollectionCntl](#)

## NNFMgrUpdateCollectionCntl

Updates an existing collection control in the database. Before calling this function, the `NNFMgrCollectionCntlInfo()` data structure must be initialized with the new values. The `NNFMgrGetCollectionCntl` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the control. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateCollectionCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrCollectionCntlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrCollectionCntlInfo* const</code>	Input	Pointer to a valid <code>NNFMgrCollectionCntlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

This API function truncates the list of output controls in the collection; it does not insert output control entries in the list. To add entries to the list, call the `NNFMgrAddCntlToCollection()` function.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrCollectionCntlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetCollectionCntl(pNNFMgr, "coll", &Info );
Strcpy(Info.cntlName,"newColl");// change name
NNFMgrUpdateCollectionCntl(pNNFMgr, "coll", &Info);

NNFMgrCntlInfo Cntl;
NNF_CLEAR(&Cntl);
Strcpy(Expr.cntlName,"substituteBlanks");
Expr.cntlType = SUBSTITUTE_CNTL;
NNFMgrAddCntlToCollection(pNNFMgr, "newColl", 0, &Cntl);
// append it
Strcpy(Expr.cntlName,"appendNewline");
Expr.cntlType = PRE_POST_FIX_CNTL;
NNFMgrAddCntlToCollection(pNNFMgr, "newColl", 0, &Cntl);
// append it
```

## See Also

[NNFMgrCollectionCntlInfo](#)

[NNFMgrCreateCollectionCntl](#)

[NNFMgrGetCollectionCntl](#)

[NNFMgrAddCntlToCollection](#)

[NNFMgrGetCntlFromCollection](#)

[NNFMgrDeleteCollectionCntl](#)

## NNFMgrDeleteCollectionCntl

Deletes a single Collection control from the database.

### Syntax

```
const short NNFMgrDeleteCollectionCntl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrCollectionCntlInfo](#)

[NNFMgrCreateCollectionCntl](#)

[NNFMgrGetCollectionCntl](#)

[NNFMgrAddCntlToCollection](#)

[NNFMgrGetCntlFromCollection](#)

[NNFMgrUpdateCollectionCntl](#)

## Custom Date/Time Formats

### NNFMgrGetDateTimeFormatString

Gets a single date/time format string from the database. Date/time format is one of the standard date/time formats.

#### 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 location of the returned control is determined by the OpCode argument.
customFlag	short	input	Indicates whether the date/time format is a custom format. 1 indicates true; zero (0) indicates false.
pFormatStr	char* const	Input/Output	Pointer to structure that contains data about NNFMgrGetDateTimeFormatString.



## 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 265.

## Return Value

Return a non-zero integer value on success and zero 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 the user is working with compound formats or collections of controls, a parent object might refer back to itself or one of its parent objects. This is called recursion.

## NNFMgrIsRecursiveFormat

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 to be checked for recursion.
IsRecursive	short * const	Output	Indicates whether the format is recursive. 1 indicates true; zero (0) indicates false.

### Remarks

If the format is recursive, the IsRecursive argument is set to 1; otherwise, IsRecursive is set to zero (0).

### Return Value

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

## Example

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. However, Compound Format B consists of Compound Format A, which consists of Compound Format B, and so on. This situation causes an infinite loop when trying to traverse the children of Compound Format A; therefore, Compound Format A is a Recursive Format.

## See Also

[NNFMgrIsRecursiveCollection](#)

## NNFMgrIsRecursiveCollection

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.
CollectionName	const char*	Input/ Output	The name of the collection.
IsRecursive	short * const	Input/ Output	Indicates whether the collection is recursive. 1 indicates true; zero (0) indicates false.

### Remarks

If the collection is recursive, the IsRecursive argument is set to 1; otherwise, IsRecursive is set to zero (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 on failure. Use GetLastError() 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](#)

## NNFMgrUpdateOutputControl

Updates an existing release 3.x output control in the database. Before calling this function, the `NNFMgrOutputControlInfo()` data structure must be initialized with the new values. The `NNFMgrGetOutputControl()` function can be used to populate this data structure with the current values. The `cntlName` parameter must be the current name of the output control. The `pInfo` structure should contain the new name if different from the current name.

### Syntax

```
const short NNFMgrUpdateOutputControl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrOutputControlInfo * const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>cntlName</code>	<code>const char * const</code>	Input	The name of the control.
<code>pInfo</code>	<code>NNFMgrOutputControlInfo * const</code>	Input	Pointer to a valid <code>NNFMgrOutputControlInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This function first deletes the named output control using the `NNFMgrDeleteOutputControl()` API call, then calls `NNFMgrCreateOutputControl()` to create it with the new values. All

references from parent components to this output control are maintained, even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrOutputControlInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetOutputControl(pNNFMgr, "StringWithNewline", &Info );
strcpy(Info.controlName, "StringWithSemicolon");// change name
strcpy(Info.suffix, "Semicolon");// change suffix literal
NNFMgrUpdateOutputControl(pNNFMgr, "StringWithNewline", &Info);
```

## NNFMgrDeleteOutputControl

Deletes a single output control from the database.

### Syntax

```
const short NNFMgrDeleteOutputControl(
    NNFMgr *pNNFMgr,
    const char * const cntlName )
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more Output formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.



---

## Literal Management APIs

Output controls are designed to reuse literal strings as much as possible. The current controls refer to these literal strings as literals. These strings were formerly used primarily for input and output field delimiters and were referred to as delimiters.

Delimiter APIs and structures are supported only for backward compatibility and should not be used for new development. See *Obsolete APIs and Structures* for details about APIs that are no longer used in NEONet.

When a user specifies a literal name, Formatter Management APIs check to see if a literal of this name already exists. If a matching literal with a default literal name can be found, then this literal is reused, and no new literal is created. Only literals with default names, that is, names not specified directly by the user, are considered for reuse.

Users do not provide literal id numbers in the structures passed to the control creation APIs in the form `NNFMgrCreatexxCntl`. Instead, they pass the literal value or the literal name to the API. If a literal value is used that does not already exist, a new literal is created with the value and is 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.

# Literal Management API Structure

## NNFMgrLiteralInfo

NNFMgrLiteralInfo is a structure containing information about 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

Creates a 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	NNFMgrLiteralInfo* const	Input/Output	Pointer to structure which will provide data about NNFMgrCreateLiteral.

### Return Value

Return a non-zero integer value on success, and 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](#)

[NNFMgrUpdateLiteral](#)

[NNFMgrDeleteLiteral](#)

## NNFMgrGetLiteral

Gets a single Literal from the database. The location of the returned control is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 265.

### 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 location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrLiteralInfo* const	Input/Output	Pointer to structure that contains data about NNFMgrGetLiteral.

### Return Value

Return a non-zero integer value on success, and 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](#)

[NNFMgrUpdateLiteral](#)

NNFMgrDeleteLiteral

## NNFMgrUpdateLiteralLiteral

Updates an existing literal in the database. Before calling this function, the `NNFMgrLiteralInfo()` data structure must be initialized with the new values. The `NNFMgrGetLiteral` function can be used to populate this data structure with the current values. The `literalName` parameter must be the current name of the literal. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateLiteral(
    NNFMgr *pNNFMgr,
    const char * const literalName,
    NNFMgrLiteralInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>literalName</code>	<code>const char * const</code>	Input	The name of the literal.
<code>pInfo</code>	<code>NNFMgrLiteralInfo * const</code>	Input	Pointer to a valid <code>NNFMgrLiteralInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

```
NNFMgrLiteralInfo Info;
NNF_CLEAR(&Info);
NNFMgrGetLiteral(pNNFMgr, "fieldDelimiter", &Info );
strcpy(Info.literalValue, ",");// change literal
Info.literalLength = 1;
NNFMgrUpdateLiteral(pNNFMgr, "fieldDelimiter", &Info);
```

## See Also

[NNFMgrCreateLiteral](#)

[NNFMgrGetLiteral](#)

[NNFMgrDeleteLiteral](#)

## NNFMgrDeleteLiteral

Deletes a single literal from the database.

### Syntax

```
const short NNFMgrDeleteLiteral(
    NNFMgr *pNNFMgr,
    const char * const literalName)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
literalName	const char * const	Input	The name of the literal.

### Remarks

This function does not perform any referential integrity checks on the database. If the literal you are deleting is used in other controls or formats, then those components will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.



## **See Also**

[NNFMgrCreateLiteral](#)

[NNFMgrGetLiteral](#)

[NNFMgrUpdateLiteral](#)

[NNFMgrDeleteDelimiter](#)

## NNFMgrDeleteDelimiter

Deletes a single delimiter from the database.

---

### Note:

The NNFMgrDeleteDelimiter function is included only for backward compatibility. You should use the NNFMgrDeleteLiteral for NEONet releases 4.0 and later.

---

### Syntax

```
const short NNFMgrDeleteDelimiter(
    NNFMgr *pNNFMgr,
    const char * const delimiterName )
```

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
delimiterName	const char * const	Input	The name of the delimiter.

### Remarks

This function does not perform any referential integrity checks on the database. If the delimiter you are deleting is used in other controls or formats, then those components will no longer function properly.

### Return Value

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

## **Example**

See *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

NNFMgrDeleteLiteral

---

# User-Defined Data Type Management APIs

---

**Note:**

You can assign user-defined data types *only* to the `data_type` portion of parse and format controls. You cannot assign user-defined data types to the `length_type` or `tag_type` portions of parse or format controls.

---

---

**WARNING!**

If you use 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 you import components exported from a case-sensitive database into a case-insensitive database, the case differences cause NNFIe to fail during import if a conflict arises between two components named the same with only case differences.

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

---

# User-Defined Data Type Management API Structures

## NNFMgrUserDefTypeInfo

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

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

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 NNFMgrUserDefTypeInfo * 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

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 NNFMgrNameValuePairInfo * 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

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 of length 1 to 32 inclusive.
pTypeInfo	const NNFMgrUserDefTypeInfo * const	Output	Information about the user-defined type.

### 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

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 NNFMgrUserDefTypeInfo * const	Output	Information about the user-defined type.

### 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

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.

### 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](#)

## NNFMgrUpdateUserDefinedType

Updates an existing user-defined type in the database. Before calling this function, the `NNFMgrUserDefTypeInfo()` data structure must be initialized with the new values. The `NNFMgrGetUserDefinedType` function can be used to populate this data structure with the current values. The `pInfo` structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateUserDefinedType(
    NNFMgr *pNNFMgr,
    const char * const pTypeName,
    NNFMgrUserDefTypeInfo* const)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>pTypeName</code>	<code>const char * const</code>	Input	The name of the user-defined type.
<code>pInfo</code>	<code>NNFMgrUserDefTypeInfo *</code> <code>const</code>	Input	Pointer to a valid <code>NNFMgrUserDefTypeInfo</code> structure.

### Remarks

A call to `NNF_CLEAR` for `pInfo` should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.



## Return Value

Returns a non-zero integer value on success and zero 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](#)

## NNFMgrDeleteUserDefinedType

Deletes a single user-defined type from the database.

### Syntax

```
const short NNFMgrDeleteUserDefinedType(
    NNFMgr *pNNFMgr,
    const char * const pTypeName )
```

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pTypeName	const char * const	Input	The name of the user-defined type.

### Remarks

This function does not perform any referential integrity checks on the database. If the user-defined type you are deleting is used in other controls or formats, then those components will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrGetUserDefinedType](#)

[NNFMgrGetFirstUserDefinedType](#)

[NNFMgrGetNextUserDefinedType](#)

---

## Parse Control Management APIs

---

---

### **WARNING!**

If you use 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 you import components exported from a case-sensitive database into a case-insensitive database, the case differences will cause NNFiE to fail during import if a conflict arises between two components named the same with only case differences.

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

---

# Parse Control Management API Structures

## NNFMgrParseControlInfo

NNFMgrParseControlInfo is a structure containing parse control information.

### Syntax

```
typedef struct NNFMgrParseControlInfo {
} NNFMgrParseControlInfo;
```

### Parameters

Name	Type	Description
parseName[33]	unsigned char	Name of parse control. NULL-terminated string of 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 acts. 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	Must be a supported data type. Any user-defined data type code defined using User-defined Type Input Field Validation can also be used.

<b>Name</b>	<b>Type</b>	<b>Description</b>
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 one of: TERMINATION_Exact_Length TERMINATION_Minimum_Length_Delimiter 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 tagTermination delimiter: TERMINATION_Delimiter TERMINATION_Minimum_Length_Delimiter
lengthType	short	Same possible values as dataType.
lengthTermination	short	Same possible values as dataTermination.

<b>Name</b>	<b>Type</b>	<b>Description</b>
lengthLength	unsigned	Length of data portion of field if lengthTermination is one of: TERMINATION_Exact_Length TERMINATION_Minimum_Length_Delimiter TERMINATION_Minimum_Length_White_Space
lengthDelimiter[33]	char	Name of delimiter if lengthTermination is one of: TERMINATION_Delimiter 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.
validationParamName[33]	char	String value that user-defined type input field validation code pass to the Callback RuntimeDataLookup function.
userDefInValNameValuePairArray	NameValuePair*	An array of name/value pairs to be associated with the 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	To use custom date/time formats, set dataAttr to your custom date/time format string. This field is ignored when using standard date/time formats.

<b>Name</b>	<b>Type</b>	<b>Description</b>
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.
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 is prefixed with 19. If a two-digit year < year cutoff, it is 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 (0). By default, yearCutoff is always set to 0 in the call to NNF_CLEAR.

## Remarks

The Date/Time fields are only used when the user specifies a dataType of date, time, default date/time, or custom date/time. If you use custom date/time formats, set dataAttr to your custom date/time format string. 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.

---

### Note:

All mandatory fields must parse correctly and have valid data for the specified dataType. Optional fields do not have to parse successfully.

---



---

**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  $\geq$  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 0. By default, `yearCutoff` is always set to 0 in the call to `NNF_CLEAR`.

---

**See Also**

See Appendix D: *Data Type Descriptions* on page 509.

# Parse Control Management APIs

## NNFMgrCreateParseControl

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().
pParseControlInfo	const NNFMgrParseControlInfo *	Input	Information about the parse control to add.

### 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() fails if dataTermination is not exact\_length or dataLength is not between 1 and 16. It also fails 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 or  
DATA\_TYPE\_IBM\_Signed\_Packed\_Integer.

The Date/Time fields are only used when the user specifies a dataType of date, time, default date/time, or custom date/time. If you use custom date/time formats, set dataAttr to your custom date/time format string. 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.

## 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

[NNFMgrParseControlInfo](#)

[NNFMgrGetParseControl](#)

[NNFMgrGetFirstParseControl](#)

[NNFMgrGetNextParseControl](#)

[NNFMgrUpdateParseControl](#)

[NNFMgrDeleteParseControl](#)

---

**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  $\geq$  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 0. By default, `yearCutoff` is always set to 0 in the call to `NNF_CLEAR`.

---

## NNFMgrGetParseControl

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 of length 1 to 32 inclusive.
pParseControlInfo	NNFMgrParseControlInfo * const	Output	Information about the parse control.

### Return Value

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

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

**See Also**

[NNFMgrParseControlInfo](#)

[NNFMgrCreateParseControl](#)

[NNFMgrGetFirstParseControl](#)

[NNFMgrGetNextParseControl](#)

[NNFMgrUpdateParseControl](#)

[NNFMgrDeleteParseControl](#)

## NNFMgrGetFirstParseControl

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().
pParseControlInfo	NNFMgrParseControlInfo * const	Output	Information about the parse control.

### Return Value

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

Use GetLastError() to retrieve the number for the error that occurred; then use GetMessage() to retrieve the error message associated with that error number.

## **See Also**

[NNFMgrParseControlInfo](#)

[NNFMgrCreateParseControl](#)

[NNFMgrGetParseControl](#)

[NNFMgrGetNextParseControl](#)

[NNFMgrUpdateParseControl](#)

[NNFMgrDeleteParseControl](#)



## NNFMgrGetNextParseControl

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().
pParseControlInfo	NNFMgrParseControlInfo * const	Output	Information about the parse control.

### Return Value

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

Use GetLastError() to retrieve the number for the error that occurred; then use GetMessage() to retrieve the error message associated with that error number.

## **See Also**

[NNFMgrParseControlInfo](#)

[NNFMgrCreateParseControl](#)

[NNFMgrGetParseControl](#)

[NNFMgrGetFirstParseControl](#)

[NNFMgrUpdateParseControl](#)

[NNFMgrDeleteParseControl](#)

## NNFMgrUpdateParseControl

Updates an existing parse control in the database. Before calling this function, the NNFMgrParseControlInfo() data structure must be initialized with the new values. The NNFMgrGetParseControl function can be used to populate this data structure with the current values. The cntlName parameter must be the current name of the literal. The pInfo structure should contain the new control name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateParseControl(
    NNFMgr *pNNFMgr,
    const char * const cntlName,
    NNFMgrParseControlInfo* const pInfo)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.
pInfo	NNFMgrParseControlInfo * const	Input	Pointer to a valid NNFMgrParseControlInfo structure.

### Remarks

A call to NNF\_CLEAR for pInfo should be made prior to populating the structure or calling this API.

This API function maintains all references from parent components even if the control's name is changed.

## Return Value

Returns a non-zero integer value on success and zero 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

[NNFMgrParseControlInfo](#)

[NNFMgrCreateParseControl](#)

[NNFMgrGetParseControl](#)

[NNFMgrGetFirstParseControl](#)

[NNFMgrGetNextParseControl](#)

[NNFMgrDeleteParseControl](#)

## NNFMgrDeleteParseControl

Deletes a single Input (Parse) Control from the database.

### Syntax

```
const short NNFMgrDeleteParseControl(
    NNFMgr *pNNFMgr,
    const char * const cntlName)
```

### Parameters

Name	Type	Input/Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
cntlName	const char * const	Input	The name of the control.

### Remarks

This function does not perform any referential integrity checks on the database. If the control you are deleting is still being used in one or more Input formats, then those formats will no longer function properly.

### Return Value

Returns a non-zero integer value on success and zero 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 *Using Format Management APIs: fmgr.cpp* on page 475.

## **See Also**

[NNFMgrParseControlInfo](#)

[NNFMgrCreateParseControl](#)

[NNFMgrGetParseControl](#)

[NNFMgrGetFirstParseControl](#)

[NNFMgrGetNextParseControl](#)

[NNFMgrUpdateParseControl](#)

[NNFMgrDeleteParseControl](#)

---

## Format Management APIs

The format management API structures are used to create or get controls. This section details the following output control API structures:

- NNFMgrFormatInfo
- NNFMgrRepeatFormatInfo
- NNFMgrFlatFormatInfo
- NNFMgrInFieldInfo
- NNFMgrOutfieldInfo

---

### **WARNING!**

If you use 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 you import components exported from a case-sensitive database into a case-insensitive database, the case differences will cause NNFIe to fail during import if a conflict arises between two components named the same with only case differences.

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

---

# Format Management API Structures

## NNFMgrFormatInfo

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 format is flat, set inputInd=1; and compoundInd can be 0, 1, 2, or 3. If format is compound, set inputInd=0; and compoundInd can be 0, 1, or 3. 1=IN_COMPOUND_ORDINAL 2=IN_COMPOUND_TAGGED 3=IN_COMPOUND_ALTERNATIVE 1=OUT_COMPOUND_ORDINAL 3=OUT_COMPOUND_ALTERNATIVE
initFlag	long	Uninitialized structure check value.



## NNFMgrRepeatFormatInfo

NNFMgrRepeatFormatInfo is a structure containing repeating format information.

### Syntax

```
typedef struct NNFMgrRepeatFormatInfo {
    char parentFormatName[33];
    char childFormatName[33];
    unsigned char optionalInd;
    unsigned char repeatInd;
    int repeatTermination;
    char repeatDelimiter[33];
    unsigned repeatCount;
    char repeatFieldName[33];
    long initFlag;
};
```

### Parameters

Name	Type	Description
parentFormatName	char[33]	Name of parent Format.
childFormatName	char[33]	Name of child format. Must be a NULL-terminated string 1 to 32 characters long, inclusive.
optionalInd	unsigned char	Set to zero (0) for a mandatory component, and 1 for an optional component.
repeatInd	unsigned char	If repeatInd in NNFMgrRepeatFormatInfo is zero, then the format is <b>not</b> repeating. Any number other than 0 indicates that the format is repeating.

<b>Name</b>	<b>Type</b>	<b>Description</b>
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[NAME_LENGTH+1]	char[33]	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	unsigned char[33]	Name of repeating field. NULL-terminated string length 1 to 32 inclusive.
initFlag	long	Uninitialized structure check value.

## NNFMgrFlatFormatInfo

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

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

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.

<b>Name</b>	<b>Type</b>	<b>Description</b>
accessMode	short	<p>One of:</p> <p>ACCESS_MODE_Not_Applicable</p> <p>ACCESS_MODE_Normal_Access</p> <p>ACCESS_MODE_Access_with_Increment</p> <p>ACCESS_MODE_Reset_then_Normal_Access</p> <p>ACCESS_MODE_Reset_then_Access_with_Increment</p> <p>ACCESS_MODE_Access_nth_Instance_of_Field</p> <p>ACCESS_MODE_Access_within_Compound</p> <p>ACCESS_MODE_Cycling_Access_stay_in_Compound</p> <p>ACCESS_MODE_Access_using_Relative_Index</p> <p>ACCESS_MODE_Create_Field</p> <p>It is an error to provide a subscript value when accessMode is not ACCESS_MODE_Access_nth_Instance_of_Field.</p>
subscript	short	
initFlag	unsigned short	Uninitialized structure check value.

# Format Management APIs

## NNFMgrCreateFormat

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 are not supported by this API. Both are 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 NNFMgrFormatInfo * const	Input	Pointer to a valid NNFMgrFormatInfo structure. This pointer cannot be NULL.
pFlatFormatInfo	const NNFMgrFlatFormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure.

## 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

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.

### 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

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.

### 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

NNFMgrAppendFormatToFormat() adds a flat or compound format to a compound format. The child format is added after all other child formats.

parentFormatName is the name of an existing compound format.

childFormatName is 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 * const pRepeatFormatInfo);
```

### 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.
pRepeatFormatInfo	const NNFMgrRepeatFormatInfo * const	Input	Pointer to insertion information. This pointer may not be NULL.

### 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

NNFMgrGetFormat() reads information about a format; whether input, output, flat, or compound. 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 to retrieve information for.
pFormatInfo	NNFMgrFormatInfo * const	Output	Pointer to a valid NNFMgrFormatInfo structure. pFormatInfo cannot be NULL. Structure fields are filled by database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlatFormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure.

## 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

NNFMgrGetFirstFormat() reads information about the first format; whether input, output, flat, or compound. 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	NNFMgrFormatInfo * const	Output	Pointer to a valid NNFMgrFormatInfo structure. pFormatInfo cannot be NULL. Structure fields is filled by database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlatFormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure.

## 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

NNFMgrGetNextFormat() reads information about all formats except the first input, 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	NNFMgrFormatInfo * const	Output	Pointer to a valid NNFMgrFormatInfo structure. pFormatInfo cannot be NULL. Structure fields will be filled by in-database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlatFormatInfo * const	Input	Pointer to a valid NNFMgrFlatFormat structure.

### 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

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	NNFMgrInFieldInfo * const	Output	Information about the field.

### Return Value

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

Use GetLastError() 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

NNFMgrGetNextFieldFromInputFormat() retrieves field information for all fields except 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	NNFMgrInFieldInfo * const	Output	Information about the field.

### Return Value

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

Use GetLastError() 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

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.

### 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](#)

## NNFMgrGetNextFieldFromOutputFormat

NNFMgrGetNextFieldFromOutputFormat() retrieves field information for all fields except 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	NNFMgrOutFieldInfo * const	Output	Information about the field.

### Return Value

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

Use GetLastError() 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

NNFMgrGetFirstChildFormat() gets 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.

### 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

NNFMgrGetNextChildFormat() gets details about all formats except 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().
pRepeatFormatInfo	NNFMgrRepeatFormatInfo * const	Output	Repetition information structure.

### Return Value

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

Use GetLastError() 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](#)

## NNFMgrUpdateFormat

Updates an existing input or output format in the database. Before calling this function, the `NNFMgrFormatInfo()` and `NNFMgrFlatFormatInfo()` data structures must be initialized with the new values. The `NNFMgrGetFormat()` function can be used to populate these data structures with the current values. The `fmtName` parameter must be the current name of the format. The `pFormatInfo` structure should contain the new name if it is different from the current name.

### Syntax

```
const short NNFMgrUpdateFormat(
    NNFMgr *pNNFMgr,
    const char * const fmtName,
    const NNFMgrFormatInfo * const pFormatInfo,
    const NNFMgrFlatFormatInfo * const pFlatInfo)
```

### Parameters

Name	Type	Input/ Output	Description
<code>pNNFMgr</code>	<code>NNFMgr*</code>	Input	Valid <code>NNFMgr</code> instance pointer previously returned by <code>NNFMgrInit</code> .
<code>fmtName</code>	<code>const char * const</code>	Input	The name of the format.
<code>pFormatInfo</code>	<code>const NNFMgrFormatInfo * const</code>	Input	Pointer to a valid <code>NNFMgrFormatInfo</code> structure This pointer cannot be NULL.
<code>pFlatInfo</code>	<code>const NNFMgrFlatFormatInfo * const</code>	Input	Pointer to a valid <code>NNFMgrFlatFormatInfo</code> structure.

## 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.

This function first deletes the named format using the `NNFMgrDeleteFormat` API call; then calls `NNFMgrCreateFormat` to create the format with the new values. All references from parent formats to this format are maintained.

## Return Value

Returns 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

```
NNFMgrFormatInfo FmtInfo;
NNFMgrFlatInfo FlatInfo;
NNF_CLEAR(&FmtInfo);
NNF_CLEAR(&FlatInfo);
NNFMgrGetFormat(pNNFMgr, "FlatFormat_1", &FmtInfo, &FlatInfo );
strcpy(FmtInfo.formatName, "FF_Unordered");
// change name
FlatInfo.decomposition = IN_FORMAT_DECOMP_Unordered;
// change to unordered fields
NNFMgrUpdateFormat(pNNFMgr, "FlatFormat_1", &FmtInfo,
                  &FlatInfo);
```

## See Also

[NNFMgrUpdateOutputControl](#)

[NNFMgrUpdateParseControl](#)

## NNFMgrDeleteFormat

Deletes a single input or output format from the database.

### Syntax

```
const short NNFMgrDeleteFormat(
    NNFMgr *pNNFMgr,
    const char * const fmtName)
```

### Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
fmtName	const char * const	Input	The name of the format.

### Remarks

This function does not perform any referential integrity checks on the database. If the format you are deleting is used in other compound formats, those formats will no longer function properly.

### Return Value

Returns 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

See *Using Format Management APIs: fmgr.cpp* on page 475.



## **See Also**

[NNFMgrDeleteOutputControl](#)

[NNFMgrDeleteParseControl](#)

---

# Format Management API Error Handling

## GetErrorNo

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

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

Code	Name	Message	Explanation	Response
	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 technical 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 technical 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 technical support.

<b>Code</b>	<b>Name</b>	<b>Message</b>	<b>Explanation</b>	<b>Response</b>
	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, User Callbacks, and Format Management APIs are used.

---

## Using Formatter APIs to Reformat a Message: msgtest.cpp

```
// 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. 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(), for example, 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 resulting output
// message to the output file.
//
// The driver exercises the following APIs:
//
```

## Appendix A

```
// 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.
    DbmsSession *Session = OpenDbmsSession("new_format_demo",
                                           NN_DBMS_TYPE);

    // 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("constructor", 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 you want.

    // 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 you want.

    // Reformat the input messages. (Internally, this calls
    // Formatter::Parse prior to generating the output messages.
    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->GetError

```

```

        Message()
        << "."
        << endl << endl;
    cerr << "Hit return." << endl;
    cin.get();
    }
    }
    }
    }
    inFile.close();
}
delete [] pInFile;
CloseDbmsSession(Session);
}

```

## GetValidationCallbacks Function: getval.cpp #1

Use the following function stub to run msgtest.cpp without validation callbacks.

```

#include "nuserfunction.h"

NNFunctionKeyPairCollection* GetValidationCallbacks();

NNFunctionKeyPairCollection* GetValidationCallbacks()
{
    return 0;
}

```

## GetValidationCallbacks Function: getval.cpp #2

Use the following function stub to run msgtest.cpp with validation callbacks.

```

// System include files

extern "C" {
#include <stdio.h>
#include <string.h>

```

## Appendix A

```
#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 "nuserfunction.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;
}

```

---

## Traversing a Parsed Message: apitest.cpp

This example illustrates how to traverse the structure of a parsed message.

```
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)
    intcode;

    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;
    char buffer[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()
{
    intmsgCount, msgIndex;
    ParsedMessage *pParsedMessage;

    DbmsSession *Session = OpenDbmsSession("new_format_demo",
                                           NN_DBMS_TYPE);

    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 inFile[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(inFile);
    if (!*inFile) break;

    ifstream inFile(inFile);
    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 you want.

    // 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;
}

```



---

## Using Format Management APIs: fmgr.cpp

The following program illustrates the use of Format Management APIs. The user can control the invocation of each of the APIs.

```

#if defined(oracle)
extern "C"{
#    include <oratypes.h>
#    include <ocidfn.h>
#    include <ocidem.h>
#    include <ociapr.h>
}
#include "orases.h"
#endif

#include "dbtypes.h"
#include "nnfmgr.h"
#include "ses.h"
#include "sqlapi.h"
#include <errno.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

void
usage(const char * const progname) {
    cerr << "usage: " << progname << " sessionName" << endl;
    exit(EINVAL);
}

void
do_help() {
    cout << "\
help or ? -- this screen\n\
quit or q -- exit the program\n\
cfi FormatName -- create flat input format named FormatName\n\

```

```

cfo FormatName -- create flat output format named FormatName\n\
cfic FormatName -- create compound input format named FormatName\n\
cfoc FormatName -- create compound output format named FormatName\n\
cf FieldName [Description] -- create new field named FieldName\n\
cd DelimiterName Value -- create new delimiter named DelimiterName\n\
gf FormatName -- get format FormatName\n\
gd DelimiterName -- get a delimiter named DelimiterName\n\
gpc ControlName -- get a parse control named ControlName\n\
goc ControlName -- get an output control named ControlName\n\
aif Parent Child ParseControl -- append field Child to flat input
                                format Parent\n\
aof Parent Child OutputFormat -- append field Child to flat output
                                format Parent\n\
acf Parent Child -- append format Child to compound format Parent\n\
laf -- list all formats\n\
lcf FormatName -- list child formats in FormatName\n\
lffo FormatName -- list child fields in flat output format
                FormatName\n\
lffi FormatName -- list child fields in flat input format
                FormatName\n\
lm ControlName -- list math expressions associated with
                ControlName\n\
ll ControlName -- list lookup entries associated with ControlName\n\
ld -- list all delimiters\n\
lf -- list all fields in db\n\
lpc -- list all parse controls\n\
loc -- list all output controls\n\
coc ControlName -- create output control\n\
cpc ControlName -- create parse control\n\
cle LookupName [data] -- create lookup entry\n\
df FormatName -- delete format FormatName\n\
doc OutputControlName -- delete output control FormatControlName\n\
dpc ParseControlName -- delete parse control ParseControlName\n\
dd DelimiterName -- delete delimiter DelimiterName\n\
dfld FieldName -- delete field FieldName\n\
\n\
" << flush;
}

ostream & operator <<(ostream &s, NNFMgrFormatInfo &i) {
    return cout << i.formatName << ',' << (int) i.inputInd << ',' <<
        << (int)

```

```

i.compoundInd ;
}

ostream & operator <<(ostream &s, NNFMgrFlatFormatInfo &i) {
    return cout << i.decomposition << ',' << i.length << ','
        << i.termination << ',' << i.delimiter;
}

ostream & operator <<(ostream &s, NNFMgrRepeatFormatInfo &i) {
    return s << i.parentFormatName << ',' << i.childFormatName << ','
        << (int) i.optionalInd << ',' << (int) i.repeatInd << ','
        << i.repeatTermination << ',' << i.repeatDelimiter << ','
        << i.repeatCount << ',' << i.repeatCount ;
}

ostream & operator <<(ostream &s, NNFMgrOutFieldInfo &i) {
    return s << i.formatName << ',' << i.fieldName << ','
        << i.controlName << ',' << i.accessMode << ','
        << i.subscript ;
}

ostream & operator <<(ostream &s, NNFMgrInFieldInfo &i) {
    return s << i.formatName << ',' << i.fieldName << ','
        << i.controlName ;
}

ostream & operator <<(ostream &s, NNFMgrMathExpressionInfo &i) {
    return s << i.outputControlName << ',' << i.decimal_precision
        << ',' << i.rounding_mode << ',' << i.expression ;
}

ostream & operator <<(ostream &s, NNFMgrLookupEntryInfo &i) {
    return s << i.controlName << ',' << i.inputValue << ','
        << i.inputValueLen << ',' << i.outputValue << ','
        << i.outputValueLen << ',' << i.outputValueType ;
}

ostream & operator <<(ostream &s, NNFMgrFieldInfo &i) {
    return s << i.fieldName << ':' << i.fieldDescription;
}

ostream & operator <<(ostream &s, NNFMgrOutputControlInfo &i) {

```

```

char c = i.defaultValue[i.defaultLength];
i.defaultValue[i.defaultLength] = '\\0';
s
    << i.controlName << ', '
    << (int) i.optionalInd << ', '
    << (int) i.tagBeforeLengthInd << ', '
    << i.fieldType << ', '
    << i.dataType << ', '
    << i.dataLength << ', '
    << i.tagType << ', '
    << i.tagValue << ', '
    << i.lengthType << ', '
    << (char *) i.defaultValue << ', '
    << i.defaultLength << ', '
    << i.exitRoutine << ', '
    << i.operationType << ', '
    << i.prefix << ', '
    << i.suffix << ', '
    << i.nullAction
    ;

if (i.fieldType == OUTFIELD_FORMAT_Substring)
    {
        s
            << ', '
            << i.substring_start << ', '
            << i.substring_length
            ;

        i.defaultValue[i.defaultLength] = c;
        return s;
    }

ostream & operator <<(ostream &s, NNFMgrParseControlInfo &i) {
    return s << i.parseName << ', ' << i.optionalInd << ', '
        << i.fieldType << ', ' << i.dataType << ', '
        << i.dataTermination << ', ' << i.dataDelimiter << ', '
        << i.dataLength << ', ' << i.tagType << ', '
        << i.tagTermination << ', ' << i.tagLength << ', '
        << (char *) i.tagValue << ', ' << i.tagDelimiter << ', '
        << i.lengthType << ', ' << i.lengthTermination << ', '
        << i.lengthLength << ', ' << i.lengthDelimiter;
}

```

```

}

ostream & operator <<(ostream &s, NNFMgrDelimiterInfo &i) {
    char c = i.delimiterValue[i.delimiterLength];
    i.delimiterValue[i.delimiterLength] = '\\0';

    s << i.delimiterName << ',' << i.delimiterLength << ','
      << (char *) i.delimiterValue;
    i.delimiterValue[i.delimiterLength] = c;
    return s;
}

int
main(int argc,char *argv[]) {
    char * sessionName;

    if(argc != 2) {
        usage(argv[0]);
    }
    sessionName=argv[1];

    cout << "Opening Dbms session " << sessionName << endl;

    DbmsSession *session = OpenDbmsSession(sessionName, NN DBMSTYPE);

    if ( !session ){
        cerr << "No session was created.\n";
        cin.get();
        exit(101);
    }

    if ( !session->Ok() ){
        cout << "session was bad" << endl;
        cerr << "Run Debugging" << endl;
        session->debug();
        cerr << "Debug ends" << endl;
#ifdef DOS
        cin.get();
#endif
        exit(101);
    }
    NNFMgr *fmgr = NNFMgrInit(session);

```

```

char buf[256];
for(;;) {
    cout << "]" << flush;
    if(!cin.getline(buf, 255).good()) {
        break;
    }
    if(!buf[0])
        continue;
    BeginXact(session);

    char *cmd = strtok(buf, " \t");
    char *arg1 = strtok(NULL, " \t");
    char *arg2 = strtok(NULL, " \t");
    char *arg3 = strtok(NULL, " \t");

    if(!arg2)
        arg2 = "";

    if(!arg3)
        arg3 = "";

    if(!strcmp(cmd, "acf")) {
        cout << "***** Append format '" << arg2 << "' to format '"
            << arg1 << '\'" << endl;
        NNFMgrRepeatFormatInfo info;
        NNF_CLEAR(&info);
        strcpy(info.childFormatName, arg2);
        if(NNFMgrAppendFormatToFormat(fmgr, arg1, &info)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED: "
                << NNFMgrGetErrorMessage(fmgr) << " *****"
                << endl;

            RollbackXact(session);
        }
        continue;
    }
    if(!strcmp(cmd, "aif")) {

```

```

cout << "***** Append field '" << arg2
      << "' parse control '" << arg3
      << "' to input format '" << arg1
      << '\'' << endl;
NNFMgrInFieldInfo info;
NNF_CLEAR(&info);
strcpy(info.fieldName, arg2);
strcpy(info.controlName, arg3);
if(NNFMgrAppendFieldToInputFormat(fmgr, arg1, &info)) {
    cout << "***** SUCCESS *****" << endl;
    CommitXact(session);
} else {
    cout << "***** FAILED : "
          << NNFMgrGetErrorMessage(fmgr) << "*****"
          << endl;
    RollbackXact(session);
}
continue;
}
if(!strcmp(cmd, "aof")) {
    cout << "***** Append field '" << arg2
          << "' parse control '" << arg3
          << "' to output format '" << arg1 << '\'' << endl;
    NNFMgrOutFieldInfo info;
    NNF_CLEAR(&info);
    strcpy(info.fieldName, arg2);
    strcpy(info.controlName, arg3);
    if(NNFMgrAppendFieldToOutputFormat(fmgr, arg1, &info)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
              << NNFMgrGetErrorMessage(fmgr) << "*****"
              << endl;
        RollbackXact(session);
    }
    continue;
}
if(!strcmp(cmd, "cfi")) {
    cout << "***** Create an input flat format named '"
          << arg1 << "' *****" << endl;
    NNFMgrFormatInfo info;

```

```

NNF_CLEAR(&info);
strcpy(info.formatName, arg1);
info.inputInd = 1;
info.compoundInd = 0;

if(NNFMgrCreateFormat(fmgr,&info,0)) {
    cout << "***** SUCCESS *****" << endl;
    CommitXact(session);
} else {
    cout << "***** FAILED : "
         << NNFMgrGetErrorMessage(fmgr) << "*****"
         << endl;
    RollbackXact(session);
}
continue;
}
if(!strcmp(cmd, "df")) {
    cout << "***** Delete format named '" << arg1 << "' *****"
         << endl;
    if(NNFMgrDeleteFormat(fmgr,arg1)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
             << NNFMgrGetErrorMessage(fmgr) << "*****"
             << endl;
        RollbackXact(session);
    }
    continue;
}
if(!strcmp(cmd, "doc")) {
    cout << "***** Delete output control named '" << arg1
         << "' *****" << endl;
    if(NNFMgrDeleteOutputControl(fmgr,arg1)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
             << NNFMgrGetErrorMessage(fmgr) << "*****"
             << endl;
        RollbackXact(session);
    }
}

```



```

        continue;
    }
    if(!strcmp(cmd, "dpc")) {
        cout << "***** Delete parse control named '" << arg1
            << "' *****" << endl;
        if(NNFMgrDeleteParseControl(fmgr,arg1)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) << "*****"
                << endl;
            RollbackXact(session);
        }
        continue;
    }
    if(!strcmp(cmd, "dd")) {
        cout << "***** Delete delimiter named '" << arg1
            << "' *****" << endl;
        if(NNFMgrDeleteDelimiter(fmgr,arg1)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) << "*****"
                << endl;
            RollbackXact(session);
        }
        continue;
    }
    if(!strcmp(cmd, "dfld")) {
        cout << "***** Delete field named '" << arg1 << "' *****"
            << endl;
        if(NNFMgrDeleteField(fmgr,arg1)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) <<"*****"
                << endl;
            RollbackXact(session);
        }
    }
}

```

```

        continue;
    }
    if(!strcmp(cmd,"cfic")) {
        cout << "***** Create an input compound format named '"
            << arg1 << "' *****" << endl;
        NNFMgrFormatInfo info;
        NNF_CLEAR(&info);
        strcpy(info.formatName, arg1);
        info.inputInd = 1;
        info.compoundInd = 1;

        if(NNFMgrCreateFormat(fmgr,&info,0)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) << "*****"
                << endl;
            RollbackXact(session);
        }
        continue;
    }
    if(!strcmp(cmd,"cfoc")) {
        cout << "***** Create an output compound format named '"
            << arg1 << "' *****" << endl;
        NNFMgrFormatInfo info;
        NNF_CLEAR(&info);
        strcpy(info.formatName, arg1);
        info.inputInd = 0;
        info.compoundInd = 1;

        if(NNFMgrCreateFormat(fmgr,&info,0)) {
            cout << "***** SUCCESS *****" << endl;
            CommitXact(session);
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) << "*****"
                << endl;
            RollbackXact(session);
        }
        continue;
    }
}

```

```

if(!strcmp(cmd,"cfo")) {
    cout << "***** Create an output flat format named '"
        << arg1 << "' *****" << endl;
    NNFMgrFormatInfo info;
    NNF_CLEAR(&info);
    strcpy(info.formatName, arg1);
    info.inputInd = 0;
    info.compoundInd = 0;

    if(NNFMgrCreateFormat(fmgr,&info,0)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
            << NNFMgrGetErrorMessage(fmgr) << "*****"
            << endl;
        RollbackXact(session);
    }
    continue;
}
if(!strcmp(cmd, "laf")) {
    cout << "***** List all formats *****" << endl;
    NNFMgrFormatInfo info;
    NNFMgrFlatFormatInfo flat_info;
    if(NNFMgrGetFirstFormat( fmgr,&info, &flat_info )) {
        cout << info << ':' << flat_info << endl;
        while(NNFMgrGetNextFormat(fmgr,&info, &flat_info)) {
            cout << info << ':' << flat_info << endl;
        }
    }
    continue;
}
if(!strcmp(cmd, "lcf")) {
    cout << "***** List child formats in format " << arg1
        << " *****" << endl;
    NNFMgrRepeatFormatInfo info;
    if(NNFMgrGetFirstChildFormat( fmgr,arg1,&info)) {
        cout << info << endl;
        while(NNFMgrGetNextChildFormat(fmgr,&info)) {
            cout << info << endl;
        }
    }
}

```

```

        continue;
    }
    if(!strcmp(cmd, "lffi")) {
        cout << "***** List input format fields for format "
             << arg1 << " *****" << endl;
        NNFMgrInFieldInfo info;
        if(NNFMgrGetFirstFieldFromInputFormat(fmgr,arg1,&info)) {
            cout << info << endl;
            while(NNFMgrGetNextFieldFromInputFormat(fmgr,&info)) {
                cout << info << endl;
            }
        }
        continue;
    }
    if(!strcmp(cmd, "lffo")) {
        cout << "***** List all output format fields for format "
             << arg1 << " *****" << endl;
        NNFMgrOutFieldInfo info;
        if(NNFMgrGetFirstFieldFromOutputFormat(fmgr,arg1,&info)) {
            cout << info << endl;
            while(NNFMgrGetNextFieldFromOutputFormat(fmgr,&info)){
                cout << info << endl;
            }
        }
        continue;
    }
    if(!strcmp(cmd, "lm")) {
        cout << "***** List all math expressions for control "
             << arg1 << " *****" << endl;
        NNFMgrMathExpressionInfo info;
        if(NNFMgrGetFirstMathExpression(fmgr,arg1,&info)) {
            cout << info << endl;
            while(NNFMgrGetNextMathExpression(fmgr,&info)) {
                cout << info << endl;
            }
        }
        continue;
    }
    if(!strcmp(cmd, "ll")) {
        cout << "***** List all lookup entries for control "
             << arg1 << " *****" << endl;
        NNFMgrLookupEntryInfo info;

```

```

        if(NNFMgrGetFirstLookupEntry(fmgr, arg1, &info)) {
            cout << info << endl;
            while(NNFMgrGetNextLookupEntry(fmgr, &info)) {
                cout << info << endl;
            }
        }
        continue;
    }
    if(!strcmp(cmd, "loc")) {
        cout << "***** List all output controls *****" << endl;
        NNFMgrOutputControlInfo info;
        if(NNFMgrGetFirstOutputControl(fmgr, &info)) {
            cout << info << endl;
            while(NNFMgrGetNextOutputControl(fmgr, &info)) {
                cout << info << endl;
            }
        }
        continue;
    }
    if(!strcmp(cmd, "goc")) {
        cout << "***** Get an output control named '" << arg1
            << "'" << endl;
        NNFMgrOutputControlInfo info;
        if(NNFMgrGetOutputControl(fmgr, arg1, &info)) {
            cout << info << endl;
        } else {
            cout << "***** FAILED : "
                << NNFMgrGetErrorMessage(fmgr) << "*****"
                << endl;
        }
        continue;
    }
    if(!strcmp(cmd, "lpc")) {
        cout << "***** List all parse controls *****" << endl;
        NNFMgrParseControlInfo info;
        if(NNFMgrGetFirstParseControl(fmgr, &info)) {
            cout << info << endl;
            while(NNFMgrGetNextParseControl(fmgr, &info)) {
                cout << info << endl;
            }
        }
        continue;
    }

```

```

}
if(!strcmp(cmd, "gf")) {
    cout << "***** Get a format named '" << arg1 << "'
        << endl;
    NNFMgrFormatInfo info;
    NNFMgrFlatFormatInfo flat_info;
    if(NNFMgrGetFormat(fmgr, arg1, &info, &flat_info)) {
        cout << info << ':' << flat_info << endl;
    } else {
        cout << "***** FAILED : "
            << NNFMgrGetErrorMessage(fmgr) << "*****"
            << endl;
    }
    continue;
}
if(!strcmp(cmd, "gpc")) {
    cout << "***** Get a parse control named '" << arg1
        << "' << endl;
    NNFMgrParseControlInfo info;
    if(NNFMgrGetParseControl(fmgr, arg1, &info)) {
        cout << info << endl;
    } else {
        cout << "***** FAILED : "
            << NNFMgrGetErrorMessage(fmgr) << "*****"
            << endl;
    }
    continue;
}
if(!strcmp(cmd, "cle")) {
    cout << "***** Create a lookup entry for '" << arg1
        << "' << endl;
    NNFMgrLookupEntryInfo info;
    NNF_CLEAR(&info);
    strcpy(info.controlName, arg1);
    if(*arg2) {
        strcpy((char *) info.inputValue, arg2);
        info.inputValueLen = strlen(arg2);
    } else {
        strcpy((char *) info.inputValue, "xxx");
        info.inputValueLen = 3;
    }
    if(*arg3) {

```

```

        strcpy((char *) info.outputValue, arg3);
        info.outputValueLen = strlen(arg3);
    } else {
        strcpy((char *) info.inputValue, "yyy");
        info.inputValueLen = 3;
    }
    if(NNFMgrAppendOutputLookupEntry(fmgr, arg1, &info)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
             << NNFMgrGetErrorMessage(fmgr) << "*****" << endl;
        RollbackXact(session);
    }
    continue;
}

if(!strcmp(cmd, "coc")) {
    cout << "***** Create an output control named '" << arg1
         << "'" << endl;
    NNFMgrOutputControlInfo info;
    NNF_CLEAR(&info);
    strcpy(info.controlName, arg1);
    info.optionalInd = 0;
    info.fieldType = 0;

    if(NNFMgrCreateOutputControl(fmgr, &info)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
             << NNFMgrGetErrorMessage(fmgr) << "*****"
             << endl;
        RollbackXact(session);
    }
    continue;
}

if(!strcmp(cmd, "cpc")) {
    cout << "***** Create a parse control named '" << arg1
         << "'" << endl;
    NNFMgrParseControlInfo info;
    NNF_CLEAR(&info);

```

```

strcpy(info.parseName, arg1);
info.optionalInd = 0;
info.fieldType = INFIELDD_PARSE_Data_Only;

info.dataType = DATA_TYPE_Ascii_String;
info.dataTermination = TERMINATION_Exact_Length;
info.dataLength = 3;

info.tagType = DATA_TYPE_Ascii_String;
info.tagTermination =
    TERMINATION_Minimum_Length_Delimiter;
info.tagLength = 3;
strcpy((char *) info.tagValue, "xxx");
strcpy(info.tagDelimiter, "pound");

info.lengthType = DATA_TYPE_Ascii_Numeric;
info.lengthTermination = TERMINATION_Delimiter;
info.lengthLength = 0;
strcpy(info.lengthDelimiter, "pound");

if(NNFMgrCreateParseControl(fmgr,&info)) {
    cout << "***** SUCCESS *****" << endl;
    CommitXact(session);
} else {
    cout << "***** FAILED : "
         << NNFMgrGetErrorMessage(fmgr) << "*****"
         << endl;
    RollbackXact(session);
}
continue;
}
if(!strcmp(cmd, "ld")) {
    cout << "***** List delimiters *****" << endl;
    NNFMgrDelimiterInfo info;
    if(NNFMgrGetFirstDelimiter(fmgr,&info)) {
        cout << info << endl;
        while(NNFMgrGetNextDelimiter(fmgr,&info)) {
            cout << info << endl;
        }
    }
    continue;
}
}

```



```

if(!strcmp(cmd, "lf")) {
    cout << "***** List fields *****" << endl;
    NNFMgrFieldInfo info;
    if(NNFMgrGetFirstField(fmgr,&info)) {
        cout << info << endl;
        while(NNFMgrGetNextField(fmgr&info)) {
            cout << info << endl;
        }
    }
    continue;
}

if(!strcmp(cmd, "gd")) {
    cout << "***** Get a delimiter named '" << arg1 << '\''
        << endl;
    NNFMgrDelimiterInfo info;
    if(NNFMgrGetDelimiter(fmgr,arg1,&info)) {
        cout << info << endl;
    } else {
        cout << "***** FAILED : "
            << NNFMgrGetErrorMessage(fmgr) << "*****"
            << endl;
    }
    continue;
}

if(!strcmp(cmd, "cd")) {
    cout << "***** Create a delimiter named '" << arg1
        << "' with value '" << arg2 << "' *****" << endl;
    NNFMgrDelimiterInfo info;
    NNF_CLEAR(&info);
    strcpy(info.delimiterName, arg1);
    strcpy((char *) info.delimiterValue, arg2);
    info.delimiterLength = (arg2) ? strlen(arg2) : 0;

    if(NNFMgrCreateDelimiter(fmgr,&info)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
            << NNFMgrGetErrorMessage(fmgr) << "*****"
            << endl;
        RollbackXact(session);
    }
}

```

```

    }
    continue;
}
if(!strcmp(cmd, "cf")) {
    cout << "***** Create a field named '" << arg1
         << "' with description '" << arg2 << "' *****"
         << endl;
    NNFMgrFieldInfo info;
    NNF_CLEAR(&info);
    strcpy( info.fieldName, arg1);
    strcpy( info.fieldDescription, arg2);

    if(NNFMgrCreateField(fmgr,&info)) {
        cout << "***** SUCCESS *****" << endl;
        CommitXact(session);
    } else {
        cout << "***** FAILED : "
             << NNFMgrGetErrorMessage(fmgr) << "*****"
             << endl;
        RollbackXact(session);
    }
    continue;
}
if(!strcmp(cmd,"help") || !strcmp(cmd,"?")) {
    do_help();
    continue;
}
if(!strcmp(cmd, "quit") || !strcmp(cmd,"q")) {
    break;
}
cout << "% invalid command: " << cmd << endl;
}

NNFMgrClose(fmgr);
CloseDbmsSession(session);

return0;
}

```

---

## 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).

<b>Access Mode</b>	<b>Description</b>
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 gets 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), 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 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 when<sup>o</sup> 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 is:

```

{
    (F0)
    {
        (
            /
                (F1 F2)
            (
                /
                    (F3)
                \

```

```

    /
    (F4 F5)
    \
    )
    {
    (F6)
    }
    \
    /
    (F1)
    (
    /
    (F8)
    \
    /
    (F9)
    \
    )
    {
    (F6)
    }
    \
    )
    }
}
```

Your output format definition should be constructed like this. Because 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 is:

```

{
/

```

```

        (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 is:

```
{
    [ (F1 F2) ]
    (F3 F4 F5)
}
```

If you want the 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 is:

```
{  
    (F1 F2 F3)  
}
```

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)  
}
```

To conduct an evaluation on the instance of F1 in that repetition, 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; 10 is 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 have the first entry. We can print it out or store it. A
// good way to store all the entries for this control is to
// allocate an array of NNFMgrSubstituteCntlInfo structures.
// The size of this array will be numRemainingEntries + 1
// (for the entry we already have).
NNFMgrSubstituteCntlInfo* entries = new
        NNFMgrSubstituteCntlInfo[numRemainingEntries + 1];
// Store the first entry.
entries[0] = info;
// This is equivalent to a memcpy from one struct to another.
// Get and store the remaining entries for (int i=1;
// i < numRemainingEntries + 1; i++)
{
        if (!NNFMgrGetNextEntryFromSubstituteCntl(pNNFMgr, &info))
            {
                // error
            }
        entries[i] = info;
    }
}

```

---

## Appendix D

# Data Type Descriptions

---

### Supported Data Types

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
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 if 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 dash (-) and period (.) characters are not valid ASCII numeric characters.

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
Binary	DATA_TYPE_Binary_ Data	<p>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.</p> <p>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.</p>
EBCDIC	DATA_TYPE_ EBCDIC_Data	<p>A string of characters encoded using the EBCDIC (Extended Binary Coded Decimal Interchange 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 20).</p>

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
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).

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
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, 123" 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.

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
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.

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
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_ LittleEndian2	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.



<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
Date and Time		<p>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.</p> <p>Combined dates and times may be represented in any of the following list of data types. The list includes String, Numeric, and EBCDIC.</p>
Time		<p>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.</p>
Date		<p>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.</p>

<b>Data Type Field Values</b>	<b>Data Type #Define</b>	<b>Description</b>
Custom Date and Time		<p>Custom Date and Time enables users to specify different formats of dates, times, and combined dates and times.</p> <p>Date/Time formats may include:</p> <ol style="list-style-type: none"> <li>1) Variations in year (2 or 4 digit year representation: YY or YYYY).</li> <li>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”.</li> <li>3) Variations in the day of the month – use of a day of the month number (01-31). “DD” is the format string.</li> <li>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.</li> <li>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 <i>Specifying a Year Cutoff Value</i> in Volume I.</li> </ol>

---

## 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 do not 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 generates 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 a 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, to convert from IBM signed packed integer to IBM packed integer, if the 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 hexadecimal 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
```

To convert between an ASCII string and binary, Formatter expects the ASCII string to be a proper representation of a binary value. If you have the input:

```
“Hello, world!”
```

and you want Formatter to generate a binary value on output.

Formatter issues an error because the ASCII string is not a proper string representation of a binary value. The string must be composed of the characters 0-9 and A-F.

The actual binary encoding of the ASCII string “Hello, world!” is:

```
48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21
```

Specify that the data type of the input is binary, not ASCII string. Formatter generates the internal ASCII string:

```
“48656c6c6f2c20776f726c6421”
```

and then converts this back to binary:

```
48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21
```

To construct a binary value that equates to the actual byte values of your input, specify that the input data type is also Binary. If you have an input data type other than Binary, Formatter attempts to interpret your input as the string representation of a binary value.

## Appendix E

# ASCII Extended Character Set

Decimal Value	Hex Value	Extended Character Set
000	00	NUL
001	01	SCH
002	02	STX
003	03	ETX
004	04	EOT
005	05	ENO
006	06	ACK
007	07	BEL
008	08	BS
009	09	HT
010	0A	LF
011	0B	VT
012	0C	FF
013	0D	CR
014	0E	SO
015	0F	SI

Decimal Value	Hex Value	Extended Character Set
016	10	DLE
017	11	DC1
018	12	DC2
019	13	DC3
020	14	DC4
021	15	NAK
022	16	SYN
023	17	ETB
024	18	CAN
025	19	EM
026	1A	SUB
027	1B	ESCAPE
028	1C	FS
029	1D	GS
030	1E	RS
031	1F	US

<b>Decimal Value</b>	<b>Hex Value</b>	<b>Extended Character Set</b>
032	20	SPACE
033	21	!
034	22	“
035	23	#
036	24	\$
037	25	%
038	26	&
039	27	‘
040	28	(
041	29	)
042	2A	*
043	2B	+
044	2C	,
045	2D	-
046	2E	.
047	2F	/
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6

<b>Decimal Value</b>	<b>Hex Value</b>	<b>Extended Character Set</b>
055	37	7
056	38	8
057	39	9
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M

<b>Decimal Value</b>	<b>Hex Value</b>	<b>Extended Character Set</b>
078	4E	N
079	4F	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	^
095	5F	_
096	60	`
097	61	a
098	62	b
099	63	c
100	64	d

<b>Decimal Value</b>	<b>Hex Value</b>	<b>Extended Character Set</b>
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{

Decimal Value	Hex Value	Extended Character Set
124	7C	
125	7D	}
126	7E	~
127	7F	DEL
128	80	€
129	81	unused
130	82	,
131	83	f
132	84	„
133	85	...
134	86	†
135	87	‡
136	88	^
137	89	‰
138	8A	Š
139	8B	<
140	8C	Œ
141	8D	unused
142	8E	unused
143	8F	unused
144	90	unused
145	91	‘
146	92	’

Decimal Value	Hex Value	Extended Character Set
147	93	“
148	94	”
149	95	•
150	96	–
151	97	—
152	98	~
153	99	™
154	9A	š
155	9B	>
156	9C	œ
157	9D	unused
158	9E	unused
159	9F	Ÿ
160	A0	nonbreaking space
161	A1	ı
162	A2	ç
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨



Decimal Value	Hex Value	Extended Character Set
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	-
174	AE	®
175	AF	ˆ
176	B0	°
177	B1	±
178	B2	²
179	B3	³
180	B4	´
181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿

Decimal Value	Hex Value	Extended Character Set
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö

Decimal Value	Hex Value	Extended Character Set
215	D7	×
216	D8	∅
217	D9	Û
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Ɔ
223	DF	β
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í

Decimal Value	Hex Value	Extended Character Set
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

---

## Appendix F

# EBCDIC Character Set

---

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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	ç		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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
082	52			0101 0010
083	53			0101 0011
084	54			0101 0100
085	55			0101 0101
086	56			0101 0110
087	57			0101 0111
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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
120	78			0111 1000
121	79			0111 1001
122	7A	:		0111 1010
123	7B	#		0111 1011
124	7C	@		0111 1100
125	7D	'		0111 1101



<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
148	94	m		1001 0100
149	95	n		1001 0101
150	96	o		1001 0110
151	97	p		1001 0111
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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

<b>Decimal Value</b>	<b>Hex Value</b>	<b>EBCDIC Value*</b>	<b>Description</b>	<b>Binary</b>
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
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 G

# 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

---

## Symbols

= operator 151

## A

access modes 25  
AddInputMessage 83  
AddOutputMessage 87  
AddPair 194  
Alternative compound input format 19  
apitest.cpp 468  
ASCII Extended Character Set 519  
automatically reformatting messages 58

## C

Callback (dbSession) 180  
Callback (dbSession, nameValuePairArray) 181  
Callback (dbSession, nameValuePairArray,  
userRuntimeData) 182  
Callback (dbSession, parsedFields) 187  
Callback (dbSession, parsedFields,  
nameValuePairArray) 188  
Callback (dbSession, parsedFields,  
nameValuePairArray,  
userRuntimeData) 189  
Callback (dbSession, userRuntimeData) 183  
Callback (nameValuePairArray) 174  
Callback (No Parameters) 173  
Callback (userRuntimeData) 175, 176  
Case controls  
    NNFMgrGetCaseCntl 339  
class <UserDerivedCallback Class>; public  
    <NN...UserFunction> 169  
class NNDBFieldsUserFunction: public  
    NNUserFunction 169  
class NNDBUserFunction: public NNUserFunction  
    169  
class NNGenericUserFunction: public

    NNUserFunction 168  
class NNUserFunction 168  
Collection controls  
    NNFMgrAddCntlToCollection 354  
    NNFMgrCreateCollectionCntl 350  
    NNFMgrGetCntlFromCollection 356  
    NNFMgrGetCollectionCntl 352  
components of a format 7  
compound input formats 19  
    Alternative 19  
    inserting compound formats 20  
    Ordinal 19  
    Tagged Ordinal 19  
compound output formats  
    Alternative 35  
    inserting compound output formats 36  
    inserting flat output formats 36  
    Ordinal 35  
Constructor (Byte Array Return Result Type) 149  
Constructor (Double Return Result Type) 148  
Constructor (General Case) 150  
Constructor (Long Return Result Type) 147  
Constructors 147  
converting data types 36, 37, 43  
converting formats automatically 58  
custom Date/Time formats  
    NNFMgrGetDateTimeFormatString 362

## D

data cleanup 205  
Data section 13  
data types  
    converting 37, 43  
    value ranges 37  
Default Control Name API  
    GetDefaultCntlName 316  
default controls  
    NNFMgrCreateDefaultCntl 313

NNFMgrGetDefaultCntl 315  
documentation set 1

## E

equals operator 151  
error handling  
    GetErrorCode 206  
    GetErrorMessage 207  
error messages 208  
error status 146  
Exit Cleanup functions 131  
Exit Function Developer Interface 131

## F

Field Management API Structures  
    NNFMgrFieldInfo 234  
Field Management APIs 234  
    NNFMgrCreateField 236  
    NNFMgrDeleteField 241  
    NNFMgrGetFirstField 237  
    NNFMgrGetNextField 238  
    NNFMgrUpdateOutputControl 239  
fkColl 196  
flat input formats 8  
flat output formats 21  
flow of calls 60  
fmgr.cpp 475  
Format Management  
    GetErrorMessage 453  
    NNFMgrAppendFieldToInputFormat 427  
    NNFMgrFormatInfo 418  
    Sample Programs  
        fmgr.cpp 475  
Format Management API Error Handling  
    Format Management Error Messages 454  
    GetErrorMessage 453  
    GetErrorNo 452  
Format Management API structures  
    NNFMgrFlatFormatInfo 421  
    NNFMgrFormatInfo 418  
    NNFMgrInFieldInfo 422  
    NNFMgrOutFieldInfo 423  
    NNFMgrRepeatFormatInfo 419  
Format Management APIs  
    General Format Management APIs 231

Literal Management API structures 371  
NNFMgrAppendFieldToInputFormat 427  
NNFMgrAppendFieldToOutputFormat 429  
NNFMgrAppendFormatToFormat 431  
NNFMgrCreateFormat 425  
NNFMgrDeleteFormat 450  
NNFMgrGetFirstChildFormat 445  
NNFMgrGetFirstFieldFromInputFormat  
    439, 441  
NNFMgrGetFirstFieldFromOutputFormat  
    442  
NNFMgrGetFirstFormat 435  
NNFMgrGetFormat 433  
NNFMgrGetNextChildFormat 447  
NNFMgrGetNextFieldFromInputFormat 444  
NNFMgrGetNextFormat 437  
NNFMgrUpdateFormat 448  
Output Format Control Management APIs  
    242  
    output operations 274

Format Management Error Messages 454  
Format Management APIs 417  
formats

    components 7  
    converting automatically 58  
    input formats 8  
    output 20  
    structure 7

Formatter

    data cleanup 205  
    flow of calls 60  
    libraries 76  
    Sample Programs  
        apitest.cpp 468  
        fmgr.cpp 475  
        msgtest.cpp 457  
    thread safety 202  
    thread safety impacts 203

Formatter API functions

    Formatter error handling 206  
    Formatter error messages 208  
    linking with thread safe Formatter 205

Formatter APIs 77

Formatter Constructor 77, 198

Formatter Destructor 79

Formatter Engine

    using 59

- Formatter error handling 206
  - GetErrorCode 206
  - GetErrorMessage 207
- Formatter error messages 208
  - parsing errors 226
- Formatter Management APIs 229
- Formatter member functions
  - AddInputMessage 83
  - AddOutputMessage 87
  - Formatter Constructor 77
  - Formatter Destructor 79
  - GetFieldAscii 101
  - GetFieldAsciiByTag 103
  - GetOutMsgCount 105
  - GetOutMsgGroup 106
  - GetParsedInMsg 109
  - GetParsedInMsgCount 108
  - parse 97
  - PreloadInFormat 90
  - PreloadOutFormat 92
  - reformat 99
  - ResetDbmsSession 81
  - SetUserTypeValidationOff 111
  - SetUserTypeValidationOn 110
  - StartDebug 94
  - StopDebug 96
  - UserTypeValidationIsOn 112
- Formatter Validation On/Off functions 199
- formatting output 22, 29

## G

- General Format Management APIs
  - NNF\_CLEAR 233
  - NNFMgrClose 232
  - NNFMgrInit 231

- GetAsciiValue 120
- GetByteOffset 124
- GetCompCount 125
- GetCurrInFldAsciiData 142
- GetCurrInFldData 141
- GetCurrInFldLength 143
- GetCurrInFldName 139
- GetCurrInFldType 144
- GetCurrOutFldName 140
- GetDefaultCntlName 316
- GetErrorCode 206
- GetErrorMessage 207, 453
- GetErrorNo 452
- GetFieldAscii 101, 138
- GetFieldAsciiByTag 103
- GetFieldComp 128
- GetFieldString 138
- GetFmtVal 130
- GetFmtValLen 129
- GetFmtValue 122
- GetFmtValueLen 123
- GetInfo 119, 127
- GetMsg 115
- GetMsgBuffer 113
- GetMsgComp 126
- GetMsgCount 116
- GetMsgLength 114
- GetOutMsgCount 105
- GetOutMsgGroup 106
- GetParsedInMsg 109
- GetParsedInMsgCount 108
- GetParsedOutMsg 118
- GetParsedOutMsgCount 117
- GetStringValue 120
- GetUserExitRoutineName 145
- GetValue 121

## I

- input controls
  - Data Only 9
  - Data section 13
  - Length and Data 9
  - Length section 18
  - Length, Tag and Data 9
  - Literal 9
  - mandatory property 12

- optional property 12
- parsing fields 12
- Regular Expression 9
- Repetition Count 9
- Tag and Data 9
- Tag section 16
- Tag,Length and Data 9
- input fields
  - validation 197
- input formats 8
  - compound 19
  - flat 8
- inserting compound formats 20
- inserting compound output formats 36
- inserting flat output formats 36

## J

- Justify controls
  - NNFMgrGetJustifyCntl 341

## L

- Length and Data control type 9
- Length controls
  - NNFMgrCreateLengthCntl 323
  - NNFMgrGetLengthCntl 325
- Length section 18
- Length,Tag and Data control type 9
- libraries 76
- linking with thread safe Formatter 205
- Literal control type 9
- Literal Management API structures
  - NNFMgrLiteralInfo 372
- Literal Management APIs
  - NNFMgrDeleteLiteral 378
- literals
  - NNFMgrCreateLiteral 373
  - NNFMgrGetLiteral 374
- Lookup 195

## M

- MakeNull 166
- mandatory property 12
- math expression controls 294
  - NNFMgrAppendSegMathExpCntl 298

- NNFMgrCreateMathExpCntl 295
- NNFMgrGetMathExpCntl 296
- NNFMgrGetSegFromMathExpCntl 299
- Math Expression syntax 32
- messages
  - reformatting automatically 58
- msgtest.cpp 457

## N

- NameValue Pair 159
- NameValuePair (Alternate Constructor) 162
- NameValuePair (Assignment Constructor) 164
- NameValuePair (Copy Constructor) 163
- NameValuePair (Default Constructor) 161
- NameValuePair (Destructor) 165
- NameValuePair member functions 161
- NN\_ERSTATUS\_ERROR 146
- NN\_ERSTATUS\_OK 146
- NNDBFieldsUserFunction 185
- NNDBFieldsUserFunction member functions 187
  - Callback (dbSession, parsedFields) 187
  - Callback (dbSession, parsedFields, nameValuePairArray) 188
  - Callback (dbSession, parsedFields, nameValuePairArray, userRuntimeData) 189
  - Callback (dbSession, parsedFields, userRuntimeData)Callback (dbSession, parsedFields, userRuntimeData) 190
  - RuntimeData Lookup 191
- NNDBUserFunction 178
- NNDBUserFunction member functions
  - Callback (dbSession) 180
  - Callback (dbSession, nameValuePairArray) 181
  - Callback (dbSession, nameValuePairArray, userRuntimeData) 182
  - Callback (dbSession, userRuntimeData) 183
- NNExitRet Class
  - User Exit Return Object 146



**NNF\_CLEAR 233**  
**NNFMgrAddCntlToCollection 354**  
**NNFMgrAddNameValuePair 387**  
**NNFMgrAppendSegMathExpCntl 298**  
**NNFMgrAppendEntryToSubstitutControl 277**  
**NNFMgrAppendFieldToInputFormat 427**  
**NNFMgrAppendFieldToOutputFormat 429**  
**NNFMgrAppendFormatToFormat 431**  
**NNFMgrCaseCntlInfo 259**  
**NNFMgrCollectionCntlInfo 263**  
**NNFMgrCreateCollectionCntl 350**  
**NNFMgrCreateDefaultCntl 313**  
**NNFMgrCreateField 236**  
**NNFMgrCreateFormat 425**  
**NNFMgrCreateLengthCntl 323**  
**NNFMgrCreateLiteral 373**  
**NNFMgrCreateMathExpCntl 295**  
**NNFMgrCreateOutMstrCntl 267**  
**NNFMgrCreateParseControl 404**  
**NNFMgrCreatePrePostFixCntl 306**  
**NNFMgrCreateSubstituteCntl 275**  
**NNFMgrCreateSubStringCntl 331**  
**NNFMgrCreateTrimCntl 342**  
**NNFMgrCreateUserDefinedType 385**  
**NNFMgrCreateUserExitCntl 287**  
**NNFMgrDefaultCntlInfo 255**  
**NNFMgrDeleteCollectionCntl 360**  
**NNFMgrDeleteDefaultCntl 321, 337**  
**NNFMgrDeleteField 241**  
**NNFMgrDeleteFormat 450**  
**NNFMgrDeleteLengthCntl 329**  
**NNFMgrDeleteLiteral 378**  
**NNFMgrDeleteMathExpCntl 303**  
**NNFMgrDeleteOutputControl 370, 380**  
**NNFMgrDeleteParseControl 415**  
**NNFMgrDeletePrePostFixCntl 311**  
**NNFMgrDeleteSubstituteControl 272, 285**  
**NNFMgrDeleteTrimCntl 348**  
**NNFMgrDeleteUserDefinedType 396**  
**NNFMgrDeleteUserExitCntl 292**  
**NNFMgrFieldInfo 234**  
**NNFMgrFlatFormatInfo 421**  
**NNFMgrFormatInfo 418**  
**NNFMgrGetCaseCntl 339**  
**NNFMgrGetCntFromCollection 356**  
**NNFMgrGetCollectionCntl 352**  
**NNFMgrGetDateTimeFormatString 362**  
**NNFMgrGetDefaultCntl 315**  
**NNFMgrGetFirstField 237**  
**NNFMgrGetFirstChildFormat 445**  
**NNFMgrGetFirstFieldFromInputFormat 439**  
**NNFMgrGetFirstFieldFromOutputFormat 442**  
**NNFMgrGetFirstFormat 435**  
**NNFMgrGetFirstParseControl 409**  
**NNFMgrGetFirstUserDefinedType 390**  
**NNFMgrGetFormat 433**  
**NNFMgrGetJustifyCntl 341**  
**NNFMgrGetLengthCntl 325**  
**NNFMgrGetLiteral 374**  
**NNFMgrGetMathExpCntl 296**  
**NNFMgrGetNextChildFormat 447**  
**NNFMgrGetNextEntryFromSubstituteCntl 281**  
**NNFMgrGetNextField 238**  
**NNFMgrGetNextFieldFromInputFormat 441**  
**NNFMgrGetNextFieldFromOutputFormat 444**  
**NNFMgrGetNextFormat 437**  
**NNFMgrGetNextParseControl 411**  
**NNFMgrGetNextUserDefinedType 392**  
**NNFMgrGetOutMstrCntl 269**  
**NNFMgrGetParseControl 407**  
**NNFMgrGetPrePostFixCntl 307**  
**NNFMgrGetSegFromMathExpCntl 299**  
**NNFMgrGetSubstituteCntl 279**  
**NNFMgrGetSubStringCntl 333**  
**NNFMgrGetTrimCntl 344**  
**NNFMgrGetUserDefinedType 388**  
**NNFMgrGetUserExitCntl 288**  
**NNFMgrInFieldInfo 422**  
**NNFMgrInit 231**  
**NNFMgrIsRecursiveCollection 366**  
**NNFMgrIsRecursiveFormat 364**  
**NNFMgrJustifyCntlInfo 260**  
**NNFMgrLengthCntlInfo 256**  
**NNFMgrLiteralInfo 372**  
**NNFMgrMathExpCntlInfo 251**  
**NNFMgrMathExpCntlSegmentInfo 252**  
**NNFMgrNameValuePairInfo 384**  
**NNFMgrOutFieldInfo 423**  
**NNFMgrOutMstrCntlInfo 243**  
**NNFMgrParseControlInfo 399**  
**NNFMgrPrePostFixCntlInfo 253**  
**NNFMgrRepeatFormatInfo 419**  
**NNFMgrSubstituteCntlInfo 248**  
**NNFMgrUpdateCollectionCntl 358**

- NNFMgrUpdateDefaultCntl 319
- NNFMgrUpdateField 239
- NNFMgrUpdateFormat 448
- NNFMgrUpdateLengthCntl 327
- NNFMgrUpdateMathExpCntl 301
- NNFMgrUpdateOutMstrCntl 270
- NNFMgrUpdateOutputControl 368
- NNFMgrUpdatePrePostFixCntl 309
- NNFMgrUpdateSubstituteCntl 283
- NNFMgrUpdateSubStringCntl 335
- NNFMgrUpdateTrimCntl 346
- NNFMgrUpdateUserDefinedType 394
- NNFMgrUpdateuserExitCntl 290
- NNFMgrUserDefTypeInfo 383
- NNFMgrUserExitCntlInfo 250
- NNFMgSubStringCntlInfo 257
- NNFunctionKeyPairCollection 193
- NNFunctionKeyPairCollection member functions
  - AddPair 194
  - Lookup 195
- NNFunctionKeyPairCollection Private Data member
  - fkColl 196
- NNGenericUserFunction 171
- NNGenericUserFunction member functions
  - Callback (nameValuePairArray) 174
  - Callback (userRuntimeData) 175, 176
  - RuntimeDataLookup 177
- NNGetUserExitFuncPtrs 133
- NNParsedFields Class member functions
  - GetCurrInFldAsciiData 142
  - GetCurrInFldData 141
  - GetCurrInFldLength 143
  - GetCurrInFldName 139
  - GetCurrInFldType 144
  - GetCurrOutFldName 140
  - GetFieldAscii 138
  - GetFieldString 138
  - GetUserExitRoutineName 145
- NNRMgrClose 232
- NNUserFunction 170

**O**

- operator overloads
  - = operator 151
  - equals operator 151
- optional property 12
- Ordinal compound input format 19
- ouput controls
  - properties
    - output operation 27
- OutMsg Class member functions
  - GetMsgBuffer 113
  - GetMsgLength 114
- OutMsgGroup Class member functions
  - GetMsg 115
  - GetMsgCount 116
  - GetParsedOutMsg 118
  - GetParsedOutMsgCount 117
- Output Control Management APIs
  - NNFMgrDeleteCollectionCntl 360
  - NNFMgrDeleteDefaultCntl 321, 337
  - NNFMgrDeleteExitCntl 292
  - NNFMgrDeleteLengthCntl 329
  - NNFMgrDeleteMathExpCntl 303
  - NNFMgrDeleteOutputControl 370, 380
  - NNFMgrDeletePrePostFixCntl 311
  - NNFMgrDeleteSubstituteControl 272, 285
  - NNFMgrDeleteTrimCntl 348
  - NNFMgrUpdateCollectionCntl 358
  - NNFMgrUpdateDefaultCntl 319
  - NNFMgrUpdateLengthCntl 327
  - NNFMgrUpdateMathExpCntl 301
  - NNFMgrUpdateOutMstrCntl 270
  - NNFMgrUpdateOutputControl 368
  - NNFMgrUpdatePrePostFixCntl 309
  - NNFMgrUpdateSubstituteCntl 283
  - NNFMgrUpdateSubstringCntl 335
  - NNFMgrUpdateTrimCntl 346
  - NNFMgrUpdateUserExitCntl 290
- output controls 22
- output fields 25
- Output Format Control Management API structures
  - NNFMgDefaultCntlInfo 255
  - NNFMgLengthCntlInfo 256
  - NNFMgMathExpCntlInfo 251
  - NNFMgMathExpCntlSegmentInfo 252
  - NNFMgprePostFixCntlInfo 253
  - NNFMgrCaseCntlInfo 259
  - NNFMgrCollectionCntlInfo 263
  - NNFMgrJustifyCntlInfo 260
  - NNFMgrOutMstrCntlInfo 243

- NNFMgrSubstituteCntlInfo 248
- NNFMgrUserExitCntlInfo 250
- NNFMgrSubStringCntlInfo 257
- Output Format Control Management APIs
  - NNFMgrCreateOutMstrCntl 267
  - NNFMgrGetOutMstrCntl 269
- output formats 20
  - compound 35
  - flat 21
- output formatting 22
- output operation collections 33
- output operation property 27
- output operations 29
  - Case controls 339
  - Collection controls 350
  - collections 33
  - custom Date/Time formats 362
  - default controls 313
  - Justify controls 341
  - Length controls 323
  - math expression controls 294
  - Math Expression syntax 32
  - Pre/PostFix controls 305
  - recursion checking 363
  - Substitute controls 274
  - SubString controls 331
  - Trim controls 342
- Overview 5

## P

- parse 97
- Parse Control Management API structures
  - NNFMgrParseControlInfo 399
- Parse Control Management APIs 398
  - NNFMgrCreateParseControl 404
  - NNFMgrDeleteParseControl 415
  - NNFMgrGetFirstParseControl 409
  - NNFMgrGetNextControl 411
  - NNFMgrGetParseControl 407
- ParsedField Class member functions
  - GetAsciiValue 120
  - GetByteOffset 124
  - GetFmtValue 122
  - GetFmtValueLen 123
  - GetInfo 119
  - GetStringValue 120

- GetValue 121
- ParsedMessage Class member functions
  - GetCompCount 125
  - GetFieldComp 128
  - GetFmtVal 130
  - GetFmtValLen 129
  - GetInfo 127
  - GetMsgComp 126
- parsing errors 226
- parsing fields 12
- Pre/PostFix controls
  - NNFMgrCreatePrePostFixCntl 306
  - NNFMgrGetPrePostFixCntl 307
- PreloadInFormat 90
- PreloadOutFormat 92
- public methods 151

## R

- RE syntax 10
- rebuilding msgtest for User Exits 156
- rebuilding ruleng for User Exits 156
- recursion checking
  - NNFMgrIsRecursiveCollection 366
  - NNFMgrIsRecursiveFormat 364
- reformat 99
- registering User Exit functions 131
- Regular Expression control type 9
- Regular Expression syntax 10
- Repetition Count control type 9
- ResetDbmsSession 81
- RuntimeDataLookup 177, 184, 191

## S

- Sample Programs
  - apitest.cpp 468
  - fmgr.cpp 475
  - msgtest.cpp 457

- Set 167
- SetByteArrayValue 152
- SetError 153
- SetUserTypeValidationOff 111
- SetUserTypeValidationOn 110
- StartDebug 94
- StopDebug 96
- Substitute controls 274
  - NNFMgrAppendEntryToSubstituteControl 277
  - NNFMgrCreateSubstituteCntl 275
  - NNFMgrGetNextEntryFromSubstituteCntl 281
  - NNFMgrGetSubstituteCntl 279
- SubString controls
  - NNFMgrCreateSubStringCntl 331
  - NNFMgrGetSubStringCntl 333

## T

- Tag and Data control type 9
- Tag section 16
- Tag.Length and Data control type 9
- Tagged Ordinal compound input format 19
- thread safety 202
- Trim controls
  - NNFMgrCreateTrimCntl 342
  - NNFMgrGetTrimCntl 344

## U

- User Callback API functions 157, 199
  - AddPair 194
  - Callback (dbSession) 180
  - Callback (dbSession, nameValuePairArray) 181
  - Callback (dbSession, nameValuePairArray, userRuntimeData) 182
  - Callback (dbSession, parsedFields) 187
  - Callback (dbSession, parsedFields, nameValuePairArray) 188
  - Callback (dbSession, parsedFields, nameValuePairArray, userRuntimeData) 189
  - Callback (dbSession, parsedFields, userRuntimeData) 190
  - Callback (dbSession, userRuntimeData) 183

- Callback (nameValuePairArray) 174
- Callback (No Parameters) 173
- Callback (userRuntimeData) 175, 176
- class <UserDerivedCallback Class>: public <NN...UserFunction> 169
- class NNDBFieldsUserFunction: public NNUserFunction 169
- class NNDBUserFunction: public NNUserFunction 169
- class NNGenericUserFunction: public NNUserFunction 168
- class NNUserFunction 168
- fkColl 196
- Lookup 195
- MakeNull 166
- NameValuePair (Alternate Constructor) 162
- NameValuePair (Assignment Constructor) 164
- NameValuePair (Copy Constructor) 163
- NameValuePair (Default Constructor) 161
- NameValuePair (Destructor) 165
- NameValuePair member functions 161
- NNDBFieldsUserFunction 185
- NNDBFieldsUserFunction member functions 187
- NNDBUserFunction 178
- NNDBUserFunction member functions 180
- NNFunctionKeyPairCollection 193
- NNFunctionKeyPairCollection member functions 194
- NNFunctionKeyPairCollection Private Data member 196
- NNGenericUserFunction 171
- NNGenericUserFunction member functions 173
- NNUserFunction 170
- RuntimeDataLookup 177, 184, 191
- Set 167
- User Callback Class definition 168
- User CallbackLookup Interface 192
- User-defined Type Input Field Validation 197
- User Callback API structures
  - NameValue Pair 159

- User Callback Class definition 168
- User CallbackLookup Interface 192
- user callbacks 157
- User Exit API functions 156
  - Constructor (Byte Array Return Result Type) 149
  - Constructor (Double Return Result Type) 148
  - Constructor (General Case) 150
  - Constructor (Long Return Result Type) 147
  - Constructors 147
  - NNGetUserExitFuncPtrs 133
  - NNParsedFields Class member functions 138, 139, 140, 141, 142, 143, 144, 145
  - operator overloads 151
  - public methods 151
  - rebuilding msgtest 156
  - rebuilding ruleng 156
  - SetByteArrayValue 152
  - SetError 153
  - summary 156
  - User Exit Cleanup function specification 155
  - User Exit Function Developer Interface 135
  - User Exit Function Specification 135
  - User Exit Lookup Interface 131
  - User Exit Return Object 146
- User Exit APIs 131
- User Exit Cleanup function specification 155
- User Exit controls
  - NNFMgrCreateUserExitCntl 287
  - NNFMgrGetUserExitCntl 288
- User Exit Function Developer Interface 135
- User Exit Function Specification 135
- User Exit Lookup Interface 131
- User-Defined Data Type Management API
  - structures
    - NNFMgrNameValuePairInfo 384
    - NNFMgrUserDefTypeInfo 383
- User-Defined Data Type Management APIs 382
  - NNFMgrAddNameValuePairs 387
  - NNFMgrCreateUserDefinedType 385
  - NNFMgrDeleteUserDefinedType 396
  - NNFMgrGetFirstUserDefinedType 390
  - NNFMgrGetNextUserDefinedType 392
  - NNFMgrGetUserDefinedType 388
  - NNFMgrUpdateUserDefinedType 394
- User-defined Type Input Field Validation 197, 198, 199
  - example 200
  - User Callback API functions 199
- UserTypeValidationIsOn 112
  - using the Formatter Engine 59

**V**

- validating input fields 197
- value ranges 37

**Y**

- Year 2000 Compliance 403, 406

**Sending your comments to IBM  
MQSeries Integrator  
Programming Reference for NEONFormatter  
SC34-5507-01**

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book only and the way in which the information is presented.

To request additional publications or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

By mail, use the Readers' Comment Form

By fax:

From outside the U.K., use your international access code followed by 44 1962 870229

From within the U.K., use 01962 870229

Electronically, use the appropriate network ID:

IBM Mail Exchange: GBIBM2Q9 at IBMMAIL

IBMLink: HURSLEY(IDRCF)

Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

The publication number and title

The page number or topic number to which your comment applies

Your name/address/telephone number/fax number/network ID

**Readers' Comments**  
**MQSeries Integrator**  
**Programming Reference for NEONFormatter**  
**SC34-5507-01**

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

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

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

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

Name	Address
Company or organization	
Telephone	Email



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

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

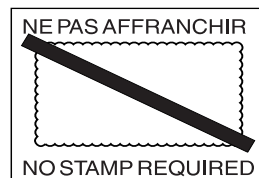
If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

Cut along this line

**2** Fold along this line

**By air mail**  
*Par avion*

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



**REPONSE PAYEE  
GRANDE-BRETAGNE**

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

**3** Fold along this line

*From:* Name \_\_\_\_\_  
 Company or Organization \_\_\_\_\_  
 Address \_\_\_\_\_  
 \_\_\_\_\_  
 EMAIL \_\_\_\_\_  
 Telephone \_\_\_\_\_

Cut along this line

**4** Fasten here with adhesive tape







Printed in U.S.A

SC34-5507-01