IBM

Rules and Formatter Extension for IBM WebSphere Message Broker for Multiplatforms

New Era of Networks Formatter Programming Reference

Version 7.0

Note: Before using this information, and the product it supports, be sure to read the general information under *notices* on page 529.

First edition (April 2010)
This edition applies to Rules and Formatter Extension for IBM WebSphere Message Broker for Multiplatforms, Version7.0, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

IBMContents

Chapter 1
Introduction

Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms is a cross-platform, guaranteed delivery, messaging middleware product designed to facilitate the synchronization, management, and distribution of information (messages) across large-scale, heterogeneous networks.

Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms is configurable and uses a content-based rules message evaluation, formatting, and routing paradigm. Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms also provides a powerful data content-based, source-target mechanism with dynamic format parsing and conversion capability.

About this Document

The ***Programming Reference*** provides descriptions and examples for each function in the New Era of Networks Formatter and New Era of Networks Formatter Management APIs. This guide is divided into the following chapters:

Chapter 1, *Introduction*, lists the documentation set and conventions.

Chapter 2, *Overview,* describes the New Era of Networks Formatter product, its architecture, and its components.

Chapter 3, *New Era of Networks Formatter APIs*, contains New Era of Networks Formatter member functions, user exit API functions, the user exit interface, user callback API functions, and error handling.

Chapter 4, New Era of Networks Formatter *Management APIs*, contains New Era of Networks Formatter general APIs, permission APIs, and management APIs.

Chapter 5, *Error Messages*, lists New Era of Networks Formatter error messages.

Appendix A, *Character Sets*, contains an ASCII extended character set and EBCDIC character set table.

Appendix B, *Data Types*, describes the available data types.

Appendix C, *Access Modes*, describes how to use access modes and provides several examples.

Appendix D, *Notices*, contains trademark and service mark information.

Documentation Set

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

- ***System Management Guide***
- ***New Era of Networks Formatter Programming Reference***
- ***New Era of Networks Rules Programming Reference***
- ***Application Development Guide***
- Rules, Formatter, and Visual Tester online help
- Installation Readme

Document Conventions

The following document conventions are used in this guide.

Text	Convention	Example
code	courier	<user ID> <password>

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

Chapter 2

Overview

New Era of Networks Formatter has two main functions: parsing and reformatting.

- Parse separates an input message into individual fields.
- Reformat converts an input message into an output message with a different format.

New Era of Networks Formatter is packaged as a library of C++ objects that have public functions that constitute the Application Programming Interface (API). An API refers to a set of object classes, structures, and functions used by an application program to invoke New Era of Networks Rules and Formatter services. Application developers develop applications that invoke public New Era of Networks Formatter functions to parse and reformat messages.

Using New Era of Networks Formatter

Formatting an input message to an output message takes three general steps:

1. Parse the input message.
2. Build a structure of the output message that looked like a tree of compound format instances and flat format instances, with field instances as the leaf nodes. Map the field instances from the input message into field instances in the output message.

3. Format each field instance in the output message according to the output format control information for the field, and append the New Era of Networks Formatter field to an output message buffer.

Users build and modify format definitions using one of two methods: the New Era of Networks Formatter graphical user interface (GUI) or the New Era of Networks Formatter Management API functions. The New Era of Networks Formatter GUI is described in the ***User's Guide***.

New Era of Networks Formatter Management API functions are a set of C functions that create format definition data in a relational database. Users can write applications that call the Management API functions to build format definitions.

New Era of Networks Formatter uses format definitions that describe how to parse fields in an input message and how to format fields in an output message. Format definition data resides in a relational database. The user selects output field access modes and associated input field names to tell New Era of Networks Formatter how to map fields in an output message. For more information on field mapping, see the New Era of Networks Formatter chapter of the ***User's Guide***.

New Era of Networks Formatter mapping uses explicit input field to output field mapping information stored in the New Era of Networks Formatter database. Mapping provides the capability to map any number of input format definitions to any number of output format definitions, assuming that fields are reused across different formats. *New Era of Networks Formatter Mapping Management APIs* on page 406

Permissions allow a layer of security to be added to New Era of Networks Formatter objects. This security defines the user's ability to read, update, and own objects. *New Era of Networks Formatter Permission APIs* on page 153

Two test programs, apitest and msgtest, are 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. For detailed information on msgtest and apitest, see the ***System Management Guide***.

The Consistency Checker verifies the integrity of the format definition data in the relational database. Run the Consistency Checker periodically to insure data integrity when you are building format definition data is being built. For

more information on the Consistency Checker, see the ***System Management Guide***.

NNFie is a New Era of Networks Formatter tool used to export format definitions from a database to an export file, and to import from the export file into a database. You can run NNFie from the command line or from the GUI. For information on running NNFie, see the ***System Management Guide***.

Formatter Naming Conventions

Several rules apply when you are creating names for format components:

- Do not use case differences to distinguish component names. Case-sensitive databases distinguish between component names that use uppercase and lowercase alphabetic characters. For example, a component named Item1 is distinct from a component named ITEM1. However, some databases do not distinguish case and would interpret both components as having the same name. Each matching component would cause a conflict during import to a case-sensitive database.
- Component names must be unique and should be descriptive.
- Do not name a component NONE. NONE is a reserved word.
- Names can be a maximum of 120 characters. If you exceed 120 characters, a message box appears and you can either edit the component name or cancel the operation.
- Single quotes, double quotes, and spaces should not be used in names.

Using the New Era of Networks Formatter Engine

The New Era of Networks Formatter engine processes the input messages, the input formats describing how the input messages are parsed, and the output formats describing how the input messages are formatted. At the end of the process, the New Era of Networks Formatter engine produces parsed input messages and reformatted output messages as the finished product.

Input messages are sent to New Era of Networks Formatter one at a time, using the AddInputMessage() function. In addition to the message specified in the function call, the user must also specify the input format used to parse the input message. The input message and format arguments are specified using char* variables to provide the address of the buffer containing each variable to the application. The name of the input format provided is used to retrieve the specified input format from the database.

Output formats describing how to format the parsed input messages are provided to New Era of Networks Formatter using the AddOutputFormat() function. An output format is specified using a char* variable to provide 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 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. Delete the Formatter instance. Close the database
    session.
```

Using the Thread Safe Release of New Era of Networks Formatter

The thread safe New Era of Networks Formatter follows the same scheme, except each thread has its own set of input messages and resulting output messages. New Era of Networks 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. Each thread can process its own data concurrently with the other threads. 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 created by a reformat on another thread created.

The template defined earlier for formatting changes slightly, as shown in the following 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 a number of threads, passing each thread the same
Formatter instance.

For each thread:

     While there are input messages to format
```

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

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

        Call reformat() on the Formatter instance.
        [Formatter formats one output message for each output
         format that was specified by AddOutputFormat.]

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

        end While

Clean up and terminate the thread.

end For
```

How Thread Safety Impacts New Era of Networks 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
Formatter::AddInputMessage	All input messages added using AddInputMessage() are processed entirely within the thread from which they were added.

Function	Description
Formatter::AddOutputFormat	The output formats added using AddOutputFormat() are used to specify formatting rules for all input messages added within the same thread.
Formatter::RemoveOutputFormat	RemoveOutputFormat() removes all output formats only for the thread from which it is called.
Formatter::PreloadInFormat	PreloadInFormat() loads input formats only for the thread from which it is called. If PreloadInFormat() is called from the main thread, its results are not available to any other threads.
Formatter::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.
Formatter::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.
Formatter::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.
Formatter::GetFieldAscii / GetFieldString	GetFieldAscii() and GetFieldString() retrieve the results of a parse within the current string.
Formatter::GetFieldAsciiByTag / GetFieldStringByTag	GetFieldAsciiByTag() retrieves the results of a parse within the current string.
Formatter::GetOutMsgCount	GetOutMsgCount() returns the number of output message groups.

Function	Description
Formatter::GetOutMsgGroup	GetOutMsgGroup() returns a specific output message group.
Formatter::GetParsedInMsgCount	GetParsedInMsgCount() returns the number of input messages parsed by the New Era of Networks Formatter within the current thread.
Formatter::GetParsedInMsg	GetParsedInMsg() returns the specified parsed input message resulting from calling parse() within the current thread.
Formatter::GetErrorCode	GetErrorCode() returns the error code of the last error that occurred within the current thread.
Formatter::GetErrorMessage	GetErrorMessage() returns the error message of the last error that occurred within the current thread.
OutMsg::GetSubscriptionList	GetSubscriptionList() returns a list of subscriptions that may be bound to an output message.
OutMsgGroup::GetName	GetName() returns the name of an output message group.
OutMsgGroup::GetMsgCount	GetMsgCount() returns the number of output messages in a group.
OutMsgGroup::GetMsg	GetOutMsg() returns a specific output message within a group.

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. In a Windows 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 New Era of Networks Formatter, the pthreads library must be linked with the final executable.

How Thread Safety Impacts New Era of Networks Formatter Management APIs

Unlike New Era of Networks Formatter Engine APIs, New Era of Networks Formatter Management APIs are not thread-safe and should not be used in a multi-threaded environment.

Automatic Format Conversion

New Era of Networks Formatter contains higher-level APIs that can request New Era of Networks Formatter to 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) required to format the new message.

API and Header Files

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

Header Files

Object Class	Header File	Description
Formatter	formatter.h	Formatter Class
OutMsgGroup	msgs.h	Output message group contained in Formatter Class

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

Formatter Class Functions

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

Return Type	Function	Arguments
Int	GetFieldCount	(char* FriledName)

OutMsg Class Functions

Return Type	Function	Arguments
char*	GetMsgBuffer	none
int	GetMsgLength	none

OutMsgGroup Class Functions

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

ParsedField Class Functions

Return Type	Function	Arguments
char*	GetInfo	none
char*	GetAsciiValue GetStringValue	(int* pDataLength)
char*	GetValue	(int* pDataType, int* pDataLength)
void	GetFmtValue	(char* pBuffer)
int	GetFmtValueLen	none
int	GetByteOffset	none

ParsedMessage Class Functions

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

NNParsedField Class Functions

Return Type	Function	Arguments
const char*	GetFieldAscii GetFieldString	(char * pFieldName, int iIndex)
const char*	GetCurrInFldName	none
const char*	GetCurrOutFldName	none
const char*	GetCurrInFldData	none
const char*	GetCurrInFldAsciiData	none
const int	GetCurrInFldLength	none
const int	GetCurrInFldType	none
char *	GetCurrOutFldData	none
int	GetCurrFldLength	none
int	GetCurrOutFldType	none
char *	GetFieldData	(char* pFieldName, int iIndex = 0)
int	GetFieldDataLength	(char* pFieldName, int iIndex = 0)
int	GetFieldDataType	(char* pFieldName, int iIndex = 0)

Formatter Management API Functions

Return Type	Function	Arguments
NNFMgr *	NNFMgrInit	(DbmsSession *session)
void	NNFMgrClose	(NNFMgr *pNNFMgr)
N/A	NNF_CLEAR	(_p)
const char *	NNFMgrGet PARTICIPANTList	(const NNSesDBBase *session)
const char *	NNFMgrGetGlobal UserList	(const NNSesDBBase *session)
const char *	NNFMgrGetObjectUser List	(const NNSesDBBase *session, const int grantId, const int participantId)
const int	NNFMgrIsOwner	(const NNSesDBBase *session, const int grantId, const int participantId)
const int	NNFMgrCanUpdate	(const NNSesDBBase *session, const int grantId, const int participantId)
const int	NNFMgrHasOwner Permission	(const NNSesDBBase *session, const int grantId, const int participantId)
const int	NNFMgrHasUpdate Permission	(const NNSesDBBase *session, const int grantId, const int participantId)
const int	NNFMgrDeleteObject	(const NNSesDBBase *session, const int grantId)
const int	NNFMgrSetOwnership	(const NNSesDBBase *session, const int grantId, const int participantId)

Return Type	Function	Arguments
const int	NNFMgrSetUpdate Permission	(const NNSesDBBase *session, const int grantId, const int participantId, const bool status = true)
const int	NNFMgrSetNewObject Permissions	(const NNSesDBBase *session, const char *objectKey, const int hierarchyId, const int participantId)
const int	NNFMgrGetParticipant Id	(const NNSesDBBase *session, const int grantId, const int participant)
const int	NNFMgrGetGrantId	(const NNSesDBBase *session, const char *objectKey, const int hierarchyId)
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*)

Return Type	Function	Arguments
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)
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)

Return Type	Function	Arguments
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)
const short	NNFMgrCreateMath ExpCntl	(NNFMgr *pNNFMgr, NFMgrMathExpCntlInfo* const pInfo)
const short	NNFMgrGetMath ExpCntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrMathExpCntlInfo * const pInfo)
const short	NNFMgrAppendSegTo MathExpCntl	(NNFMgr * pNNFMgr, const char* const CntlName, const NNFMgrMathExpCntl Segment Info * const pInfo)
const short	NNFMgrGetSegFrom MathExpCntl	(NNFMgr * pNNFMgr, const char* const CntlName, NNGetOp OpCode, const NNFMgrMathExpCntl Segment Info * const pInfo)
const short	NNFMgrUpdateMath ExpCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrMathExpCntlInfo* const pInfo)
const short	NNFMgrDeleteMath ExpCntl	(NNFMgr *pNNFMgr, const char * const cntlName)

Return Type	Function	Arguments
const short	NNFMgrCreatePrePost FixCntl	(NNFMgr *pNNFMgr, NFMgrPrePostFixCntlInfo* const pInfo)
const short	NNFMgrGetPrePostFix Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrPrePostFixCntlInfo * const pInfo)
const short	NNFMgrUpdatePre PostFixCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrPrePostFixCntlInfo* const pInfo)
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)

Return Type	Function	Arguments
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)
const short	NNFMgrCreate SubStringCntl	(NNFMgr *pNNFMgr, NFMgrSubStringCntlInfo* const pInfo)
const short	NNFMgrGetSubString Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrSubStringCntlInfo * const pInfo)
const short	NNFMgrUpdate SubstringCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrSubStringCntlInfo* const pInfo)
const short	NNFMgrDeleteSub StringCntl	(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)

Return Type	Function	Arguments
const short	NNFMgrCreate TrimCntl	(NNFMgr *pNNFMgr, NFMgrTrimCntlInfo* const pInfo)
const short	NNFMgrGetTrim Cntl	(NNFMgr * pNNFMgr, NNGetOp OpCode, NNFMgrTrimCntlInfo * const pInfo)
const short	NNFMgrUpdate TrimCntl	(NNFMgr *pNNFMgr, const char * const cntlName, NNFMgrTrimCntlInfo* const pInfo)
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)

Return Type	Function	Arguments
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 FormatName, short * const IsRecursive)
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, NNFMgrLiteralInfo* 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)

Return Type	Function	Arguments
const short	NNFMgrDelete Delimiter	(NNFMgr *pNNFMgr, const char * const delimiterName)
const short	NNFMgrCreateUser DefinedType	(NNFMgr *pNNFMgr, const NNFMgrUserDefTypeInfo * const pTypeInfo)
const short	NNFMgrAddName ValuePairs	(NNFMgr *pNNFMgr, const NNFMgrNameValuePairInfo * const pPairInfo)
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, char * pParseName, NNFMgrParseControlInfo * const pParseControlInfo)

Return Type	Function	Arguments
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)
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)

Return Type	Function	Arguments
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)
const short	NNFMgrGetNext Format	(NNFMgr * pNNFMgr, NNFMgrFormatInfo * const pFormatInfo, const NNFMgrFlatFormatInfo * const pFlatFormatInfo)
const short	NNFMgrGetFirstField FromInputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetNextField FromInputFormat	(NNFMgr *pNNFMgr, NNFMgrInFieldInfo * const pInFieldInfo)
const short	NNFMgrGetFirstField FromOutputFormat	(NNFMgr *pNNFMgr, const char * const pFormatName, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetNextField FromOutputFormat	(NNFMgr *pNNFMgr, NNFMgrOutFieldInfo * const pOutFieldInfo)
const short	NNFMgrGetFirstChild Format	(NNFMgr *pNNFMgr, const char * const pParentName, NNFMgrRepeatFormatInfo * const pRepeatFormatInfo)

Return Type	Function	Arguments
const short	NNFMgrGetNextChild Format	(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)
const short	NNFMgrGetFormat GroupInfo	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrCreateFormat Group	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrGetFirst FormatGroup	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrGetNext FormatGroup	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrDeleteFormat Group	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrGetFormatGro upInfo	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrAddFormat GroupItem	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)

Return Type	Function	Arguments
const short	NNFMgrGetFirst FormatGroupItem	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrGetNext FormatGroupItem	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrRemoveForma tGroupItem	(NNFMgr *pNNFMgr, NNFMgrFormatGroupInfo * pInfo)
const short	NNFMgrCreateMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrUpdateMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pOldInfo, NNFMgrMapInfo* const pNewInfo)
const short	NNFMgrGetFirstMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrGetNextMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrDeleteMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrGetMapInfo	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrGetFirstMap	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrCreateField Mapping	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrUpdateField Mapping	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)

Return Type	Function	Arguments
const short	NNFMgrGetFirstField Mapping	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrGetNextField Mapping	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrDeleteField Mapping	(NNFMgr *pNNFMgr, NNFMgrMapInfo* const pInfo)
const short	NNFMgrCreateMap Link	(NNFMgr *pNNFMgr, NNFMgrMapLink* const pLink)
const short	NNFMgrGetFirstMap Link	(NNFMgr *pNNFMgr, NNFMgrMapLink* const pLink)
const short	NNFMgrGetNextMap Link	(NNFMgr *pNNFMgr, NNFMgrMapLink* const pLink)
const short	NNFMgrDeleteMap Link	(NNFMgr *pNNFMgr, NNFMgrMapLink* const pLink)

Shared Libraries

Shared libraries are archived collections of object files. They are installed during the component installation process bin directory. The following is the path to the libraries that must be linked with the application object files:

- On Windows, the shared libraries and DLLs are in {installroot}\bin. The libraries needed to compile custom code are in {installroot} \lib. Libraries are referred to as Dynamic Link Libraries. You can identify shared libraries as files with a .dll extension.
- On UNIX, the libraries are in {installroot}/bin. Depending on the platform, you can identify shared libraries as files with an .so or .sl extension.
- On z/OS, DLL library files are in {installroot} /lib. You can identify shared libraries as files with both .dll and .a extensions.

For more library information, see the example makefiles.

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

WARNING!
It is important to set your library paths correctly, so that the executables can locate them in the specified folder. If you move or delete the libraries, the executables are rendered useless.

Chapter 3

New Era of Networks Infrastructure

This chapter includes the following information:

- *Infrastructure Overview*
- *Standard Template Library*
- *Namespaces*

Infrastructure Overview

The New Era of Networks Infrastructure is a framework that allows programs to be ported more easily. The New Era of Networks Infrastructure provides a wrapper around Standard Template Library (STL), namespaces, and streams. The resulting code is portable and platform-independent.

The New Era of Networks Infrastructure is used by New Era of Networks Rules. Adapter developers must understand its applicability to the code they write, specifically management of the Standard Template Library (STL), initialization of the infrastructure for the e-ADK, and namespace guidelines.

Each of these topics is discussed in the following sections.

Standard Template Library

Versions of the Standard Template Library (STL) on different compilers can require different ways of instantiating types. The New Era of Networks Infrastructure provides abstractions for instantiating types correctly and provides wrapper functions for portions of the STL implementation.

Use the following guidelines to determine the vendor and version of the C++ Standard Template Library (STL) to compile against:

- If the STL is supported natively by a compiler, that version of the STL is used on that platform and compiler, without exception. The platforms currently include Windows with Visual C++, HP-UX with the aCC compiler, and Solaris with the Sun WorkShop 6.0 compiler.
- For z/OS, STLport is supported with the C++ compiler.

The New Era of Networks Infrastructure provides macros for the STL Containers.

The STL_STRING macro is made available to a source module by including any header file from the Infrastructure. Use the Infrastructure's StringProxy class as a template argument in place of STL_STRING to prevent problems with the length of generated template names.

The containers are included in header files from the Infrastructure of the form INFR/STL_CONTAINER.h, where CONTAINER is one of BITSET, DEQUE, HASHMAP, HASHSET, LIST, MAP, PAIR, QUEUE, SET, STACK, or VECTOR.

The macros prefix container names with an appropriate namespace, if needed, and automatically take care of differences between STL container declarations. All containers and functions are in the STD_NAMESPACE, which may or may not be std::, depending on the STL used.

Example

To include and use an STL map to correlate a string to an integer, the following code could be used:

```
#include <INFR/STL_MAP.h>
```

```
typedef STL_MAP(NNSY_NAMESPACE StringProxy, int32_t)
        OurMapType;
```

The predicate and allocator are not used in the preceding example. For a different predicate, use the predicate-form of the macro. Here is an example:

```
#include <INFR/STL.MAP.h>
typedef STL_MAP_PRED(
        NNSY_NAMESPACE StringProxy,
        int32_t,
        STD_NAMESPACE greater<StringProxy>) OurMapType;
```

Configuring the New Era of Networks Infrastructure

The New Era of Networks Infrastructure must be configured to use a specified STL when a native STL version is not available. The different STL versions must be configured for use in multi-threaded applications. For z/OS, STLport is supported by New Era of Networks Infrastructure.

Namespaces

Many New Era of Networks components are wrapped within a pair of namespaces on compilers that support namespaces. Namespaces partition the global namespace so that classes and functions from one component do not cause a problem with classes and functions with the same name from another component. The outer namespace is NNSY. The inner namespace is the name of the component, for example, INFR for the Infrastructure and NDO.

The basic rules of namespaces are:

- If you are writing a header file for a library, do not open namespaces. Opening a namespace is accomplished by using a *using* or *using namespace* declaration.
- If you are writing a source module, do not open any Namespaces until all header files are included. Opening namespaces before often causes errors in header files.

- If you are writing a header file for an application, namespaces can be opened, but you must then ensure that ambiguities do not occur in other header files that may be included after the current header file. In this case, you might want to have a single header file control the opening of the namespaces and include all other header files in this header file.
- Global operators, for example: streaming operators and equivalence operators, should be placed in the global namespace.

The Infrastructure provides a set of macros for working with namespaces and each component also provides a set of macros for working with the specific namespaces declared in the component.

Using Namespaces

Library header files should use names from New Era of Networks components.

Example:

```
#include <INFR/RCPointer.h>
#include <NDO/NNDOObject.h>

class MyClass
{
protected:

  NNSY_INFR_NAMESPACE RCPointer<NNSY_NDO_NAMESPACE NNDOObject>
     m_pNDO;

public:

  NNSY_NAMESPACE e_SF Foo();
};
```

The following is an example application header file.

```
MyClass.h:

#include <INFR/RCPointer.h>
#include <NDO/NNDOObject.h>

//preferred method:
#include <OurAppSymbols.h>
// or, less preferable method
#include <OurAppNamespaces.h>

class MyClass
{
protected:

  RCPointer<NNDOObject> m_pNDO:

public:
```

```
  e_SF Foo():
};

OurAppSymbols.h:

// Names can be selectively pulled that you are interested in
// without opening entire namespaces. This is the safer,
// preferred method.

#include <INFR/RCPointer.h
#include <NDO/NNDOObject.h>

USING_SYMBOL(NNSY::e_SF)
USING_SYMBOL(NNSY::INFR::RCPointer)
USING_SYMBOL(NNSY::NDO::NNDOObject)
//...etc...

OurAppNamespaces.h:

// You can open everything. Make sure the namespaces that you
// are going to open exist. A header file may not be included
// that declares the namespaces that you are opening.

BEGIN_NAMESPACE(NNSY)
BEGIN_NAMESPACE(ND0)
END_NAMESAPCE(NDO)
BEGIN_NAMESPACE(INFR)
END_NAMESPACE(INFR)
END_NAMESPACE(NNSY)

// The application guarantees that we do not conflict with any
// of the names in the NNSY namespace, or its contained NDO or
// INFR namespaces. Further, no other component conflicts with
// any of the names.

USING_NAMESPACE(NNSY)
USING_NAMESPACE(NDO)
USING_NAMESPACE(INFR)

Source modules usually look like the following:

#include <MyClass.h>
```

```
#include <OtherStuff.h>

USING_NAMESPACE_STD
USING_NAMESPACE(NNSY)
USING_NAMESAPCE(INFR)
USING_NAMESPACE(NDO)

e_SF
MyClass::Foo()
{
   RCPointer<NNDOObject> pTempNDO = m_pNDO;

   m_pNDO = getNDO();    //etc.

   return SF_Success:
}
```

Using Streams

To allow use of the standard C++ streams for compilers that support them, and also allow the use of old streams in compilers that do not, the Infrastructure provides a set of typedefs and macros to abstract the stream types. On systems that support both stream types, the standard C++ streams are used by Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms by default.

The stream typedefs are in the NNSY namespace. Stream typedefs must follow the guidelines for using namespaces.

For non-platform independent code, the macros do not need to be used. Either old streams or new streams can be directly used.

The following is an example library header file.

```
#include <INFR/Streams.h>

class MyClass
{
public:

   virtual void dump(NNSY_NAMESPACE OStream& os) const;
};
inline NNSY_NAMESPACE OStream&
```

```
operator << (NNSY_NAMESPACE OStream& os, const MyClass& rhs)
{
   rhs.dump(os);
   return os;
}
```

The macros are important when both old streams and new streams are mixed. This situation occurs frequently with third party libraries that use old streams. In this situation, the two stream types can be mixed if the namespaces are explicitly used for every name.

To use old streams, each name must be prefixed by `::`, for example, `::ostream`.

To use new streams, `std::` must be prefixed to each name, for example, `STD_NAMESPACE cout`. Using the STD_NAMESPACE macro makes the code portable when the new style of streams are used only on platforms that support them; the old style of streams are used otherwise.

Chapter 4

New Era of Networks Formatter APIs

Formatter Class Member Functions

Constructor

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

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

Note:
THREAD SAFETY: For multi-threaded applications, the constructor should be called by the main thread before spawning threads to perform parsing or reformatting.

Syntax

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

Parameters

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

Constructor

This overloaded version of the Constructor is used when there are user-defined type input field validation callback objects to register with New Era of Networks Formatter. For information on user callbacks in general, see *User Callback API Functions* on page 108. For information on user-defined type input field validation, see *User-Defined Data Type Management API Structures* on page 181.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
DatabaseSession Object	DbmsSession*	Input	Name of the 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

Sample code is available in the /examples/NNSYRF/ directory in the msgtest.cpp and apitest.cpp files.

See Also

OpenDbmsSession

Formatter Destructor

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

Note:
THREAD SAFETY: For multi-threaded applications, the destructor should only be called by the main thread after all threads complete parsing or reformatting.

Syntax

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

Parameters

None.

Remarks

Formatter::~Formatter must be called after Formatter::Formatter and after all New Era of Networks Formatter processing is complete.

Return Value

None

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

Example

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

ResetSession

Closes or changes the database session used by New Era of Networks Formatter.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
DatabaseSessionObject	DbmsSession*	Input	Null pointer or pointer to a new, open database session.

Remarks

Use this function to do the following:

1. Let Formatter know the database session is closed:

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

   then close the database session.

2. Open a new database session and tell Formatter to use the new connection.

Return Value

None

Example

```
// Open a database session...
DbmsSession*myDbmsSession;
myDbmsSession = OpenDbmsSession("format_session_name",DB25);

// Construct a Formatter instance.
Formatter formatter(myDbmsSession);

// Close database session and inform Formatter.
CloseDbmsSession(myDbmsSession);
```

```
formatter.ResetSession((DbmsSession *)0);

// Open a new database session and inform Formatter.
myDbmsSession = OpenDbmsSession("new_format_session_name",
DB25);
formatter.ResetSessionResetSession(myDbmsSession);
```

See Also

OpenDbmsSession

CloseDbmsSession

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 New Era of Networks Formatter database.

Note:
THREAD SAFETY: All input messages added using AddInputMessage() are processed entirely within the thread from which they were added.

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. When the error-handling routines are used to return the error message, the message indicates which parameter has 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.

The message buffer passed into this function must be allocated by the user, not by a New Era of Networks Formatter API call, such as OutMsgGroup::GetMsg. All New Era of Networks Formatter APIs can change buffers allocated by any other New Era of Networks Formatter APIs.

Return Value

None

Example

Sample code is available in the /examples directory in the apitest.cpp and msgtest.cpp files.

See Also

Parse

Reformat

PreloadInFormat

AddInputMessage

This overloaded version of AddInputMessage() directs New Era of Networks 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 New Era of Networks Formatter database.

Syntax

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

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. The caller must destroy the buffer only after the Formatter object has been destroyed. This version of the method will help with memory use and performance.

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.

Return Value

None

See Also

Parse

PreloadInFormat

AddOutputFormat

Tells New Era of Networks Formatter to create an output message of the type specified by FormatName when reformatting. The optional parameters mapName and mapVersion are used to identify a specific version of a map object to use for mapping input fields to output fields. If these parameters are omitted, default input field to output field mapping occurs.

Note:
THREAD SAFETY: All output formats added using AddOutputFormat() are used to specify formatting commands for all input messages added within the same thread.

Syntax

```
int Formatter::AddOutputFormat(char* FormatName)
                               [const char* mapName],
                               [int mapVersion]);
```

Parameters

Name	Type	Input/ Output	Description
FormatName	char*	Input	Identifier to retrieve a format configuration definition from the database.
mapName	const char*	Input	Optional user-defined name for map. If mapName is not provided, defaults to NULL, and the default mapping specified in the format definition is used.
mapVersion	int	Input	Optional version number of the map defined by mapName. If mapVersion is not provided, defaults to zero (0), and the most recent version is used.

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 has a bad value.

Return Value

Returns an integer value that is a zero-based index into an internal array that contains the output format and map information. Each call to AddOutputFormat returns increasingly higher values, assuming that the formatName parameter is different on each call.

Returns a value of -1 (error condition) if one of the following is true:

- The formatName parameter is NULL or has a zero length or a length greater than 120 characters.
- The mapping object identified by the mapName and mapVersion parameters does not exist in the database.

Note:
An application can call AddOutputFormat more than once before calling Reformat, causing the Formatter to generate one output message for each of the output formats specified by the AddOutputFormat calls.

Example

```
int retcode;
retcode = pFormatter->AddOutputFormat( "OUT_1", NULL, 0 );
// retcode should be 0
retcode = pFormatter->AddOutputFormat( "OUT_2", NULL, 0 );
// retcode should be 1
retcode = pFormatter->AddOutputFormat( "OUT_3", NULL, 0 );
// retcode should be 2
retcode = pFormatter->AddOutputFormat( "OUT_2", NULL, 0 );
// retcode should be 1 because "OUT_2" is already in the list
pFormatter->Reformat();
```

See Also

PreloadOutFormat

RemoveOutputFormat

Removes the output format from the list of output formats to be reformatted.

Note:
THREAD SAFETY: All output formats added using AddOutputFormat() are used to specify formatting commands for all input messages added within the same thread.

Syntax

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

Parameters

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

Remarks

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

Return Value

Void

PreloadInFormat

Preloads an input format into memory. If you do not use this function call, input formats are 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 New Era of Networks Formatter, calling it allows the application programmer to control where time is spent to access the database during the application.

Note:
THREAD SAFETY: PreloadInFormat() only loads input formats for the thread from which it is called. If this function is called from the main thread, its results are not available to any other threads.

Syntax

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

Parameters

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

Return Value

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

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

Example

You built input formats IFFormat1, IFFormat2, and IFFormat3 using the Formatter GUI or the Format Management API functions, and your New Era of Networks Formatter application uses these formats. You can preload these format definitions

prior to calling the other New Era of Networks 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

Preloads an output format into memory. If you do not use this function call, output formats are 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 New Era of Networks Formatter, calling it allows the application programmer to control where time is spent to access the database during the application.

Note:
THREAD SAFETY: PreloadOutFormat() only loads output formats 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.

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 OFFormat1, OFFormat2, and OFFormat3 using the Formatter GUI 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
FormattermyFormatter;

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

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

See Also

PreloadInFormat

StartDebug

Initializes the parse debugger. 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,
                NNSY_NAMESPACEOStream& pOutputBuffer);
```

Parameters

Name	Type	Input/ Output	Description
pDebugCategory	const NN_DEBUG_CATEGORY	Input	Indicates which New Era of Networks 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 currently has only one value (PARSE), but it is extendable for the future.

NN_VERBOSE_LEVEL currently has only one value (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

Stops the debugger and cleans up memory used by the debugging process.

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

Syntax

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

Parameters

None

Return Value

None

See Also

StartDebug

SetInputCodeSet

Provides a mechanism for setting the code set associated with the input message.

Syntax

```
int Formatter::SetInputCodeSet(const char*CodeSet)
```

Parameters

Name	Type	Input/ Output	Description
CodeSet	const char*	Input	A NULL-terminated string which identifies a code set or code set alias.

Remarks

For a listing of valid code sets, see the ***System Management Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid code set value was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set code set for input message
myFormatter.SetInputCodeSet ("ibm-850");

//Set code set for outbound message
myFormatter.SetOutputCodeSet ("ibm-1252");
```

See Also

GetInputCodeSet

SetInputCodeSet

Provides a mechanism for setting the code set associated with the input message. This overloaded method differs from the previous SetInputCodeSet in that it receives a character code set identifier.

Syntax

```
int Formatter::SetInputCodeSet(int CodeSet)
```

Parameters

Name	Type	Input/ Output	Description
CodeSet	int	Input	A code set identifier.

Remarks

For a listing of valid code sets, see the ***System Management Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid code set value was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set code set for input message
myFormatter.SetInputCodeSet (819);

//Set code set for outbound message
myFormatter.SetOutputCodeSet (850);
```

See Also

GetInputCodeSet

SetOutputCodeSet

Provides a mechanism for setting the code set associated with the output message.

Syntax

```
int Formatter::SetOutputCodeSet(const char *CodeSet)
```

Parameters

Name	Type	Input/ Output	Description
CodeSet	const char*	Input	A NULL-terminated string which identifies a code set or code set alias.

Remarks

For a listing of valid code sets, see the ***System Management Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid code set value was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set code set for input message
myFormatter.SetInputCodeSet ("ibm-850");

//Set code set for outbound message
myFormatter.SetOutputCodeSet ("ibm-1252");
```

See Also

GetOutputCodeSet

SetOutputCodeSet

Provides a mechanism for setting the code set associated with the input message. This overloaded method differs from the previous SetOutputCodeSet in that it receives a character code set identifier.

Syntax

```
int Formatter::SetOutputCodeSet(int CodeSet)
```

Parameters

Name	Type	Input/ Output	Description
CodeSet	int	Input	A code set identifier.

Remarks

For a listing of valid code sets, see the ***System Management Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid code set value was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set code set for input message
myFormatter.SetInputCodeSet (850);

//Set code set for outbound message
myFormatter.SetOutputCodeSet (819);
```

See Also

GetOutputCodeSet

SetInputLocale

Provides a mechanism for setting the locale of an input message.

Syntax

```
int Formatter::SetInputLocale(const char *Locale)
```

Parameters

Name	Type	Input/ Output	Description
Locale	const char *	Input	A NULL-terminated string representing a locale, such as en_US.

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid locale was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set locale for input message
myFormatter.SetInputLocale ("en_US");
```

See Also

GetInputLocale

SetOutputLocale

Provides a mechanism for setting the locale of an output message.

Syntax

```
int Formatter::SetOutputLocale(const char *Locale)
```

Parameters

Name	Type	Input/ Output	Description
Locale	const char *	Input	A NULL-terminated string representing a locale, such as en_US.

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns 1 if the code was successfully set, zero (0) if not successful. A return value of zero normally indicates that an invalid locale was provided.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Set locale for output message
myFormatter.SetOutputLocale ("en_US");
```

See Also

GetOutputLocale

GetInputCodeSet

Provides a mechanism for retrieving the code set of the input message.

Syntax

```
const char *Formatter::GetInputCodeSet()
```

Parameters

None

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns the current input code set. If the code was not explicitly set to SetInputCodeSet, the code set of the host system is used.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Write input code set to standard out
cout << myFormatter.GetInputCodeSet() << endl;
```

See Also

SetInputCodeSet

GetOutputCodeSet

Provides a mechanism for retrieving the code set of the output message.

Syntax

```
const char *Formatter::GetOutputCodeSet()
```

Parameters

None

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns the current output code set. If the code was not explicitly set to SetOutputCodeSet, the code set of the input code is used.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Write output code set to standard out
cout << myFormatter.GetOutputCodeSet() << endl;
```

See Also

SetInputCodeSet

SetOutputCodeSet

GetInputLocale

Provides a mechanism for retrieving the locale of an input message.

Syntax

```
const char *Formatter::GetInputLocale()
```

Parameters

None

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns the current input locale. If the locale was not explicitly set to SetInputLocale, the locale of the host system is used.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Write input locale to standard out
cout << myFormatter.GetInputLocale() << endl;
```

See Also

SetInputLocale

GetOutputLocale

Provides a mechanism for retrieving the locale of an output message.

Syntax

```
const char *Formatter::GetOutputLocale()
```

Parameters

None

Remarks

For a listing of valid locales, see the ***User's Guide.***

Return Value

Returns the current output locale. If the locale was not explicitly set to SetOutputLocale, the locale of the input message is used.

Example

```
//Construct Formatter Instance
Formatter myFormatter;

//Write output locale to standard out
cout << myFormatter.GetOutputLocale() << endl;
```

See Also

SetInputLocale

SetOutputLocale

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: Parse() only parses the input messages added with AddInputMessage() within the thread from which it is called. The parsed message is only available from within this thread.

Syntax

```
int Formatter::Parse();
```

Parameters

None

Remarks

Parse() can be called without reformatting the input messages into output messages. New Era of Networks 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 GetValue().

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!
Do not 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, the result is 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

Sample code is available in the /examples/NNSYRF directory in the apitest.cpp file.

See Also

AddInputMessage

GetFieldAscii

Reformat

GetValue

Reformat

Translates input messages that are added using AddInputMessage() into output messages that are specified using AddOutputFormat(). Output messages are formatted into dynamically allocated character buffers.

Note:
THREAD SAFETY: 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.

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

When a field is 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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

AddOutputFormat

Parse

GetFieldAscii / GetFieldString

Allows direct access to field contents based on the field name. GetFieldAscii() must be called after Parse().

Note:
THREAD SAFETY: GetFieldAscii() retrieves the results of a Parse() within the current thread.

Syntax

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

Parameters

Name	Type	Input / Output	Description
FieldName	char*	Input	NULL-terminated string specifying the field name.
SequenceNumber	int	Input	Zero-based index specifying the instance number of a repeating group of fields.

Remarks

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

Return Value

Returns a pointer to a NULL-terminated ASCII representation of the field 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

An input format IFFormat has a field named F1. Parse a message with this input format to get the value of the input field F1. The following sequence of New Era of Networks Formatter function calls includes a call to GetFieldAscii():

```
// Construct Formatter instance
FormattermyFormatter;

// Declare variables.
charmyBuffer[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

GetFieldCount

Returns the number of instances in a repeating group, given the name of a field in that group.

Syntax

```
const int NNParsedFields::GetFieldCount(char* fldName)
```

Parameters

Name	Type	Input/ Output	Description
fldName	char *	Input	Name of a field in the parsed data message.

Return Value

Returns a positive integer value representing the number of instances in the repeating group. Returns -1 if the field name is not found.

Example

```
Formatter *pFormatter = new Formatter (pSession);
PFormatter->GetFieldCount("fld_1");
```

See Also

GetFieldString

GetFieldAsciiByTag / GetFieldStringByTag

Allows direct access to a tagged field contents by tag name. This must be called after Parse().

Note:
THREAD SAFETY: GetFieldAsciiByTag() retrieves the results of a Parse() within the current thread.

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 field.
SequenceNumber	int	Input	Zero-based index specifying the instance number of a repeating group of fields.

Remarks

GetFieldStringByTag() performs the same function that GetFieldAsciiByTag does, but it is a portable version that should be used if porting across different 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

An input format IFFormat has a field named F1. Field F1 is a tagged field, and the value of the tag is TagForF1. Parse a message with this input format to get the value of the input field F1 and to refer to F1 by its tag value, not its name. The following sequence of New Era of Networks Formatter function calls includes 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

Returns the number of output message groups in the Formatter object for the current reformat.

Note:
THREAD SAFETY: GetOutMsgCount() returns the resulting number of output message groups after calling Reformat() within the current thread.

Syntax

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

Parameters

None

Return Value

GetOutMsgCount() is called after Reformat and 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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

AddOutputFormat

GetOutMsgGroup

Returns a pointer to the group of output messages for a particular format.

Note:
THREAD SAFETY: GetOutMsgGroup() returns the specified resulting output message group from calling Reformat() within the current thread.

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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetMsg

GetParsedInMsgCount

Returns the number of input messages parsed by New Era of Networks Formatter. The number should equal the number of input messages added by AddInputMessage().

Note:
THREAD SAFETY: GetParsedInMsgCount() returns the number of input messages parsed by Formatter within the current thread.

Syntax

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

Parameters

None

Return Value

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest and apitest.cpp files.

See Also

AddInputMessage

GetParsedInMsg

Returns a pointer to a parsed input message at the specified index.

Note:
THREAD SAFETY: GetParsedInMsg() returns the specified parsed input message resulting from calling Parse() within the current thread.

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 is 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

Sample code is available in the /examples /NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

AddInputMessage

SetUserTypeValidationOn

Turns on user-defined type input field validation. 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: No matter what thread this function is called from, it sets the validation state for all threads of a Formatter instance.

Syntax

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

Parameters

None

Return Value

None.

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

See Also

SetUserTypeValidationOff

UserTypeValidationIsOn

SetUserTypeValidationOff

Turns off user-defined type input field validation. 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: No matter what thread this function is called from, it sets the validation state for all threads of a Formatter instance.

Syntax

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

Parameters

None

Return Value

None.

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

SetUserTypeValidationOn

UserTypeValidationIsOn

UserTypeValidationIsOn

Returns the current state of user-defined type input field validation.

Note:
THREAD SAFETY: 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.

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().

See Also

SetUserTypeValidationOn

SetUserTypeValidationOff

OutMsg Class Member Functions

GetMsgBuffer

Returns a pointer to the buffer containing message text for a particular output message.

Note:
THREAD SAFETY: GetMsgBuffer() returns a pointer to the buffer within the current thread.

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 should be modified by the calling application. To maintain persistence beyond 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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

AddInputMessage

GetMsgLength

GetMsgLength

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

Note:
THREAD SAFETY: GetMsgLength() returns the length (in bytes) of the internal message buffer within the current thread.

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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetMsgBuffer

GetSubscriptionList

Returns the RulesSubscriptionList for the OutMsg. This will be NULL unless this OutMsg was created by a Field Evaluation control which called New Era of Networks Rules and then performed a reformat action.

For more information on subscription lists, see ***New Era of Networks Rules Programming Reference*** and the ***User's Guide***.

Note:
THREAD SAFETY: GetSuscriptionList() returns the subscription list for the output message within the current thread.

Syntax

```
RulesSubscriptionList* OutMsg::GetSubscriptionList();
```

Parameters

None

Return Value

A pointer to the subscription list to perform for the OutMsg. NULL if the OutMsg was not created with a Field Evaluation control.

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetMsgBuffer

GetMsgLength

OutMsgGroup Class Member Functions

GetMsg

Returns a pointer to an output message in an output message group.

Note:
THREAD SAFETY: GetMsg() returns a pointer to an output message within the current thread.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
index	int	Input	Index into OutMsg array inside OutMsgGroup. Zero (0) is the first element.

Return Value

Returns a pointer to the OutMsg at the index position in the internal outMsg array; NULL if not present.

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

GetName

Returns the name of an output message group.

Syntax

```
char* OutMsgGroup::GetName()
```

Parameters

None

Remarks

If the OutMsgGroup was generated by a Field Evaluation Control doing a reformat, the OutMsgGroup will be named based on the output format of the reformat action.

Return Value

Returns the name of the output format used to generate this OutMsgGroup.

See Also

GetMsgCount

GetMsg

GetMsgCount

Returns the number of messages in an output message group.

Note:
THREAD SAFETY: GetMsgCount() returns the number of messages in an output message group within the current thread.

Syntax

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

Parameters

None

Return Value

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

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetMsg

GetParsedOutMsgCount

Returns count of output messages constructed by New Era of Networks Formatter.

Note:
THREAD SAFETY: GetParsedOutMsgCount() returns the number of messages in an output message group within the current thread.

Syntax

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

Parameters

None

Return Value

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

See Also

GetMsg

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 can be used.

Note:
THREAD SAFETY: GetParsedOutMsg() returns the number of messages in an output message group within the current thread.

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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetMsg

GetFieldComp

GetMsgComp

ParsedField Class Member Functions

GetInfo

Returns a pointer to the name of the field in a parsed message.

Note:
THREAD SAFETY: GetInfo() returns a pointer to the name of the field in a parsed message within the current thread.

Syntax

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

Parameters

None

Remarks

Do not modify this memory. Allocate your own memory and copy the name if you wish to modify its value or need the data past the lifetime of the ParsedField.

Return Value

Returns a pointer to the name of the specified field.

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetParsedInMsg

GetFieldComp

GetAsciiValue / GetStringValue

Returns the ASCII value of the specified field in a parsed message. GetStringValue() performs the same function as GetAsciiValue, but it is a portable version that should be used if porting across different types of platforms.

Note:
THREAD SAFETY: GetAsciiValue() returns the ASCII value of the specified field in a parsed message within the current thread.

Syntax

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

Parameters

Name	Type	Input / Output	Description
pDataLength	int*	Output	Address of integer variable to receive data length. Must not be NULL.

Remarks

Do not modify this memory. Allocate your own memory and copy the name if you wish to modify its value or need the data past the lifetime of the ParsedField.

Return Value

Returns the value of the specified field in ASCII format.

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

See Also

Parse

GetParsedInMsg

GetFieldComp

GetValue

Returns the value of the current field in a parsed message in its original data type. This function returns the buffer of a parsed message.

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

Note:
THREAD SAFETY: GetValue() returns the value of the specified field in a parsed message within the current thread.

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. Must not be NULL.
pDataLength	int*	Output	Address of integer variable to receive data length. Must not be NULL.

Remarks

Do not modify this memory. Allocate your own memory and copy the name if you wish to modify its value or need the data past the lifetime of the ParsedField.

Return Value

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

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetFmtValLen

Returns length of parsed submessage in bytes.

Note:
THREAD SAFETY: GetFmtValLen() returns the value of the specified field in a parsed message within the current thread.

Syntax

```
int ParsedField::GetFmtValLen()
```

Parameters

None

Return Value

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

See Also

Parse

GetParsedInMsg

GetFieldComp

GetFmtVal

Returns a buffer containing the parsed submessage.

Note:
THREAD SAFETY: GetFmtVal() returns the component field within the current thread.

Syntax

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

Parameters

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

Return Value

Buffer containing the Formatter submessage.

See Also

Parse

GetParsedInMsg

GetMsgComp

GetByteOffset

Returns byte offset in the original message where the field was found.

Note:
THREAD SAFETY: GetByteOffset() returns the value of the specified field in a parsed message within the current thread.

Syntax

```
int ParsedField::GetByteOffset()
```

Parameters

None

Return Value

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

See Also

Parse

GetParsedInMsg

GetFieldComp

ParsedMessage Class Member Functions

GetCompCount

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

Note:
THREAD SAFETY: GetCompCount() returns the number of components in a parsed message within the current thread.

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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetParsedInMsg

GetMsgComp

Returns the message component at the specified index.

Note:
THREAD SAFETY: GetMsgComp() returns the message component within the current thread.

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 returned 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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetParsedInMsg

GetFieldComp

GetInfo

Returns the format name of the parsed message.

Note:
THREAD SAFETY: GetInfo() returns the format name of the parsed message within the current thread.

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.

Remarks

Do not modify this memory. Allocate your own memory and copy the name if you wish to modify its value or need the data past the lifetime of the ParsedField.

Return Value

Returns the format name of the parsed message.

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetParsedInMsg

GetFieldComp

Returns the component field at the index specified.

Note:
THREAD SAFETY: GetFieldComp() returns the component field within the current thread.

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

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

Parse

GetParsedInMsg

GetMsgComp

GetFmtValLen

Returns the length in bytes of the parsed submessage.

Note:
THREAD SAFETY: GetFmtValLen() returns the component field within the current thread.

Syntax

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

Parameters

None

Return Value

Returns an integer containing the length in bytes of the formatted submessage.

See Also

Parse

GetParsedInMsg

GetMsgComp

GetFmtVal

GetFmtVal

Returns a buffer containing the parsed submessage.

Note:
THREAD SAFETY: GetFmtVal() returns the component field within the current thread.

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.

See Also

Parse

GetParsedInMsg

GetMsgComp

User Callback API Functions

The User Callback API provides a flexible mechanism for defining functions New Era of Networks 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 the nnuserfunction.h header file.

You define user callbacks as methods in a callback class derived from a defined abstract base class. Objects of the callback class are then passed to New Era of Networks Formatter at construction.

In addition to New Era of Networks Formatter-created data, user callbacks can also be passed user-defined parameters. Static data is defined at format definition time. Dynamic data is created by the user application at run time.

The callback and lookup methods described in this section must be thread-safe because New Era of Networks Formatter can have multiple threads running. For purposes of this section, thread safety is satisfied by not using static or global variables.

User Callback API Structure

New Era of Networks Formatter performs input field validation of a user-defined type after an input message is completely parsed. Input parse controls and output format controls can be specified in terms of a user-defined type, but only input fields are subject to 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 each field set to NULL. You must allocate an array of NameValuePairs with one additional element than those required to hold your data, and call NameValuePair::MakeNull on the last pair to mark the end of the array.

The functions on which validation is based are described in the User Callback API functions, the Formatter Member functions, and Format Management APIs. If you create callback objects, put them in a collection, and pass the collection to the Formatter constructor, validation occurs for input fields with parse controls defined in terms of a user-defined type.

Use the following Formatter constructor version. The second argument is a collection of callback objects described in the User Callback API functions.

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

The validation functions return zero (0) for validation failure and non-zero for validation success. If any validation callback returns failure, New Era of Networks Formatter fails the entire parse, just as it does now with its own internal validation.

By default, validation is ON. You cannot turn validation on or off for individual fields. Validation is on or off for all fields in a message that are defined in terms of a user-defined type. To turn validation off or on, or to check the current validation state, use one of the following three Formatter class member functions.

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

If you are using a multi-threaded version of New Era of Networks Formatter, these three functions apply to all threads globally. You cannot turn validation off for some threads and on for others.

Users can create objects of the derived class to support validation of all user types. Validation uses only NNDBFieldsUserFunction(). Users derive a callback class from NNDBFieldsUserFunction() and define all the pure virtual methods from that class.

The following code illustrates user-defined type input field validation. For a more complete example, see msgtest.cpp in the examples/NNSYRF directory.

```
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();
...
```

NameValuePair Structure

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

Syntax

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

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

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

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

private:
    int strLength;
};
```

Parameters

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

NameValuePairMember Functions

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

NameValuePair (Default Constructor)

Default constructor. Length of name and value is set to [NAME_LENGTH+1].

Syntax

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

Parameters

None

NameValuePair (Alternate Constructor)

Alternate constructor. Length of name and value is 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	Reference to name value pair to copy.

NameValuePair (Assignment Operator)

Assignment operator.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
rhs	const NameValuePair&	Input	Reference to name value pair to copy.

NameValuePair (Destructor)

Destructor.

Syntax

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

Parameters

None

MakeNull

De-allocates 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 the name and value to the input values.

Syntax

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

Parameters

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

Defining a User Callback Class

User callback functions are derived from abstract base classes. Depending on the feature you are working with, derive from one of the following abstract base classes. Derive from the abstract base class that declares the minimum set of methods that must be defined for your derived class.

The callback class hierarchy is:

```
NNUserFunction
      NNGenericUserFunction abstract base class
           <UserDerivedCallbackClass>
      NNDBUUserFunction abstract base class
           <UserDerivedCallbackClass>
      NNDBFieldsUserFunction abstract base class
           <UserDerivedCallbackClass>
```

NNUserFunction

NNUserFunction is the class from which all callback classes are derived. It provides a general class for passing callback objects to NNFunctionKeyPairCollection.

Syntax

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

Parameters

None

Remarks

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

NNGenericUserFunction

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	Return Type	Arguments
Callback	virtual int	()
Callback	virtual int	(NameValuePair* nameValuePairArray)
Callback	virtual int	(void* userRuntimeData)
Callback	virtual int	(NameValuePair* nameValuePairArray, void* userRuntimeData)
RuntimeDataLookup	virtual void	(const char* parmName)

NNGenericUserFunction Member Functions

Callback (No Parameters)

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,userRuntimeData)

A New Era of Networks Formatterfeature 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 (nameValuePairArray,)

A New Era of Networks 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 New Era of Networks 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

New Era of Networks 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	const char*	Input	Name passed to RuntimeDataLookup(), which can use it to determine the data address to pass back.

NNDBUserFunction

Derive from this class if the feature 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	Return Type	Arguments
Callback	virtual int	(const DbmsSession& dbSession)
Callback	virtual int	(const DbmsSession& dbSession, NameValuePair* nameValuePairArray)
Callback	virtual int	(const DbmsSession& dbSession, void* userRuntimeData)

Name	Return Type	Arguments
Callback	virtual int	(const DbmsSession& dbSession, NameValuePair* nameValuePairArray, void* userRuntimeData)
RuntimeDataLookup	virtual void	(const char* parmName)

NNDBUserFunction Member Functions

Callback (dbSession)

A New Era of Networks Formatterfeature 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 New Era of Networks 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.
nameValuePairArray	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

Callback (dbSession, nameValuePairArray, userRuntimeData)

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

Syntax

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

Parameters

Name	Type	Input/ Output	Description
dbSession	constDbmsSession&	Input	Handle to the current database session.
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, userRuntimeData)

Objects derived from this class call 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	const DbmsSession&	Input	Handle to the current database session.
userRuntimeData	void*	Input	User-allocated runtime data obtained by the RuntimeDataLookup() method.

RuntimeDataLookup

New Era of Networks 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	const char*	Input	parmName is passes it to RuntimeDataLookup(), which can 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

Name	Return Type	Arguments
Callback	virtual int	(const DbmsSession& dbSession, const NNParsedFields& parsedFields)
Callback	virtual int	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, NameValuePair* nameValuePairArray)
Callback	virtual int	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, void* userRuntimeData)
Callback	virtual int	(const DbmsSession& dbSession, const NNParsedFields& parsedFields, NameValuePair* nameValuePairArray, void* userRuntimeData)
RuntimeDataLookup	virtual void	(const char* parmName)

NNDBFieldsUserFunction Member Functions

Callback (dbSession, parsedFields)

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)

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 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 NNParsed Fields&	Input	The set of all parsed fields.
nameValuePairArray	NameValuePair*	Input	Array of name/value pairs retrieved from the database.

Callback (dbSession, parsedFields, nameValuePairArray, userRuntimeData)

Objects derived from this class call this method if there are both name/value pairs and 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,
        NameValuePair* nameValuePairArray,
        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.
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)

Objects derived from this class call 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

New Era of Networks 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	const char*	Input	parmName is passed it to RuntimeDataLookup(), which uses it to determine the data address to pass back.

User Callback Lookup Interface

When New Era of Networks Formatter calls a user callback, it attempts to look up the address of a callback object in the collection of callback objects passed to New Era of Networks Formatter at construction. The collection of objects holds object/key pairs. New Era of Networks Formatter obtains a key, does a lookup on the object collection with that key, and receives the address of the corresponding callback object. New Era of Networks 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 New Era of Networks 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 );
```

Member Functions

Name	Return Type	Arguments
AddPair	int	(NNUserFunction* funcObject, const char* key)
Lookup	NNUserFunction*	(const char* key)

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 callback class derived from an abstract base class.
key	const char*	Input	Key used to look up funcObject.

Return Value

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

Lookup

Obtains a pointer to the required callback object.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
key	const char*	Input	Key used to look up funcObject.

New Era of Networks Formatter Error Handling

GetErrorCode

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

Note:
THREAD SAFETY: GetErrorCode() returns the error code of the last error that occurred within the current thread.

Syntax

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

Parameters

None

Return Value

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp file.

See Also

GetErrorMessage

GetErrorMessage

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

Note:
THREAD SAFETY: GetErrorMessage() returns the error message of the last error that occurred within the current thread.

Syntax

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

Parameters

None

Return Value

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

Example

Sample code is available in the /examples/NNSYRF directory in the msgtest.cpp and apitest.cpp files.

See Also

GetErrorCode

Chapter 5

New Era of Networks 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

For information of naming components, see *Formatter Naming Conventions* on page 7.

WARNING!
New Era of Networks Formatter Management APIs are not thread-safe and should not be used in a multi-threaded environment.

General New Era of Networks Formatter Management APIs

NNFMgrInit

Allocates and returns a pointer to an instance of NNFMgr tied to the DBMS specified by session.

Syntax

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

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.

See Also

NNFMgrClose

OpenDbmsSession

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. 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

Use to clear structures when using New Era of Networks Formatter Management APIs prior to invoking each function. Clearing structures should be done with a call to the NNF_CLEAR() macro. NNF_CLEAR() clears a structure in such a way that the Format Management APIs can alert the user to a non-initialized structure.

Syntax

```
NNF_CLEAR(_p)
```

Parameters

Name	Type	Input/ Output	Description
_p	Any format management structure	Input	Pointer to any structure used during format management. See structure descriptions for details.

Example

```
struct NNFMgrFormatInfo f_info;

NNF_CLEAR(&f_info);
```

New Era of Networks Formatter Permission APIs

New Era of Networks Formatter permissions allow a layer of security to be added to New Era of Networks Formatter objects. This security defines the ability to read, update, and own objects. Objects that contain definable security include: flat and compound formats, input and output controls, output operations and collections, fields, literals, and user-defined types.

Permissions can be designated for Read, Update, and Owner. The user who creates an object is the default owner of the object. The owner has Update and Read access. The owner can grant and revoke Update permissions to other users. The owner can also grant Owner permission to another user. Because only one user can have Owner permission, granting another user Owner permission revokes the grantor's Owner permission. Successful Create APIs associate Owner and Update permissions based on session user. Update and Delete calls verify that the session user has the permission to perform the requested function.

By default, PUBLIC is given Read permission. Users with Read access are allowed to view an object and all of its attributes. PUBLIC cannot be granted Owner permission for an object and cannot be granted Update permission.

NNFie treats permissions associated with a New Era of Networks Formatter component as a separate entity. For more information, see *Upgrading Formats and Rules* in the ***System Management Guide***.

The Permission APIs require the use of the following structures:

```
typedef struct NNMgrCmpntData
{
   char cmpntName[NN_COMPONENT_VALUE_LEN]; //Component Name.
    int hierarchyId; //Hierarchy ID of cmpntName.
    long version; //Component version.
} NNMgrCmpntData;
```

```
typedef struct NNMgrPermData
{
   int dataLevels; //Number of NNMgrCmpntData objects in the
                   //NNMgrCmpntData data array.
   NNMgrCmpntData data[MX_LEVELS]. //Array containing
                                   //information on the
                                   //individuals levels that
                                   //make up one object.
} NNMgrPermData;
```

For more information on the Hierarchy IDs required by NNMgrPermData, see *Hierarchy IDs* on page 525

NNPMgrGetParticipantInfo

Retrieves permission information for a specified object (New Era of Networks Formatter component). The information to be retrieved depends on the selection parameter (selection = 1).

The available selections are:

1. If selection is equal to one (selection = 1), the function retrieves a list of all NN_PARTICIPANT participant names. This includes all people who currently have components assigned to them (with ownership or update privileges.)
2. If selection is equal to two (selection = 2), the function retrieves an object user list with the associated permissions on each user in the order of read, update, owner.
3. If selection is equal to three (selection = 3), the function returns the name of the logged-in participant and identifies any permissions (owner, update, owner and update, or no permissions) associated with the specified object. If the participant does not have any permission with the object, the function returns an empty string ("").

Syntax

```
const int *NNPMgrGetParticipantInfo(DbmsSession *pSession,
                    const NNMgrPermData *NNMgrPermData,
                    const char * pszBuffer,
                    int nBuffer,
                     const int selection = 1)
```

Parameters

Name	Type	Input / Output	Description
pSession *	const DbmsSession	Input	Provides database session information.
PermData*	NNMgrPermData*	Input	Define New Era of Networks Formatter component for which read, update, and owner privileges are sought.
pszBuffer	const char*	Output	Defines the pointer for the buffer to populate.
nBufferSize	int	Input	Defines the size of the buffer.
selection = 1	const int	Input	Defines the selection to use in determining retrieval information. Defaults to 1.

Return Value

Returns 1 to indicate a successful retrieval.
Returns 0 when no data is available.

Returns -1 when a SQL error occurs.
Returns -2 when allocation error occurs.
Returns -3 when a general error occurs.

NNPMgrGetGlobalUserList

Retrieves a list of all DBMS global system users based on the individual DBMS system tables. If there are no users listed, this function returns an empty string ("").

Syntax

```
const int *NNPMgrGetGlobalUserList(DbmsSession *pSession,
                                   const char * pszBuffer,
                                   int nBuffer);
```

Parameters

Name	Type	Input/ Output	Description
pSession*	const DbmsSession	Input	Provides database session information.
pszBuffer	const char	Output	Defines the pointer for the buffer to populate.
nBufferSiz	int	Output	Defines the size of the buffer.

Return Value

Returns 1 to indicate a successful list return.
Returns 0 when no list is available.

Returns -1 when a SQL error occurs.
Returns -2 when allocation error occurs.
Returns -3 when a general error occurs.

NNPMgrHasOwnerPermDefined

Retrieves the status of Owner permission for a specified participantId with the given grantId.

Syntax

```
const int NNPMgrHasOwnerPermDefined(DbmsSession *pSession,
                      const NNMgrPermData *NNmgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object to be checked for Owner permission.

Remarks

If the PUBLIC user has been granted Owner permission, then an individual user has Owner permission, even if the specified participantId is not marked as having Owner status.

Return Value

Returns 1 if the specified participantId has Owner permission for the given grantID; returns zero (0) if the specified participantId does not have Owner permission for the given grantId.

NNPMgrHasUpdatePermDefined

Retrieves the status of Update permission for a specified participantId with the given grantId.

Syntax

```
const int NNPMgrHasUpdatePermDefined(DbmsSession *pSession,
                        const NNMgrPermData *NNmgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object to be checked for Update permission.

Remarks

If the PUBLIC user has been granted Update permission, then an individual user has Update permission, even if the specified participantId is not marked as having Update status.

Return Value

Returns 1 if the specified participantId has Update permission for the given grantID; returns zero (0) if the specified participantId does not have Update permission for the given grantId.

NNPMgrHasOwnerPermAssociated

Retrieves the status of Owner permission for a specified participantId with the given grantId.

Syntax

```
const int NNPMgrHasOwnerPermAssociated (DbmsSession
                    *pSession,
                     const NNMgrPermData *NNmgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object to be checked for Owner permission.

Remarks

If the PUBLIC user has been granted Owner permission, then an individual user has Owner permission, even if the specified participantId is not marked as having Owner status.

Return Value

Returns 1 if the given participant has Owner permission for the given object, or 0 if the given participant does not have Owner permission for the given object.

NNPMgrHasUpdatePermAssociated

Retrieves the status of Update permission for a specified participantId with the given grantId.

Syntax

```
const int NNPMgrHasUpdatePermAssociated(DbmsSession *pSession,
                          const NNMgrPermData *NNmgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object to be checked for Update permission.

Remarks

If the PUBLIC user has been granted Update permission, then an individual user has Update permission, even if the specified participantId is not marked as having Update status.

Return Value

Returns 1 if the specified participantId has Update permission for the given grantID; returns zero (0) if the specified participantId does not have Update permission for the given grantId.

NNPMgrDeleteObject

Deletes an object associated with a component.

Syntax

```
const int NNPMgrDeleteObject(const DbmsSession *pSession,
                             const NNMgrPermData *NNMgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object to be deleted.

Note:
NNPMgrDeleteObject() allows deletion of the permissions on a component. Extra caution should be taken when using this API, since deleting a component's permissions will make deleting the component itself impossible. In order to restore permissions on a component, use the API NNPMgrSetNewObjectPermissions().

NNPMgrSetOwnership

Grants Owner permission for a specified participantId with a given grantId.

Syntax

```
const int NNPMgrSetOwnership(const DbmsSession *pSession,
                             const NNMgrPermData *NNMgrPermData,
                             const char *newOwnerName);
participantId)
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	const NNMgrPermData	Input	Object containing values for setting Owner permission.
newOwnername	const char	Input	Name to be granted Ownership with the specified object.

NNPMgrSetUpdatePermission

Sets or revokes Update permission for a specified participantId with a given grantId.

Syntax

```
const int NNPMgrSetUpdatePermission(const DbmsSession
                                    *pSession,
                                    const NNMgrPermData*NNMgrPermData,
                                    const  char *nameForUpdate);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	NNMgrPermData*	Input	Object containing values for setting update permission.
nameForUpdate	const char*	Input	Name to be given update permission with the specific object.

NNPMgrSetNewObjectPermissions

Sets permissions for the specified object.

Syntax

```
const int NNPMgrSetNewObjectPermissions(const DbmsSession
                            *pSession,
                            const NNMgrPermData *NNMgrPermData);
```

Parameters

Name	Type	Input/ Output	Description
*pSession	const DbmsSession	Input	Provides database session information.
NNMgrPermData	NNMgrPermData*	Input	Permissions object to be added to the database.

Remarks

Read, Update, and Owner permission can be granted for the specified participantId. Use NNMgrGetParticipantId() to retrieve the participantId. If the object already exists, it is not added again.

Field Management APIs

Field Management API Structure

NNFMgrFieldInfo

NNFMgrFieldInfo is a structure containing field information.

Syntax

```
typedef struct NNFMgrFieldInfo {
        unsigned char fieldName[NAME_LENGTH+1];
        unsigned char fieldDescription[DESCRIPTION_LENGTH+1];

        long initFlag;
};
```

Parameters

Name	Type	Description
fieldName	unsigned char[ ]	Name of field to add. NULL-terminated string of length 1 to 120 inclusive.
fieldDescription	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

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	Valid FMgr session previously returned by NNFMgrInit().
pFieldInfo	const NNFMgrField Info * 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

NNFMgrFieldInfo

NNFMgrGetFirstField

NNFMgrGetNextField

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,
            NNFMgrFieldInfo * pFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFieldInfo	NNFMgrFieldInfo * 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 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

NNFMgrFieldInfo

NNFMgrCreateField

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,
            NNFMgrFieldInfo * pFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFieldInfo	NNFMgrFieldInfo * 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 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

NNFMgrFieldInfo

NNFMgrCreateField

NNFMgrGetFirstField

NNFMgrUpdateField

Updates an existing field in the database. Before calling this function, the NNFMgrFieldInfo data structure must be initialized with the new values. The NNFMgrGetFirstField() and NNFMgrGetNextField() functions can be used to populate this data structure with the current values. The fieldName 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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

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	NNFMgrFieldCntlInfo* const	Input	Pointer to a valid NNFMgrFieldCntlInfo 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 GetErrorNo() 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

NNFMgrFieldInfo

NNFMgrCreateField

NNFMgrGetFirstField

NNFMgrUpdateParseControl

NNFMgrUpdateOutMstrCntl

NNFMgrDeleteField

Deletes a single field from the database.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

Syntax

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

Parameters

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

Remarks

This function performs no referential integrity checks on the database. If the deleted field is used in one or more formats, those formats will no longer function properly. The permissions for the field will be deleted as well.

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

NNFMgrDeleteParseControl

NNFMgrDeleteOutMstrCntl

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.

When a user specifies a literal name, New Era of Networks 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 NNFMgrCreatexxxCntl. 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[NAME_LENGTH+1];
        unsigned char literalValue[127];
        unsigned short literalLength;
        unsigned short dataTypeID;
        long initFlag;
};
```

Parameters

Name	Type	Description
literalName	unsigned char[ ]	Name of literal to create. NULL-terminated string length 1 to 120 inclusive.
literalValue	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	long	Uninitialized structure check value.
dataTypeID	unsigned short	Data type of literal. For binary data, the dataTypeId should be set to 0 and for string data, the dataTypeId should be 1.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrLiteral Info* const	Input/ Output	Pointer to structure which will provide data about NNFMgrCreateLiteral.

Return Value

Returns 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 215.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrLiteral Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetLiteral.

Return Value

Returns 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

NNFMgrUpdateLiteral

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

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

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.
pInfo	NNFMgrLiteral Info * const	Input	Pointer to a valid NNFMgrLiteralInfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted literal is used in other controls or formats, 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.

See Also

NNFMgrCreateLiteral

NNFMgrGetLiteral

NNFMgrUpdateLiteral

User-Defined Data Type Management APIs

User-defined data types can only be assigned 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.

User-Defined Data Type Management API Structures

NNFMgrUserDefTypeInfo

NNFMgrUserDefTypeInfo is a structure containing user-defined type information.

Syntax

```
typedef struct NNFMgrUserDefTypeInfo {
char userDefTypeName[NAME_LENGTH+1];
char nativeTypeName[33];
char validationRoutineName[33];
long initFlag;
} NNFMgrUserDefTypeInfo;
```

Parameters

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

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[CODE_TYPE_LENGTH+1];
               NameValuePair* pairs;
               long initFlag;
} NNFMgrNameValuePairInfo;
```

Parameters

Name	Type	Description
objectName	char[ ]	Name of object associated with this structure's name/value pair array.
pairType	char[ ]	Type of 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	long	Uninitialized structure check value.

User-Defined Data Type Management APIs

For information on user-defined type validation, see *User Callback API Structure* on page 109.

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

Remarks

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

Return Value

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

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

See Also

NNFMgrGetUserDefinedType

NNFMgrGetFirstUserDefinedType

NNFMgrGetNextUserDefinedType

NNFMgrDeleteUserDefinedType

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

Remarks

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

Return Value

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

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

NNFMgrGetUserDefinedType

Retrieves user-defined type information for the user-defined type named in pTypeName.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	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	NNFMgrUserDefType Info * 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

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,
            NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

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

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,
            NNFMgrUserDefTypeInfo * const pTypeInfo);
```

Parameters

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

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

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

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.
pInfo	NNFMgrUser DefTypeInfo * const	Input	Pointer to a valid NNFMgrUserDefTypeInfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted, user-defined type is used in other controls or formats, 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.

See Also

NNFMgrGetUserDefinedType

NNFMgrGetFirstUserDefinedType

NNFMgrGetNextUserDefinedType

Parse Control Management APIs

Parse Control Management API Structures

NNFMgrParseControlInfo

NNFMgrParseControlInfo is a structure containing parse control information.

Syntax

```
typedef struct NNFMgrParseControlInfo {
    char parseName[NAME_LENGTH+1];
    unsigned char optionalInd;
    short fieldType;-- not 8 anymore, now would be
    appropriate type - Data only, etc.

    /* parse data information */
    short dataType;-- can be RE, RE + whitespace or RE +
    delimiter
    short dataTermination;
    char dataRegExp[NAME_LENGTH+1];-- contains the data RE
    char dataDelimiter[NAME_LENGTH+1];
    unsigned dataLength;
    short dataLengthUnit;

    /* parse tag information */
    short tagType;                          // tag data type
    can be RE, RE + whitespace or RE + delimiter
    short tagTermination;
    unsigned tagLength;                     // length of tag
    value
    char tagLitrlName[NAME_LENGTH+1];       // literal name of
    tag
    unsigned char tagValue[LITRL_LENGTH+1]; // tag value that
    appears on input-- contains the tag RE
```

```
    char tagDelimiter[NAME_LENGTH+1];       // literal name of
    delimiter for tag
    short tagLengthUnit;

    /* length information */
    short lengthType;
    short lengthTermination;
    unsigned lengthLength;
    char lengthDelimiter[NAME_LENGTH+1];
    short lengthLengthUnit;

    /* Decimal Location information */
    unsigned short decimalLocation;

    // aaron: ser1079 2/97
    // user defined type validation information
    char           validationParamName  [NAME_LENGTH+1];
    NameValuePair* userDefInValNameValuePairArray;

  // Date/Time data type fields

    char dataAttr[NAME_LENGTH+1];
    short baseDataType;
    short yearCutoff; // For two digit to four digit year
    conversion
    short useZeroYearCutoffInd; // 1 allows user to specify 0
    yearCutoff value

    long initFlag;

} NNFMgrParseControlInfo;
```

Parameters

Name	Type	Description
parseName	char[ ]	Name of parse control. NULL-terminated string of length 1 to 120 inclusive.
optionalInd	unsigned char	Zero (0) if the control is mandatory, non-zero if optional.

Name	Type	Description
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 data type code defined using user-defined type input field validation can also be used. For more information, see *Data Types* on page 463
dataRegExp	char[ ]	The expression for the data if fieldType is INFIELD_PARSE_Regex.
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 TERMINATION_RegExp TERMINATION_RegExp_Delimited TERMINATION_RegExp_White_Space_ Delimited
dataDelimiter	char[ ]	Name of delimiter if dataTermination is TERMINATION_Delimiter or TERMINATION_Minimum_Length_ Delimiter.

Name	Type	Description
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
dataLengthUnit	short	Distinguishes between character-oriented or byte-oriented processing. Character data should be set to 0 and byte data should be set to 1.
tagType	short	Must be a supported data type.
tagTermination	short	Same possible values as dataTermination.
tagLiteralName	char[ ]	Literal name of tag.
tagValue	unsigned char[ ]	Tag value, literal value, or regular expression.
tagDelimiter	char[ ]	Name of tagTermination delimiter: TERMINATION_Delimiter TERMINATION_Minimum_Length_ Delimiter
tagLengthUnit	short	Distinguishes between character-oriented or byte-oriented processing. Character data should be set to 0 and byte data should be set to 1.
lengthType	short	Must be a supported data type.
lengthTermination	short	Same possible values as dataTermination.

Name	Type	Description
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	char[ ]	Name of delimiter if lengthTermination is one of: TERMINATION_Delimiter TERMINATION_Minimum_Length_ Delimiter.
lengthLengthUnit	short	Distinguishes between character-oriented or byte-oriented processing. Character data should be set to 0 and byte data should be set to 1.
decimalLocation	unsigned short	Number indicating the decimal point location. Zero (0) indicates no decimal point change. Positive numbers indicate the number of digits in the input field considered to be right of the decimal point. Must not be longer than the maximum number of digits the field can hold. Negative numbers are not allowed.
validationParam Name	char[ ]	String value that user-defined type input field validation code pass to the Callback RuntimeDataLookup function.
userDefInValName ValuePairArray	NameVal uePair*	An array of name/value pairs associated with the parse control user-defined type input field validation operations. The values in this array will be passed to a validation callback function.
initFlag	long	Uninitialized structure check value.

Name	Type	Description
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.
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 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 the 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.

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

Year 2000 Compliance

The yearCutoff field is used to indicate a cutoff year when dealing with two-digit years on input. By default, yearCutoff is always set to 0 in the call to NNF_CLEAR. The following logic controls the century assigned to two-digit years.

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

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

To specify a yearCutoff of 0, the user must specify yearCutoff = 0, and specify useZeroYearCutoffInd = 1. The useZeroYearCutoffInd field indicates the user intentionally set yearCutoff to 0.

See Also

For more information, see Appendix D*: Data Types* on page 463.

Parse Control Management APIs

NNFMgrCreateParseControl

Adds a new parse control to the database.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid FMgr session previously returned by NNFMgrInit().
pParseControl Info	const NNFMgrParse ControlInfo *	Input	Information about the parse control to add.

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 one of the following types, delimiter information is ignored:

DATA_TYPE_IBM_Packed_Integer,
DATA_TYPE_IBM_Signed_Packed_Integer,
DATA_TYPE_IBM_Zoned_Integer, or
DATA_TYPE_IBM_Signed_Zoned_Integer,

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. To use custom date / time formats, set dataAttr to the 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

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseName	char *	Input	Name of parse control. NULL-terminated string of length 1 to 32 inclusive.
pParseControl Info	NNFMgrParse ControlInfo * 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 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

NNFMgrGetFirstParseControl

NNFMgrGetNextParseControl

NNFMgrUpdateParseControl

NNFMgrDeleteParseControl

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseControl Info	NNFMgrParse ControlInfo * 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 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

NNFMgrGetNextParseControl

NNFMgrUpdateParseControl

NNFMgrDeleteParseControl

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pParseControl Info	NNFMgrParse ControlInfo * 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 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

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

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	NNFMgrParse ControlInfo * 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other input formats, 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.

See Also

NNFMgrParseControlInfo

NNFMgrCreateParseControl

NNFMgrGetParseControl

NNFMgrGetFirstParseControl

NNFMgrGetNextParseControl

NNFMgrUpdateParseControl

NNFMgrDeleteParseControl

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:

- NNFMgrUserExitCntlInfo
- NNFMgrMathExpCntlInfo
- NNFMgrMathExpCntlSegmentInfo
- NNFMgrPrePostFixCntlInfo
- NNFMgrDefaultCntlInfo
- NNFMgrLengthCntlInfo
- NNFMgrSubStringCntlInfo
- NNFMgrCaseCntlInfo
- NNFMgrDataTypeCntlInfo
- NNFMgrJustifyCntlInfo
- NNFMgrTrimCntlInfo
- NNFMgrCollectionCntlInfo
- NNFMgrGetOutMstrCntlInfo
- NNFMgrUpdateCntlInfo
- NNFMgrDeleteCntlInfo

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 populated 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 GET_FIRST argument.

Getting a Specific Control

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

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

2. Type the name of the control 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 NNFMgrGetxxxCntl 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.

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;
        short optionalInd;
        short dataType;
        char dataAttr[NAME_LENGTH+1];
        short baseDataType;
        short tagType;
        char tagLitrlName[NAME_LENGTH+1]
        unsigned char tagValue[LITRL_LENGTH+1];
        unsigned short tagValueLen;
        short tagBeforeLengthInd;
        short lengthType;
        short operationType;
        char fldLitrlName[NAME_LENGTH+1];
        unsigned char fldValue[LITRL_LENGTH+1];
        unsigned short fldValueLen;
        char childCntlName[NAME_LENGTH+1];
        NNCntlType childCntlType;
        long initFlag;
} NNFMgrOutMstrCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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.
dataAttr	char[ ]	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.

Name	Type	Description
baseDataType	short	baseDataType 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.
tagLitrlName	short	The name of the literal to use for output tag. Used only for field type of OUTFIELD_FORMAT_Data_Field_Tag_ Search.
tagValue	unsigned char[ ]	The literal value to use for output tag. Used only for field type of OUTFIELD_FORMAT_Data_Field_Tag_ Search. If tagLitrlName is specified, this field is ignored.
tagBeforeLengthInd	short	Used only for field with both tag and length information. Indicates whether the tag is to be written to the output field before length. If tagBeforeLengthInd = 1, tag is written before length. Otherwise, length is written before tag.
lengthType	short	Used only for field with both tag and length information. Indicates the data type of the length portion of this field. Must be one of the valid data types.

Name	Type	Description
operationType	short	Used only for fieldType = OUTFIELD_FORMAT_Calculated_Field. Valid values are: OPERATION_Add OPERATION_Subtract OPERATION_Multiply OPERATION_Divide
fldLitrlName	char[ ]	Used for fieldType = and Literal output control types. OUTFIELD_FORMAT_Input_Field_Equals. Used to specify the name of a literal to use as the comparison value.
fldValue	char[ ]	Used only for fieldType = OUTFIELD_FORMAT_Input_Field_Equals. Used to specify the comparison value. If fldLitrlName is specified, this field is ignored.
childCntlName	char[ ]	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. The output master control can refer to a single output control that can be a single operation, such as Trim, or a collection of controls. The child control determines the additional functionality of the output master control.

Name	Type	Description
childCntlType	NNCntl Type	Indicates the type of the child control named by childCntlName. The child control can be any valid control type. To specify no child control is to be used with this output master control, set childCntlType to NO_CNTL. The valid values for the ChildCntlType are as follows: NO_CNTL = 0 SUBSTITUTE_CNTL = 1 PRE_POST_FIX_CNTL= 2 DEFAULT_CNTL= 3 LENGTH_CNTL = 4 SUBSTRING_CNTL= 5 CASE_CNTL= 6 USER_EXIT_CNTL= 7 MATH_EXP_CNTL= 8 JUSTIFY_CNTL= 10 COLLECTION_CNTL= 11 TRIM_CNTL= 12
initFlag	long	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[NAME_LENGTH+1];
        char inputLitrlName[NAME_LENGTH+1];
        unsigned char inputValue[128];
        unsigned short inputValueLen;
        char outputLitrlName[NAME_LENGTH+1];
        unsigned char outputValue[128];
        unsigned short outputValueLen;
        short outputValueType; // Data type of outputvalue
        long initFlag;
} NNFMgrSubstituteCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
inputLitrlName	char[ ]	Name of the input literal to use for the input value of this substitute.
inputValue	char[ ]	The value to use as the input value of this substitute. If inputLitrlName is specified, this field is ignored.
inputValueLen	unsigned short	The length (in bytes) of the valid data stored in inputValue.

Name	Type	Description
outputLitrlName	char[ ]	Name of the literal to use for the output value of this substitute.
outputValue	char[ ]	The value to use as the output value of this substitute. If outputLitrlName is specified, this field is ignored for a value not Litrl.
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 New Era of Networks Formatter data type.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrUserExitCntlInfo

Houses information used to create a newUser Exit control or gets an existing user exit control.

Syntax

```
typedef struct NNFMgrUserExitCntlInfo {
        char cntlName[NAME_LENGTH+1];
        char exitRoutine[33];
        long initFlag;
} NNFMgrUserExitCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
exitRoutine	char[ ]	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[NAME_LENGTH+1];
        unsigned short decimalPrecision;
        unsigned short roundingMode;
        long initFlag;
} NNFMgrMathExpCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
decimalPrecision	unsigned short	The decimal precision to carry with this math expression. The maximum value is 11.
roundingMode	unsigned short	Provides options to round the math expression up or down. 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
expression	char[ ]	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[NAME_LENGTH+1];
        char litrlName[NAME_LENGTH+1];
        unsigned char value[128];
        unsigned short valueLen;
        NNFPrePostFix place;
        short nullActionInd;
        long initFlag;
} NNFMgrPrePostFixCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
litrlName	char[ ]	The name of the literal which contains the data to be added to the output field is a prefix or postfix (suffix). If no name is entered, a new literal component is created based on the trimChar and trimCharLen.
value	unsigned char[ ]	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 if a literal is not provided.

Name	Type	Description
place	NNFPrePostFix	PREFIX or POSTFIX
nullActionInd	short	Flags this control to be used in the case of NULL input field data. Set this field to 1 to activate it for NULL action.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrDefaultCntlInfo

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

Syntax

```
typedef struct NNFMgrDefaultCntlInfo {
        char cntlName[NAME_LENGTH+1];
        char litrlName[NAME_LENGTH+1];
        unsigned char value[128];
        unsigned short valueLen;
        long initFlag;
} NNFMgrDefaultCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
litrlName	char[ ]	Name of the literal that contains the data for this default. If no name is entered, a new literal component is created based on the trimChar and trimCharLen.
value	unsigned char[ ]	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 if a literal is not provided.
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[NAME_LENGTH+1];
        char padLitrlName[NAME_LENGTH+1];
        unsigned char padValue[128];
        unsigned short padValueLen;
        unsigned long dataLen;
        short dataLenUnit;
        long initFlag;
} NNFMgrLengthCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
padLitrlName	char[ ]	The name of the literal that contains the data to be used to pad to the length specified in dataLen. If no name is entered, a new literal component is created based on the trimChar and trimCharLen.
padValue	unsigned char[ ]	The value to be used to pad to the length specified in dataLen. If padLitrlName is specified, this field is ignored.
padValueLen	unsigned short	The length (in bytes) of the valid data in padValue if a literal is not provided.
dataLen	unsigned long	The length for this length control.

Name	Type	Description
dataLenUnit	short	Distinguishes between character-oriented or byte-oriented processing.
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[NAME_LENGTH+1];
        unsigned short start, len;
        char padLitrlName[NAME_LENGTH+1];
        unsigned char padValue[128];
        unsigned short padValueLen;
        short substringUnit;
        long initFlag;
} NNFMgrSubStringCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by 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. 0 = go to the end of the string
padLitrlName	char	The name of the literal that contains the data to be used to pad to the length specified in len. If no name is entered, a new literal component is created based on the trimChar and trimCharLen.

Name	Type	Description
padValue	unsigned char[ ]	The value used to pad to the length specified in len. If padLitrlName is specified, this field is ignored.
padValueLen	unsigned short	The length (inbytes) of the valid data in padValue if a literal is not provided.
substringUnit	short	Distinguishes between character-oriented or byte-oriented processing.
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[NAME_LENGTH+1];
        NNFCase caseId;
        long initFlag;
} NNFMgrCaseCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by 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.

NNFMgrDataTypeCntlInfo

Houses information used to get an existing data type control. Users never create data type controls.

Syntax

```
typedef struct NNFMgrDataTypeCntlInfo {
        char cntlName[NAME_LENGTH+1];
        long initFlag;
} NNFMgrDataTypeCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
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[NAME_LENGTH+1];
        NNFJustify justify;
        long initFlag;
} NNFMgrJustifyCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by 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

Houses information used to create a trim control.

Syntax

```
NNFMgrTrimCntlInfo {
      char cntlName[NAME_LENGTH+1];
      char trimCharLitrlName[NAME_LENGTH+1];
      unsigned char trimChar[LITRL_LENGTH+1];
      unsigned short trimCharLen;
      NNFTrim trim;
      long initFlag;
} NNFMgrTrimCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char[ ]	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. The cntlName field is populated with the default name returned by the API call.
trimCharLitrlName	char[ ]	The name of the Literal New Era of Networks 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[ ]	The value of the trim pad character; same value as the referenced literal.
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.

Name	Type	Description
trim	NNtrim	This value is an enumerated type identifying which side of the field to trim: Left_TRIM, Right_TRIM, and BOTH_TRIM.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

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[NAME_LENGTH+1];
        long initFlag;
} NNFMgrCollectionCntlInfo;
```

Parameters

Name	Type	Description
cntlName	char	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. The cntlName field is populated with the default name returned by the API call.
initFlag	long	Set by NNF_CLEAR to indicate a properly initialized structure. Do not modify this value.

NNFMgrCntlInfo

Houses information used to create a new control or get an existing control from a collection.

Syntax

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

Parameters

Name	Type	Description
cntlName	char	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. The cntlName field is populated with the default name returned by the API call.
cntlType	NNCntlType	The type of the control. Valid types are: NO_CNTL SUBSTITUTE_CNTL PRE_POST_FIX_CNTL DEFAULT_CNTL SUBSTRING_CNTL CASE_CNTL USER_EXIT_CNTL MATH_EXP_CNTL JUSTIFY_CNTL COLLECTION_CNTL TRIM_CNTL DATATYPE_CNTL
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 the opcode argument in the API call.

NNFMgrCreateOutMstrCntl

Creates a new output master control and associates it to a single child control that determines the additional functionality of the output master 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	NNFMgrOut MstrCntlInfo*	Input/ Output	Pointer to structure that provides data about OutMstrCntl.

Remarks

You can specify the literal name to use for tag by populating the tagLitrlName field of the pInfo structure, and specify field value (used with Input Value

Equals and Literal field types) by populating the fldLitrlName 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

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.

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	Pointer to structure that contains data about NNGetOpCode.
pInfo	NNFMgrOutMstr CntlInfo	Input/ Output	Pointer to structure that contains data about NNFMgrGetOutMstrCntl.

Return Value

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

See Also

NNFMgrCreateOutMstrCntl

NNFMgrUpdateOutMstrCntl

OpCode

NNFMgrUpdateOutMstrCntl

Updates an existing Output Control in the database. Before calling this function, the NNFMgrOutMstrCntlInfo data structure must be initialized with the new values. The API function NNFmgrGetOutMstrCntl 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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

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	NNFMgrOutMstrCntlInfo * const	Input	Pointer to a valid NNFMgrOutMstrputCntl Info 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 NNFMgrDeleteOutMstrCntl API call; then calls NNFMgrCreateOutMstrCntl to create a new output control 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);
NNFMgrGetOutMstrCntl(pNNFMgr, "StringWithNewline", &Info );
strcpy(Info.controlName,"StringWithSemicolon");// change name
strcpy(Info.suffix, "Semicolon");// change suffix literal
NNFMgrUpdateOutMstrCntl(pNNFMgr, "StringWithNewline", &Info);
```

See Also

NNFMgrCreateOutMstrCntl

NNFMgrGetOutMstrCntl

NNFMgrOutMstrCntlInfo

NNFMgrDeleteOutMstrCntl

Deletes a single output master control from the database.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control 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.

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 ***User Guide***.

This section details the following output operation control APIs:

- Substitute
- User Exit
- Math Expression
- PrePostFix
- Default
- Length
- SubString
- Case
- Data type
- Justify
- Trim
- Collection

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr SubstituteCntl Info*	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

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.

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 NNFMgrCreateSubstitureCntl() with the same cntlName before making this call. An error is returned if no control exists by this name.

Syntax

```
const short NNFMgrAppendEntryToSubstituteControl(
            NNFMgr* pNNFMgr,
            const NNFMgrSubstituteCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr SubstituteCntl Info*	Input	Pointer to structure that provides data about NNFMgrCreateSubstituteCntl.

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.

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 215.

Syntax

```
const short NNFMgrGetSubstituteCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrSubstituteCntlInfo* const pInfo,
            int* const NumRemainingEntries)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	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

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.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr SubstituteCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetNextEntryFrom SubstituteCntl.

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

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.

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateSubstituteCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrSubstituteCntlInfo* 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 NNFMgrSubstituteCntlInfo structure.

Remarks

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

NNFmgrUpdateSubstituteCntl() maintains all references from parent components even if the control's name is changed.

NNFmgrUpdateSubstituteCntl() first truncates the list of substitution strings 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 (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

```
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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control 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.

See Also

NNFMgrSubstituteCntlInfo

NNFMgrCreateSubstituteCntl

NNFMgrAppendEntryToSubstituteControl

NNFMgrGetSubstituteCntl

NNFMgrUpdateSubstituteCntl

User Exit Controls

User exits compute the value of an output field. The user can write a C function to perform a task outside what New Era of Networks Formatter can currently do. See the ***System Management*** guide for details.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrUser ExitCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateUserExitCntl.

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.

See Also

NNFMgrUserExitCntlInfo

NNFMgrGetUserExitCntl

NNFMgrGetUserExitCntl

Gets a single User Exit control from the database. The location of the returned control within the database is determined by the OpCode argument. See *OpCode* on page 215.

Syntax

```
const short NNFMgrGetUserExitCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrUserExitCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrUser ExitCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetUserExitCntl.

Return Value

Returns 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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateUserExitCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrUserExitCntlInfo* 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	NNFMgrUser ExitCntlInfo* const	Input	Pointer to a valid NNFMgrUserExitCntlInfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrUserExitCntlInfo

NNFMgrCreateUserExitCntl

NNFMgrGetUserExitCntl

NNFMgrUpdateUserExitCntl

Math Expression Controls

Math expression controls can contain any length of math expression 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 using the standard Create and Get APIs. For detailed information on math expressions, see the New Era of Networks Formatter chapter of the ***User's Guide***.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrMath ExpCntlInfo* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateMathExpCntl.

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.

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 215.

Syntax

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

Parameters

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

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.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
CntlName	const char* const	Input	The name of the control
pInfo	const NNFMgrMath ExpCntl SegmentInfo* const	Input	Pointer to structure that contains data about NNFMgrGetSegFrom MathExpCntl.

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.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
CntlName	const char* const	Input	The name of the control.
OpCode	NNGetOp	Input	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 NNFMgrGetSegFrom MathExpCntl.

Remarks

The position of the returned segment is determined by the OpCode argument. See *OpCode* on page 215.

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.

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateMathExpCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrMathExpCntlInfo* 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	NNFMgrMath ExpCntlInfo* const	Input	Pointer to a valid NNFMgrMathExpCntlnfo structure.

Remarks

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

NNFMgrUpdateMathExpCntl() function maintains all references from parent components even if the control's name is changed.

This API function truncates the list of math expressions 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrMathExpCntlSegmentInfo

NNFMgrCreateMathExpCntl

NNFMgrGetMathExpCntl

NNFMgrAppendSegToMathExpCntl

NNFMgrUpdateMathExpCntl

PrePostFix Controls

PrePostFix controls are used to add user-defined information to the front (prefix) 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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgrPre PostFixCntl Info* const	Input/ Output	Pointer to structure that provides data about NNFMgrCreatePrePostFixCntl.

Return Value

Returns 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 215.

Syntax

```
const short NNFMgrGetPrePostFixCntl (
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrPrePostFixCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument.
pInfo	NNFMgrPre PostFixCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetPrePostFixCntl.

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.

See Also

NNFMgrPrePostFixCntlInfo

NNFMgrCreatePrePostFixCntl

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdatePrePostFixCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrPrePostFixCntlInfo* 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	NNFMgrPrePost FixCntlInfo* const	Input	Pointer to a valid NNFMgrPrePostFixCntlInfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrPrePostFixCntlInfo

NNFMgrCreatePrePostFixCntl

NNFMgrGetPrePostFixCntl

NNFMgrUpdatePrePostFixCntl

Default Controls

Default output controls provide a default value for an output field if an input field does not exist in the input message or the input field has a length of zero (0).

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	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. You can also specify a literal name to use for the default by populating the litrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control.

Return Value

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.

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 215.

Syntax

```
const short NNFMgrGetDefaultCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrDefaultCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The position of the returned segment is determined by the OpCode argument.
pInfo	NNFMgr DefaultCntl Info* const	Input/ Output	Pointer to structure that provides data for NNFMgrGetDefaultCntl.

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 the error number.

See Also

NNFMgrDefaultCntlInfo

NNFMgrCreateDefaultCntl

NNFMgrUpdateDefaultCntl

NNFMgrDeleteDefaultCntl

NNFMgrGetDefaultCntlName

Creates a new control name baed on the control type and the number of those controls that exist.

Syntax

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

Parameters

Name	Type	Input/ Output	Description
Type	NNCntlType	Input	Indicates the type of control. The valid values for Type are described in the following table.
CntlName	char*	Input/ Output	Name of the control. The user must allocate NAMELENGTH characters for this parameter before calling this function.

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

Type	Root Value	Reusable control
SUBSTRING_CNTL	SubString	Yes
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 NNFMgrGetDefaultCntlName(). The generated name is stored in the cntlName field of the structure. 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 NNFMgrGetDefaultCntlName() 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 NNFMgrGetDefaultCntlName() and NNFMgrCreatexxxCntl() calls. This locks out other transactions and prevents duplicate key errors.

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

```
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);
```

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateDefaultCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrDefaultCntlInfo* 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	NNFMgrDefault CntlInfo* const	Input	Pointer to a valid NNFMgrDefaultCntlInfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrDefaultCntlInfo

NNFMgrCreateDefaultCntl

NNFMgrGetDefaultCntl

NNFMgrUpdateDefaultCntl

Length Controls

Length controls ensure that an output string is given a length. If the data length is longer than the specified length, the data is truncated. If the data length is shorter than the specified length, pad characters are used. Non-numeric data types are padded on the right; numeric data types are padded on the left. The Length and Justify controls are related.

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	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. You can also specify a literal name to use for the pad character by populating the padLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control. Only the first byte of the literal is used for padding.

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.

See Also

NNFMgrLengthCntlInfo

NNFMgrGetLengthCntl

NNFMgrUpdateLengthCntl

NNFMgrDeleteLengthCntl

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	NNFMgr LengthCntl Info* 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 215.

Return Value

Returns 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

NNFMgrCreateLengthCntl

NNFMgrUpdateLengthCntl

NNFMgrDeleteLengthCntl

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateLengthCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrLengthCntlInfo* 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	NNFMgrLength CntlInfo* const	Input	Pointer to a valid NNFMgrLengthCntlInfo 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

NNFMgrCreateLengthCntl

NNFMgrGetLengthCntl

NNFMgrDeleteLengthCntl

NNFMgrDeleteLengthCntl

Deletes a single length control from the database.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrLengthCntlInfo

NNFMgrCreateLengthCntl

NNFMgrGetLengthCntl

NNFMgrUpdateLengthCntl

SubString Controls

Substring controls allow you to extract a portion of an input string. The extraction is defined by start character position and length that is placed in the output field. A length of zero (0) means to extract until the end of the input field.

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	NNFMgrSub StringCntlInfo * const	Input/ Output	Pointer to structure that provides data about NNFMgrCreateSubStringCntl.

Remarks

You can specify a literal value to use for the pad character by populating the padValue field of the pInfo structure. You can also specify a literal name to use for the pad character by populating the padLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control. Note that only the first byte of the literal is used for padding.

Return Value

Returns 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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrSub StringCntlInfo * 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 215.

Return Value

Returns 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

NNFMgrDeleteSubStringCntl

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateSubStringCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrSubStringCntlInfo* 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	NNFMgrSub StringCntlInfo* const	Input	Pointer to a valid NNFMgrSubStringCntlInfo 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

NNFMgrGetSubStringCntl

NNFMgrDeleteSubStringCntl

NNFMgrDeleteSubStringCntl

Deletes a single substring control from the database.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrSubStringCntlInfo

NNFMgrCreateSubStringCntl

NNFMgrGetSubStringCntl

NNFMgrUpdateSubStringCntl

Case Controls

Case controls affect the case of the field data for string data (ASCII). The defined case controls are:

- LOWER_CASE
- UPPER_CASE

You cannot create, update, or delete Case controls.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrCase CntlInfo* 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 215.

Return Value

Returns 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

Data Type Controls

Data Type controls allow data type conversions to occur at a specified point in a sequence of output controls. The defined Data Type control is the CONVERT control. This control is used in an output collection when you want a conversion to take place in a sequence of output controls. If CONVERT is not used, conversion occurs at the beginning of the sequence.

You cannot create, update, or delete data type controls.

NNFMgrGetDataTypeCntl

Gets a single data type control from the database.

Syntax

```
const short NNFMgrGetDatTypeCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrDataTypeCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
OpCode	NNGetOp	Input	The location of the returned control is determined by the OpCode argument.
pInfo	NNFMgrData TypeCntlInfo* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetDataTypeCntl.

Remarks

The location of the returned control within the list of all data type controls is determined by the OpCode argument. The OpCode argument is an enumerated type. See *OpCode* on page 215.

Return Value

Returns 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

NNFMgrDataTypeCntlInfo

Justify Controls

Justify controls justifies field data within the length of the field. The defined controls are:

- CENTER_JUSTIFY
- LEFT_JUSTIFY
- RIGHT_JUSTIFY

You cannot create, update, or delete Justify controls.

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 215.

Syntax

```
const short NNFMgrGetJustifyCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrJustifyCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	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 JustifyCntl Info* const	Input/ Output	Pointer to structure that contains data about NNFMgrGetJustifyCntl.

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.

See Also

NNFMgrJustifyCntlInfo

Trim Controls

Trim controls remove a defined trim character at the right or left of the output data.

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
pInfo	NNFMgr TrimCntlInfo*	Input/ Output	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. You can also specify a literal name to use for the trim character by populating the trimCharLitrlName field of the pInfo structure. If you specify both a value and name, the name takes precedence. If a literal name is specified, the named literal must exist in the database before creating this control. Note that only the first byte of the literal is used to designate the trim character.

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.

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 215.

Syntax

```
const short NNFMgrGetTrimCntl(
            NNFMgr* pNNFMgr,
            NNGetOp OpCode,
            NNFMgrTrimCntlInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	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

Returns 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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateTrimCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrTrimCntlInfo* 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	NNFMgrTrim CntlInfo* const	Input	Pointer to a valid NNFMgrTrimCntlnfo 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted 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.

See Also

NNFMgrTrimCntlInfo

NNFMgrCreateTrimCntl

NNFMgrGetTrimCntl

NNFMgrUpdateTrimCntl

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

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

Returns 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

To retrieve child controls associated with this collection, use NNFMgrGetCntlFromCollection().

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 215.

Return Value

Returns 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	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
CollName	const char* const	Input	The name of the collection to which the control will be added.
SeqNum	int	Input	Indicates the position where the control is added starting at 1.
pInfo	const NNFMgrCntl Info* const	Input	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.

There is no NNFMgrRemoveCntlFromCollection function. To remove a control from a collection, do the following:

1. Use NNRMgrGetCntlFromCollection to walk through entire list of controls for the collection and save them.
2. Call NNFMgrUpdateColletionCntl to remove all controls from the collection.
3. Add each control back to the collection using NNFMgrAddCntlToCollection.

Return Value

Returns 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	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 215.

Return Value

Returns 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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateCollectionCntl(
            NNFMgr *pNNFMgr,
            const char * const cntlName,
            NNFMgrCollectionCntlInfo* 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	NNFMgrCollection CntlInfo* const	Input	Pointer to a valid NNFMgrCollectionCntl Info structure.

Remarks

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

NNFMgrUpdateCollectionCntl() 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.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted control is used in other formats, 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.

See Also

NNFMgrCollectionCntlInfo

NNFMgrCreateCollectionCntl

NNFMgrGetCollectionCntl

NNFMgrAddCntlToCollection

NNFMgrGetCntlFromCollection

NNFMgrUpdateCollectionCntl

Date and Time Data Types

Based on the international ISO-8601:1988 standard datetime notation: YYYYMMDDHHMMSS.

Combined dates and times can be represented in Numeric, String, and EBCDIC base data types. For some data types, a minimum of eight bytes is required.

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	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 buffer 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 215.

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.

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	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

Returns 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.

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 CollectionName for recursion.

Syntax

```
const short NNFMgrIsRecursiveCollection(
            NNFMgr *pNNFMgr,
            const char* const CollectionName,
            short * const IsRecursive)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit.
CollectionName	const char*	Input	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

Returns 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

NNFMgrIsRecursiveFormat

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

Format Management API Structures

NNFMgrFormatInfo

NNFMgrFormatInfo is a structure containing format information.

Syntax

```
typedef struct NNFMgrFormatInfo {
        unsigned char formatName[NAME_LENGTH+1];
        unsigned char inputInd;
        unsigned char compoundInd;
        long initFlag;
};
```

Parameters

Name	Type	Description
formatName	unsigned char[ ]	Name of format. NULL-terminated string 1 to 120 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 to compound and chose 0. If inputInd = 1, choose 1- 5. If inputInd = 0, choose 1 or 3. 1=IN_COMPOUND_ORDINAL 2=IN_COMPOUND_TAGGED 3=IN_COMPOUND_ALTERNATIVE 4=IN_COMPOUND_RANDOM 5=IN_COMPOUND_TAGGED_RANDOM 1=OUT_COMPOUND_ORDINAL 3=OUT_COMPOUND_ALTERNATIVE
initFlag	long	Uninitialized structure check value.

NNFMgrRepeatFormatInfo

NNFMgrRepeatFormatInfo is a structure containing repeating format information. In earlier versions of New Era of Networks Formatter, the number of repeating output formats was defined by the number of repeating input formats. The user can now specify a fixed number of repeats for a repeating format or repeating group of formats.

Syntax

```
typedef struct NNFMgrRepeatFormatInfo {
        char parentFormatName[NAME_LENGTH+1];
        char childFormatName[NAME_LENGTH+1];
        unsigned char optionalInd;
        unsigned char repeatInd;
        int repeatTermination;
        char repeatDelimiter[NAME_LENGTH+1];
        unsigned repeatCount;
        char repeatFieldName[NAME_LENGTH+1];
        long initFlag;
};
```

Parameters

Name	Type	Description
parentFormatName	char[ ]	Name of parent Format. Must be a NULL-terminated string 1 to 120 characters long, inclusive.
childFormatName	char[ ]	Name of child format. Must be a NULL-terminated string 1 to 120 characters long, inclusive.
optionalInd	unsigned char	Set to zero (0) for a mandatory component, and 1 for an optional component.

Name	Type	Description
repeatInd	unsigned char	If repeatInd in NNFMgrRepeatFormatInfo is zero, then the format is not repeating. Any number other than 0 indicates that the format is repeating.
repeatTermination	int	Termination of repetition. For a list of Termination types, see Remarks.
repeatDelimiter	char[ ]	Name of repetition delimiter separator. Ignored unless repeatTermination is TERMINATION_Delimiter. NULL-terminated string length 1 to 120, inclusive.
repeatCount	unsigned	Positive integer that defines the number of times that format repeats.
repeatFieldName	unsigned char[ ]	Name of field containing the count. NULL-terminated string length 1 to 120, inclusive.
initFlag	long	Uninitialized structure check value.

Remarks

If you are creating a format with fixed output that uses either TERMINATION_NotApplicable or TERMINATION_Delimiter, the following values are available for the repeatTermination parameter:

Termination Type	Value	Description
TERMINATION_Not_Applicable	00	Delimiter not applicable
TERMINATION_Delimiter	01	Delimit this format
TERMINATION_Exact_Length	30	Format contains count; delimiter not applicable

Termination Type	Value	Description
TERMINATION_White_Space_Delimited	31	Format contains count; delimit this format. The count does not appear in the parse tree.
TERMINATION_Minimum_Length_Delimited	40	Field contains count; delimiter not applicable. The incoming message contains the repeat count.
TERMINATION_Minimum_Length_White_Space	41	Field contains count; delimit this format. The incoming message contains the repeat count.

NNFMgrFlatFormatInfo

NNFMgrFlatFormatInfo is a structure containing flat format information.

Syntax

```
typedef struct NNFMgrFlatFormatInfo{
        unsigned int decomposition;
        unsigned int length
        unsigned int termination;
        char delimiter[NAME_LENGTH+1];
        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	unsigned int	Termination of format. One of: TERMINATION_Not_Applicable TERMINATION_Delimiter TERMINATION_Exact_Length TERMINATION_White_Space_Delimited TERMINATION_Minimum_Length_ Delimited TERMINATION_Minimum_Length_White_ Space
delimiter	char[ ]	Name of format delimiter separator. Ignored unless Termination is TERMINATION _Delimiter. NULL-terminated string length 1 to 120 inclusive.
initFlag	long	Uninitialized structure check value.

NNFMgrInFieldInfo

NNFMgrInFieldInfo is a structure containing input field information.

Syntax

```
typedef struct NNFMgrInFieldInfo{
        char formatName[NAME_LENGTH+1];
        char fieldName[NAME_LENGTH+1];
        char controlName[NAME_LENGTH+1];
        long initFlag;

}
```

Parameters

Name	Type	Description
fieldName	char[ ]	Name of field to add to format. NULL-terminated string of length 1 to 120 inclusive.
controlName	char[ ]	Name of output format control associated with new field in format. NULL-terminated string of length 1 to 120 inclusive.
formatName	char[ ]	The format name to add the field to control mapping to.
initFlag	long	Uninitialized structure check value.

NNFMgrOutFieldInfo

NNFMgrOutFieldInfo is a structure containing output field information in an output format.

Syntax

```
typedef struct NNFMgrOutFieldInfo{
        char formatName[NAME_LENGTH+1];
        char fieldName[NAME_LENGTH+1];
        char controlName[NAME_LENGTH+1];
        short accessMode;
        short subscript;
        char inFieldName[NAME_LENGTH+1];
        long initFlag;
}
```

Parameters

Name	Type	Description
formatName	char[ ]	Name of flat output format. NULL-terminated string of length 1 to 120 inclusive.
fieldName	char[ ]	Name of field to add to format. NULL-terminated string of length 1 to 120 inclusive.
controlName	char[ ]	Name of output format control associated with new field in format. NULL-terminated string of length 1 to 120 inclusive.

Name	Type	Description
accessMode	short	One of: ACCESS_MODE_Not_Applicable ACCESS_MODE_Normal_Access ACCESS_MODE_Access_with_Increment ACCESS_MODE_Reset_then_Normal_ Access ACCESS_MODE_Reset_then_Access_ with_ Increment ACCESS_MODE_Access_within_ Compound ACCESS_MODE_Cycling_Access_stay_ in_ Compound ACCESS_MODE_Access_using_Relative_ Index ACCESS_MODE_Create_Field
subscript	short	Used for ACCESS_MODE_Access_nth_Instance_ of_Field where subscript is the instance number starting at 0. It is an error to provide a subscript value when accessMode is not ACCESS_MODE_ Access_nth_Instance_of_Field.
initFlag	unsigned short	Uninitialized structure check value.

Format Management APIs

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.

Syntax

```
const short NNFMgrCreateFormat(
      NNFMgr * pNNFMgr,
      const NNFMgrFormatInfo * const pFormatInfo;
      const NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid NNFMgr session previously returned by NNFMgrInit().
pFormatInfo	const NNFMgrFormat Info * const	Input	Pointer to a valid NNFMgrFormatInfo structure. This pointer cannot be NULL.
pFlatFormat Info	const NNFMgrFlat FormatInfo * 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 ed 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

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 is performed on formatName or fieldName.

Syntax

```
const short NNFMgrAppendFieldToInputFormat(
            NNFMgr *pNNFMgr,
            const char * const pFormatName,
            const NNFMgrInFieldInfo * const pInFieldInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid FMgr session previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of the parent format. NULL-terminated string length 1 to 120 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

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 is performed on formatName or fieldName.

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 120 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

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 is performed on parentFormatName or childFormatName.

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 120 characters in length, inclusive.
pRepeat FormatInfo	const NNFMgr Repeat FormatInfo* const	Input	Pointer to insertion information. This pointer cannot 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

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,
      NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of format to retrieve information for.
pFormatInfo	NNFMgrFormat Info * 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 NNFMgrFlat FormatInfo * const	Output	Pointer to a valid NNFMgrFlatFormat structure.

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

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,
      NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input/ Output	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatInfo	NNFMgrFormat Info * const	Output	Pointer to a valid NNFMgrFormatInfo structure. pFormatInfo cannot be NULL. Structure fields is filled by database values if the call is successful.
pFlatFormatInfo	const NNFMgrFlat FormatInfo * const	Output	Pointer to a valid NNFMgrFlatFormat structure.

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

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,
      NNFMgrFlatFormatInfo * const pFlatFormatInfo);
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr *	Input	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatInfo	NNFMgrFormat Info * 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.
pFlatFormat Info	const NNFMgrFlat FormatInfo * const	Output	Pointer to a valid NNFMgrFlatFormat structure.

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

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pFormatName	const char * const	Input	Name of the parent format. NULL-terminated string length 1 to 120 inclusive.
pInFieldInfo	NNFMgrInField 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

NNFMgrAppendFieldToInputFormat

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pInFieldInfo	NNFMgrInField 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

NNFMgrAppendFieldToInputFormat

NNFMgrGetFirstFieldFromInputFormat

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	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 120 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 is read successfully; zero (0) on failure.

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

See Also

NNFMgrAppendFieldToOutputFormat

NNFMgrGetNextFieldFromOutputFormat

NNFMgrGetNextFieldFromOutputFormat

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pOutFieldInfo	NNFMgr OutField Info * const	Output	Information about the field.

Return Value

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

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

See Also

NNFMgrAppendFieldToOutputFormat

NNFMgrGetFirstFieldFromOutputFormat

NNFMgrGetFirstChildFormat

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	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 120 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 is 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

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	Valid NNFMgr instance pointer previously returned by NNFMgrInit().
pRepeatFormat Info	NNFMgr RepeatFormat Info * const	Output	Repetition information structure.

Return Value

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

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

See Also

NNFMgrGetFirstChildFormat

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.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

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

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.
pFormatInfo	const NNFMgrFormat Info * const	Input	Pointer to a valid NNFMgrFormatInfo structure This pointer cannot be NULL.
pFlatInfo	const NNFMgrFlat FormatInfo * 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 the contents of the NNFMgrFlatFormatInfo structure is not important, pass a zero (0) pointer as the third argument. Input flat formats are 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

NNFMgrUpdateOutMstrCntl

NNFMgrUpdateParseControl

NNFMgrDeleteFormat

Deletes a single input or output format from the database.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

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 deleted format 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.

See Also

NNFMgrDeleteParseControl

NNFMgrDeleteOutMstrCntl

Format Group Management APIs

The Format Group Management APIs are used to get and create format collections.

Format Group API Structure

NNFMgrFormatGroupInfo

NNFMgrFormatGroupInfo is a structure containing format group information.

Syntax

```
typedef struct NNFMgrFormatGroupInfo {
        char groupName[NAME_LENGTH+1];
        char formatName[NAME_LENGTH+1];
        long initFlag;
} NNFMgrFormatGroupInfo;
```

Parameters

Name	Type	Description
groupName	char[ ]	Name of group. NULL-terminated string 1 to 120 characters long, inclusive.
formatName	char[ ]	Name of format. NULL-terminated string 1 to 120 characters long, inclusive.
initFlag	long	Uninitialized structure check value.

Format Group Management APIs

NNFMgrCreateFormatGroup

Creates a new format group.

Syntax

```
const short NNFMgrCreateFormatGroup (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input / Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Takes the groupName in pInfo and creates an empty format group with that name.

When creating a new format group, the formatName parameter in the NNFMgrFormatGroupInfo() structure is ignored.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrGetFormatGroup

NNFMgrGetFirstFormatGroup

NNFMgrGetNextFormatGroup

NNFMgrDeleteFormatGroup

NNFMgrGetFormatGroup

Gets a format group.

Syntax

```
const short NNFMgrGetFormatGroup (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input/ Output	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Based on the groupName in pInfo, it validates whether the format group exists.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrGetFirstFormatGroup

NNFMgrGetNextFormatGroup

NNFMgrDeleteFormatGroup

NNFMgrGetFirstFormatGroupInfo

Gets the first format group.

Syntax

```
const short NNFMgrGet FirstFormatGroup (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input/ Output	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Populates the groupName value of pInfo with the name of the first format group.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrGetFormatGroup

NNFMgrGetNextFormatGroup

NNFMgrDeleteFormatGroup

NNFMgrGetNextFormatGroup

Gets the next new format group.

Syntax

```
const short NNFMgrGetNextFormatGroup (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input/ Output	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Populates the groupName value of pInfo with the name of the next format group.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrGetFormatGroup

NNFMgrGetFirstFormatGroup

NNFMgrDeleteFormatGroup

NNFMgrDeleteFormatGroup

Deletes a format group.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

Syntax

```
const short NNFMgrDeleteFormatGroup (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Deletes the format group with the groupName in pInfo.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrGetFormatGroup

NNFMgrGetFirstFormatGroup

NNFMgrGetNextFormatGroup

NNFMgrAddFormatGroupItem

Adds a format to a format group and verifies that the format group exists.

Syntax

```
const short NNFMgrAddFormatGroupItem (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Adds the format named in pInfo to the named group in pInfo.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrGetFirstFormatGroupItem

NNFMgrGetNextFormatGroupItem

NNFMgrRemoveFormatGroupItem

NNFMgrGetFirstFormatGroupItem

Gets the first item in a format group.

Syntax

```
const short NNFMgrGetFirstFormatGroupItem (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input/ Output	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Populates the first formatName for the group given in the groupName.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrAddFormatGroupItem

NNFMgrGetNextFormatGroupItem

NNFMgrRemoveFormatGroupItem

NNFMgrGetNextFormatGroupItem

Gets the next item in a format group.

Syntax

```
const short NNFMgrGetNextFormatGroupItem (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input/ Output	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Populates the next formatName for the group.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrAddFormatGroupItem

NNFMgrGetFirstFormatGroupItem

NNFMgrRemoveFormatGroupItem

NNFMgrRemoveFormatGroupItem

Removes an item from a format group.

To remove formats, you must have update permission for the format group.

Syntax

```
const short NNFMgrRemoveFormatGroupItem (
      NNFMgr* pNNFMgr,
      NNFMgrFormatGroupInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrFormat GroupInfo* const	Input	Pointer to an initialized NNFMgrFormatGroupInfo structure.

Remarks

Removes the formatName from groupName.

Return Value

Returns a non-zero integer value if the child format is 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

NNFMgrFormatGroupInfo

NNFMgrCreateFormatGroup

NNFMgrAddFormatGroupItem

NNFMgrGetFirstFormatGroupItem

NNFMgrGetNextFormatGroupItem

New Era of Networks Formatter Mapping Management APIs

New Era of Networks Formatter mapping uses explicit input field to output field mapping information. This information is stored in the New Era of Networks Formatter repository as unique map objects.

The data structure NNFMgrMapInfo is used to define a map structure. By calling NNFMgrCreateMap(), the user can create a specific version of a map object. The user can create a new map or a new version of an existing map. By defining a map name and a map version, the user identifies a specific version of a map object to use for mapping input fields to output fields. These two parameters in the AddOutputFormat() function define explicit input field to output field mapping information.

A map object contains the following components:

- Input to output field name mapping
- Output operation or output operation collection
- Output control
- Access mode and subscript specifications

The field name mapping is a required component; the remaining components are optional. If an optional component is not defined, the New Era of Networks Formatter engine uses the corresponding component from the default mapping specified in the output format definition.

During a reformat, the map object is queried by the engine to obtain the input field, access mode or subscript, output operation or collection, and output control. After successfully mapping an input field to an output field, the output operation or operation collection is applied to that input data, and the output control formatting is applied. If the field mapping does not define an access mode or subscript, or an output control, then the corresponding components from the default mapping are used instead. If the specified map object does not define a mapping for a given output field, then the default mapping is used instead.

When a map is used, the access modes move from the output format definition into the mapping layer. Moving the access mode controls is necessary because the structure of the input message combined with the output format definition defines how nested and repeating structures behave.

Mapping can provide input data enrichment and transformation during a message reformat. Data transformation components in the mapping object are applied to field data in the following order:

1. If an access mode is specified in the mapping object, then map the input field specified in the mapping object to the output field using the access mode or subscript.
2. If the access mode in the mapping object specifies "use default," then map the input field specified for the mapping object to the output field using the access mode or subscript specified in the output format definition.
3. If an output operation or output operation collection is specified in the mapping object, then apply the specified operations to the input field data.
4. If the output operation or output operation collection in the mapping object specifies "use default," then map the input field specified in the mapping object to the output field using the access mode or subscript specified in the output format definition.
5. If the output operation or output operation collection in the mapping object specifies "NONE," then the output operation or output operation collection is ignored.
6. If an output control is specified in the mapping object, then apply the operations from that control to the input field data next.
7. If an output control in the mapping object specifies "use default," then apply the output control from the default mapping to the input field data.

Each map object is a named entity. Each map can have any number of versions. A map object can be based on other map objects; for example, a specialized map based on standard maps. Maps can be derived from multiple map definitions.

NNFMapLink is used to define the inheritance linkage of map information from a base map into a derived map. By calling NNFMgrCreateMapLink(), the user constructs a linking of map entries. The map specified by mapName and mapVersion in the NFMgrMapLink() structure inherits all field mapping from the map specified by baseMapName and baseMapVersion. The inheritance is serialized. A previous definition for an output field can be overwritten with a higher level definition. This property can be used to create map repositories.

Note:
Map versions are currently supported only in the New Era of Networks Formatter Management APIs.

A child map can inherit from either a specific version of an ancestor map or from the most current version. Static inheritance means that a child map will only inherit from a specific version of a parent map. Dynamic inheritance means that a child map will always inherit from the most recent version of a parent map. If a child map dynamically inherits from an ancestor map, and a new version of the ancestor map is created, those changes are reflected in the child map.

Calling NNFMgrGetFirstMapLink(), returns the first ancestor map for the specified mapName. The ancestor-child relationships are returned in bottom-up, left-to-right order. The child map inherits field mappings from four different ancestors. The order in which these relationships are returned by NNFMgrGetFirstMapLink() and NNFMgrGetNextMapLink() is: Child-Parent1, Child-Parent4, Parent4-Parent2, and Parent4-Parent3.

Management Data API Structures

NNFMgrMapInfo

NNFMgrMapInfo defines explicit map information.

Syntax

```
typedef struct NNFMgrMapInfo {
        char mapName[NAME_LENGTH+1];
        long mapVersion;
```

```
        char mapDescription[DESCRIPTION_LENGTH+1];
        char outFieldName[NAME_LENGTH+1];
        char inFieldName[NAME_LENGTH+1];
        short accessMode;
        short subscript;
        char childCntlName[NAME_LENGTH+1];
        NNCntlType childCntlType;
        char outputCntlName[NAME_LENGTH+1];
        long initFlag;
}
```

Parameters

Name	Type	Description
mapName	char[ ]	User-defined name for the affected map. NULL-terminated string 1 to 120 characters long, inclusive.
mapVersion	long	Version of the map defined by mapName. If a value of zero (0) is used to access the latest map, the map version in the structure is set to the most current version.
mapDescription	char[ ]	User defined optional description of the map.
outFieldName	char[ ]	Name of the output field. Because there is no restriction for duplicating field names within an output flat format, a mapping on a duplicated output field name results in that field being bound in each instance to the defined inField Name. The outFieldName must exist in the NNF_FIELD table to be accepted.

Name	Type	Description
inFieldName	char[ ]	Name of the input field. Because all input field names must be unique, there is no name space problem. The inFieldName must exist in the NNF_FIELD table to be accepted.
accessMode	short	Enumerated value corresponding to the supported access mode definitions.
subscript	short	Integer value required by some access modes.
childCntlName	char[ ]	Name of existing output operation or output operation collection.
childCntlType	NNCntlType	Defines the type of output operation described by childCntlName. If an entry in the operation table based on the childCntlName and childCntlType cannot be found, the entry is rejected. A value of zero (0) in both childCntlName and chilCntlType specifies that an output operation is not defined for this map entry.
outputCntlName	char[ ]	Name of an existing output control Instead of, or in addition to, specifying an output operation or collection, a field mapping can specify an output control to define the data format and length of an output field. If the named output control does not exist in the database, the entry is rejected.
initFlag	long	Unintialized structure check value.

Remarks

For descriptions of supported access modes, see *Using Access Modes* on page 495.

NNFMgrMapLink

Defines the inheritance linkage of map information from a base map into a derived map.

Syntax

```
typedef struct NNFMgrMapLink {
        char childMapName[NAME_LENGTH+1];
        long childMapVersion;
        char parentMapName[NAME_LENGTH+1];
        long parentMapVersion;
        long level;
        long initFlag;
}
```

Parameters

Name	Type	Description
childMapName	char[ ]	Name of the map object that uses the map defined in parentMapName as its base. Any map entries that exist in the child map override conflicting entries in the parent map.
childMapVersion	long	Version of the map defined by childMapName. If the value is zero (0), the most current version is used.
parentMapName	char[ ]	Existing map object name used as a base for childMapName.
parentMapVersion	long	Version of the map defined by parentMapName. If the value is zero (0), the childMap is dynamically bound to the parent map, which means that the child map always inherits from the most recent version of the parent.

Name	Type	Description
level	long	Map level in the inheritance tree. Level 1 indicates a child-to-immediate-parent relationship; higher numbers are increasingly older ancestors. This parameter is returned from the Management APIs; it is ignored when passed to the APIs.
initFlag	long	Unintialized structure check value.

New Era of Networks Formatter Mapping Management APIs

NNFMgrCreateMap

Creates a new map or a new version of an existing map. If mapVersion is zero (0), the most current version of the map is modified. If the version number is greater than zero (0) and is less than or equal to the latest map version, then that existing version is affected. If mapVersion is greater than the latest map version, a new version of the map is created and given a version number one higher than the most current version.

Syntax

```
const short NNFMgrCreateMap (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrUpdateMap

NNFMgrGetFirstMap

NNFMgrGetNextMap

NNFMgrDeleteMap

NNFMgrGetFirstMap

Returns the first entry from the map table. The entries are returned in order sorted by the map name and map version number from the lowest to the highest.

Syntax

```
const short NNFMgrGetFirstMap (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateMap

NNFMgrUpdateMap

NNFMgrGetNextMap

NNFMgrDeleteMap

NNFMgrGetNextMap

Returns the next map entry. NNFMgrGetFirstMap() must previously be invoked.

Syntax

```
const short NNFMgrGetNextMap (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateMap

NNFMgrUpdateMap

NNFMgrGetFirstMap

NNFMgrDeleteMap

NNFMgrUpdateMap

Updates the map name and version.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateMap (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pOldInfo,
      NNFMgrMapInfo* const pNewInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pOldInfo	NNFMgrMapInfo* const	Input	Pointer to initialized NNFMgrMapInfo structure containing original map name and version.
pNewinfo	NNFMgrMapInfo* const	Input	Pointer to initialized NNFMgrMapInfo structure containing new map name and description.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateMap

NNFMgrGetFirstMap

NNFMgrGetNextMap

NNFMgrDeleteMap

NNFMgrDeleteMap

Deletes an existing map and its associated field mappings. If mapVersion is set to zero (0), the most current version of the map is deleted. If any maps rely on this specific version of the map through the inheritance model, the delete will fail.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

Syntax

```
const short NNFMgrDeleteMap (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMapInfo* const	Input/ Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateMap

NNFMgrUpdateMap

NNFMgrGetFirstMap

NNFMgrGetNextMap

NNFMgrGetMapInfo

Returns information about the map specified by mapName and mapVersion in the NNFMapInfo() structure passed in. If mapVersion is zero (0), information about the most current version of the named map is returned. The mapVersion argument contains the version number of this map.

Syntax

```
const short NNFMgrGetMapInfo (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

Constructs an input field / output field mapping entry in the specified map. If mapVersion is set to zero (0), the most current version of the map is used. If the version number is greater than zero (0) and less than or equal to the latest map version, the latest map version is used. If mapVersion is greater than the latest map version, an error is returned. Duplicate entries based on the key of mapName, mapVersion, and outFieldName are not allowed.

Syntax

```
const short NNFMgrCreateFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input / Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrUpdateFieldMapping

NNFMgrGetFirstFieldMapping

NNFMgrGetNextFieldMapping

NNFMgrDeleteFieldMapping

NNFMgrGetFirstFieldMapping

Returns the first field mapping information for the mapName and mapVersion pair. If mapVersion is set to zero (0), the most current map is used. NNFMgrGetFirstFieldMapping() returns any inherited members of this map.

Syntax

```
const short NNFMgrGetFirstFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input/ Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

NNFMgrUpdateFieldMapping

NNFMgrGetNextFieldMapping

NNFMgrDeleteFieldMapping

NNFMgrGetNextFieldMapping

Returns the next field mapping entry including any inherited members of this map. NNFMgrGetFirstFieldMapping() must previously be invoked.

Syntax

```
const short NNFMgrGetNextFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input/ Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

NNFMgrUpdateFieldMapping

NNFMgrGetFirstFieldMapping

NNFMgrDeleteFieldMapping

NNFMgrGetFieldMapping

Returns the field mapping information for a specific mapName, mapVersion, and outputFieldName. If mapVersion is set to zero (0), the most current map is used. The specified may must contain an entry for the given outputFieldName, or the function returns an error code. NNFMgrGetFieldMapping() returns any inherited members of this map.

Syntax

```
const short NNFMgrGetFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input/ Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

NNFMgrUpdateFieldMapping

NNFMgrGetFirstFieldMapping

NNFMgrGetNextFieldMapping

NNFMgrDeleteFieldMapping

NNFMgrUpdateFieldMapping

Updates an existing input field / output field mapping entry. The key is the mapName, outFieldName, and mapVersion. If mapVersion is set to zero (0), the most current map version is updated. If any of the specified parameters do not exist, an error is returned.

Update permission is based on the username in the database session. The user must have Update permission, or the call fails.

Syntax

```
const short NNFMgrUpdateFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input / Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input / Output	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

NNFMgrGetFirstFieldMapping

NNFMgrGetNextFieldMapping

NNFMgrDeleteFieldMapping

NNFMgrDeleteFieldMapping

Deletes an existing input field / output field mapping entry. The key is the combination of mapName, mapVersion, and outFieldName. If mapVersion is set to zero (0), the most current version of the map is deleted.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

Syntax

```
const short NNFMgrDeleteFieldMapping (
      NNFMgr* pNNFMgr,
      NNFMgrMapInfo* const pInfo)
```

Parameters

Name	Type	Input / Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pInfo	NNFMgrMap Info* const	Input	Pointer to an initialized NNFMgrMapInfo structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapInfo

NNFMgrCreateFieldMapping

NNFMgrUpdateFieldMapping

NNFMgrGetFirstFieldMapping

NNFMgrGetNextFieldMapping

NNFMgrCreateMapLink

Constructs a linkage of map entries. This causes the map specified by mapName and MapVersion in the NNFMgrMapInfo structure to inherit all field mappings from the map specified by baseMapName and baseMapVersion. If mapVersion is set to zero (0), the most current version of the map is used. Duplicate links are not allowed. If baseMapVersion has a value of zero (0), the latest version of the parent map is used. This causes dynamic updates to occur for this linkage. Both child and parent map names and versions must exist.

Syntax

```
const short NNFMgrCreateMapLink (
      NNFMgr* pNNFMgr,
      NNFMgrMapLink* const pLink)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pLink	NNFMgrMap Link* const	Input	Pointer to an initialized NNFMgrMapLink structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapLink

NNFMgrGetFirstMapLink

NNFMgrGetNextMapLink

NNFMgrDeleteMapLink

NNFMgrGetFirstMapLink

Returns the first ancestor map for the specified mapName and mapVersion. If mapVersion is set to zero (0), the most current map is queried for map linkage. The ancestor-child relationships are returned in bottom-up, left-to-right order, starting with the child map.

Syntax

```
const short NNFMgrGetFirstMapLink (
      NNFMgr* pNNFMgr,
      NNFMgrMapLink* const pLink)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pLink	NNFMgrMap Link* const	Input/ Output	Pointer to an initialized NNFMgrMapLink structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapLink

NNFMgrCreateMapLink

NNFMgrGetNextMapLink

NNFMgrDeleteMapLink

NNFMgrGetNextMapLink

Returns the next ancestor map for the specified mapName and mapVersion. NNFMgrGetFirstMapLink() must have been invoked previously.

Syntax

```
const short NNFMgrGetNextMapLink (
      NNFMgr* pNNFMgr,
      NNFMgrMapLink* const pLink)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pLink	NNFMgrMap Link* const	Input/ Output	Pointer to an initialized NNFMgrMapLink structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapLink

NNFMgrCreateMapLink

NNFMgrGetFirstMapLink

NNFMgrDeleteMapLink

NNFMgrDeleteMapLink

Deletes a parent-child map linkage. If either mapVersion or baseMapVersion has a value of zero (0), the latest versions of the child and parent maps are deleted. Both child and parent map names and versions must exist, and a linkage between the two maps must exist.

Delete permission is based on ownership in the database session. The user must have Owner and Update permission, or the call fails.

Syntax

```
const short NNFMgrDeleteMapLink (
      NNFMgr* pNNFMgr,
      NNFMgrMapLink* const pLink)
```

Parameters

Name	Type	Input/ Output	Description
pNNFMgr	NNFMgr*	Input	Pointer to NNFMgr structure returned by NNFMgrInit().
pLink	NNFMgrMap Link* const	Input	Pointer to an initialized NNFMgrMapLink structure.

Return Value

Returns 1 if successful; returns zero (0) if an error occurs. 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

NNFMgrInit

NNFMgrMapLink

NNFMgrCreateMapLink

NNFMgrGetFirstMapLink

NNFMgrGetNextMapLink

New Era of Networks Formatter Management API Error Handling

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

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

Chapter 6
Error Messages

The following error messages are available:

- General New Era of Networks Formatter errors
- Parsing errors
- New Era of Networks Formatter Management API errors

The listed errors are generic. When an error code is set, the error message is enhanced with contextual information. For example, when a format name or field name is referenced in the error message, the appropriate <format_name> or <field_name> is inserted into the error message.

General New Era of Networks Formatter Errors

Code	Description	Response
1001	Mandatory output field <field_name> contains no data due to the following: no associated input field, using an access mode of Not Applicable, or using no data-producing Output Operation (math expression, user exit, or default).	Specify field as optional, or supply default if field is mandatory.
1002	Input format <format_name> not in database.	Supply a correct format name.
1003	<number_of_ characters> trailing characters <trailing_ characters> after message parse. Input format specified incorrectly according to input message; unparsed data remaining in input message.	Check that format specified correctly.

Code	Description	Response
1004	Output format <format_name> not in database.	Supply a correct format name.
1005	Mandatory input field <field_name> has invalid data <data>. Data does not conform to specified data type.	Correct the data type for the field.
1006	Cannot parse entire message into fields.	Call IBM tech support.
1007	Tag <field> parsed with zero length.	Add length to tag <field>
1008	Not all fields found. Missing field: <field_name>. Not all mandatory fields found in random flat format.	Check that format specified correctly.
1009	Output message has zero length.	Call New Era of Networks tech support.
1010	End of sibling sequence.	Call New Era of Networks tech support.
1011	Error in creating New Era of Networks Broker.	Call New Era of Networks tech support.
1012	Error evaluating rules.	Call New Era of Networks tech support.
1013	Error reading output format control: <control_name> from database. Nonexistent output format control specified for field in output format.	Check that format was specified correctly.
1014	<error_text_ from_rules>. New Era of Networks Rules component error evaluating rules	Call New Era of Networks tech support.
1015	Option not found after rules evaluation.	Call New Era of Networks tech support.
1016	Action ‘OutputFormat Control‘not found after rules evaluation.	Call New Era of Networks tech support.

Code	Description	Response
1017	Multiple controls returned from New Era of Networks Rules for field: <field_name>. Control <control_name> and control <control_name>. User specified conflicting evaluation rules. More than one output format control selected based on rules evaluation.	Correct the field rules for conditional branching.
1018	Invalid output format name <format_name> returned from rules evaluation.	Check that format specified correctly.
1019	Invalid format control name <control_name> returned from rules evaluation.	Check that format specified correctly.
1020	No math expression to parse (NULL expression string). User selected Math Expression format control in the GUI, without specifying a math expression.	Specify a math expression.
1021	Math expression parse failed: <expression>. Used improper syntax when creating math expression in Formatter GUI for output format control.	Correct the expression syntax.
1022	Invalid math expression precision value: <precision_value>. Negative value precision selected for a math expression output format control.	Choose a precision that is non-negative.
1023	<name_of_exit_routine>. User exit routine failed.	Debug the exit routine to determine why it failed. Use msgtest and debugger.
1024	<name_of_cleanup_routine>. User exit cleanup routine failed.	Debug the exit cleanup routine to determine why it failed. Use msgtest and debugger.

Code	Description	Response
1025	<name_of_exit_ routine>. User specified exit routine in New Era of Networks Formatter GUI, but did not link routine with application calling New Era of Networks Formatter.	Relink the application with the exit routine.
1026	NULL DBMS session pointer. Zero-valued DbmsSession pointer passed to Formatter constructor.	Pass a valid pointer.
1027	NULL input message buffer pointer. Zero-valued message buffer pointer passed AddInputMessage function.	Pass a valid pointer.
1028	NULL input format name. Zero-valued pointer to input format name passed to New Era of Networks Formatter function.	Pass a valid pointer to a format name.
1029	NULL output format name. Zero-valued pointer to output format name passed to New Era of Networks Formatter function.	Pass a valid pointer to a format name.
1030	Field <field_name> not found. Invalid field name supplied to GetFieldAscii().	Supply a correct field name.
1031	Field in domain <domain_id> not found. Invalid domain identifier supplied to GetFieldAscii()	Supply a correct field name.
1032	Unable to load format <format_name>.	Supply a correct format name.
1033	Cannot find field group.	Call New Era of Networks tech support.
1034	Sequence break occurred.	Call New Era of Networks tech support.

Code	Description	Response
1035	Infinite loop detected while parsing format: <format_name>. Input format specified incorrectly according to the input message. Parser stuck in message and cannot advance pointer into input message buffer.	Check that format specified correctly.
1036	Negative message length is invalid: <length>. Negative message length passed to AddInputMessage().	Pass positive message length to the function.
1037	Repeat count field not found prior to repeating component. Repeating component of compound input format specified as having a repeat count embedded in preceding field value; field not in the message.	Check that format was specified correctly.
1038	Boolean control evaluates to FALSE.	Response not required.
1039	IBM Field specified too wide. IBM types: Zoned and Packed, Signed and Unsigned, restricted to 16 bytes in width. Parse control specified as larger than 16 bytes.	Use API or GUI to correct field width.
1040	IBM Field Decimal Location specified out of range. Parse control specified with decimal location larger than field width. IBM Zoned fields restricted to decimal location between zero and n, where n=specified field width; IBM Packed fields restricted to decimal location of (2*n - 1).	Use API or GUI to correct field decimal location.

Code	Description	Response
1041	parse or reformat called without supplying an input message. AddInputMessage() not used to add input message prior to parsing or reformatting.	Use AddInputMessage() to add message prior to using parse or reformat.
1042	reformat called without specifying an output format. AddOutputFormat() not used to add output format prior to reformatting.	Use AddOutputFormat() to add output format prior to using reformat.
1043	Syntax error in regular expression < > for input field <field_name>.	Troubleshoot operating system to find error.
1044	Fatal error encountered in thread library component.	Troubleshoot operating system to find error.
1045	Memory allocation attempt failed.	Troubleshoot operating system to find error.
1046	Initialization of thread-safe mechanism failed.	Troubleshoot operating system to find error.
1047	User-defined type, input field validation, no validation function found, validation name (of validation routine). User-type validation version of constructor not called or collection supplied to constructor did not contain callback function object with key matching user-defined type validation routine name for the current input field user-defined type.	Call correct Formatter constructor or add required object / key pair to the collection before calling constructor.

Code	Description	Response
1048	User-defined type, input field validation, name/value pair and runtime pointer callback returned failure, validation name (of validation routine). Version of validation callback that takes name/value pair array and pointer to run-time data returned failure.	If validation callback is written correctly, fix input message. If input message is correct, fix validation callback.
1049	User-defined type, input field validation, name/value pair callback returned failure, validation name (of validation routine). Version of validation callback that takes name/value pair array returned failure.	If validation callback written correctly, fix input message. If input message correct, fix validation callback.
1050	User-defined type, input field validation, runtime pointer callback returned failure, validation name (of validation routine). Version of validation callback that takes pointer to run-time data returned failure.	If validation callback written correctly, fix input message. If input message correct, fix validation callback.
1051	User-defined type, input field validation, no-user-parameters callback returned failure, validation name (of validation routine); Version of validation callback that takes no user parameters returned failure.	If validation callback written correctly, fix input message. If input message correct, fix validation callback.
1052	Corrupt New Era of Networks Formatter database data detected;	Run the Consistency Checker.
1053	Conversion to output type failed for <type of error>; Error translating tag, length, data or error in conversion.	If error indicates tag, length, or data conversion resulted in invalid data type output, check format data type and format definition to verify they correspond. If error indicates out data type, the conversion not attempted.

Code	Description	Response
1054	Repeat termination delimiter <delimiter value> not found for repeating format <format>. Data field has termination delimiter defined in definition and input message does not.	Check that the data field has termination delimiter defined in input format definition, or change input format definition to match data. Run apitest –d to help isolate field missing termination delimiter.
1055	Infinite loop detected mapping fields for format <format> (possible cause: compound format using normal access mode control). Failed to terminate mapping portion defined by access modes.	Check that access modes match the data.
1056	Mandatory input field <field> not found.	Check input message data field using apitest –d to isolate specific field or input format definition.
1057	Output format <format> not found; Attempt to remove output format that does not exist.	No impact.
1058	Access mode specified for field <field> in format <format> requires controlling field.	No longer used.
1059	Zero length substring for output field <field> invalid.	No longer used.
1060	<field> field (data type <datatype>) has invalid data <data>.	Check output field data type against data from associated input data.
1061	Output field <field> (data type <datatype>) mapped to input field <field> at byte offset <byte offset> has invalid data <data>.	Check datatype of output field and check parsed input field data.
1062	Failed to determine correct machine type endian of type Int4. Processing unknown Endian type.	Call New Era of Networks technical support.

Code	Description	Response
1063	Conversion of (data type <datatype>) failed for <value>, of length <length>. Input field data does not match input field definition.	Check that input field data is compatible with input control definition.
1064	Data type (<datatype>) invalid.	Call New Era of Networks technical support.
1065	Data overflow for data <data value> (datatype <datatype>).	Check format definition, input field data and corresponding output field format definition.
1066	Output mandatory format <format> contains no fields.	Check output format definition and input data fields used in output format.
1067	Data (<data>) has odd length (<length>) or cannot be converted to Binary. Binary data must have even length.	Check that input data length is even.
1068	Y2K-conversion of (data type <datatype>, base data type <base datatype>) to internal ISO ASCII String representation failed for <data>, Format Attr Id <format>, Century <century>, Year Cut-off <year cutoff>. Year is empty, year string is greater than 2 digits, year is negative, or year cutoff is not between 0 and 100 inclusive.	Check input message field data and input format definition.
1069	Invalid component (<type>=<value>) in <type of filed> data <data value>.	The date or time component value is invalid. Check input message and format definition.
1070	Invalid year cut-off.	The year cut-off in the format definition needs to be between 1 and 100 inclusive.
1071	Cannot resolve data type <data_type> from supplied parameters.	The custom date time datatype needs appropriate base datatype.

Code	Description	Response
1072	[Parse Debugger] Bad output stream.	Connection to open file stream list. Restart application.
1073	[Parse Debugger] Debug mode is already on.	No response.
1074	[Parse Debugger] Failed to assign debug object. Thread-specific data not stored correctly.	Troubleshoot threading.
1075	[Parse Debugger] Invalid debug category.	No longer used.
1076	[Parse Debugger] Invalid verbose level.	No longer used.
1077	Error loading <subcontrol>.	Check operation specified in error message.
1078	Sub-control error: <subcontrol>.	Check operation definitions and data type converted input field data.
1079	Cannot create datatype <datatype> in output field control <output field control>.	Check output control data type.
1080	Database internal error. Database fails.	Review the status log for errors. Run the Consistency Checker.
1081	Unable to invoke user exit.	Check that the user exit function name is defined.
1082	Unable to load map <map_name> or version <map.version>.	Check that the map name and version are correct.
1083	Unable to explicitly map the field <field names> using output field control type 'Field Eval.'	Check format and rules definitions.
1084	Format name exceeds 120 byte maximum.	Check for valid length of format name.

Code	Description	Response
1085	User-defined data type error <error code> (<hint>).	Check the user-defined data type. It failed the user-defined validation routine. The values of <error code> and <hint> are supplied by the routine.
1086	User exit returned an undefined data type.	Check the data type of the user exit. It should be an integer, "double," or an array of characters.
1087	Sub-control error: Behavior failed on output field <infield>, <outfield>.	Check each output control in the output operation collection. At least one output control in the collection failed when the operation was applied to the indicated output field using the data from the indicated input field.
1088	Sub-controls (in reverse order)	Check the order of each sub-control and the components of the operation definitions and converted input field data type.

Parsing Errors

Code	Description	Response
-1	User specified exact length for field in parse control; actual field data too large for input field.	Correct length in parse control.
-2	User specified length component for field in parse control; message length too large for input field	Correct length in parse control.
-3	Minimum length too large for input field.	Correct length in parse control.
-4	Delimiter not found for input field.	Check that format specified correctly.

Code	Description	Response
-5	User specified white space delimited field in parse control; white space not found.	Check that format specified correctly.
-6	Literal not found for input field.	Check that format specified correctly.
-7	Literal delimiter not found for input field.	Check that format specified correctly.
-8	No match for regular expression for input field.	Check that regular expression in input control is correctly specified.
-9	Regular expression delimiter not found for input field.	Check that the format is correctly specified.

New Era of Networks Formatter Management API Error Messages

Code	Description	Response
0	No error encountered	None required.
-2600	Feature not implemented.	None required.
-2601	One or more of the required fields in the API data structure missing.	Check data passed to function.
-2602	Prerequisite New Era of Networks Formatter component missing.	Check data passed to function.
-2603	Database inconsistency detected.	Call New Era of Networks technical support.
-2604	Database error occurred.	Call New Era of Networks technical support.
-2605	Requested component not found in database.	Check data passed to function.
-2606	Database in inconsistent state; probably resulting from failed update.	Call New Era of Networks technical support.
-2607	One or more required fields in API data structure contains invalid value.	Check data passed to function.

Code	Description	Response
-2608	Data type field in API structure contains invalid value.	Check data passed to function.
-2609	Operation type field in API structure contains invalid value.	Check data passed to function.
-2610	Base data type field in API structure contains invalid value.	Check data passed to function.
-2611	Field type field in API structure contains invalid value.	Check data passed to function.
-2612	Control type field in API structure contains invalid value.	Check data passed to function.
-2613	Specified output operation not found in database.	Check data passed to function.
-2614	Trim operation is invalid; probably resulting from corrupt database.	Run Consistency Checker.
-2615	Prefix / Postfix operation invalid; probably resulting from corrupt database.	Run Consistency Checker.
-2616	Case operation invalid; probably resulting from corrupt database.	Run Consistency Checker.
-2617	Justify operation invalid; probably resulting from corrupt database.	Run Consistency Checker.
-2618	NULL action invalid; probably resulting from corrupt database.	Run Consistency Checker.
-2619	Tag literal specified in output control invalid.	Check data passed to function.
-2620	Field name specified in "Input Value=" output control invalid.	Check data passed to function.
-2621	Substitute value not specified in output operation.	Check data passed to function.

Code	Description	Response
-2622	Trim literal not specified in output operation.	Check data passed to function.
-2623	Prefix / Postfix literal not specified in output operation.	Check data passed to function.
-2624	Default literal not specified in output operation.	Check data passed to function.
-2625	Output operation cannot be converted from 4.0 output control to 3.2 output control.	Check data passed to function.
-2626	Substring start or length parameters invalid.	Check data passed to function.
-2627	Output master control does not have a collection as its child and cannot accept sub-controls.	Check data passed to function.
-2628	Date / time fields require data only or tag / data and exact length termination.	Check data passed to function.
-2629	Date / time fields with 2 digits years must be given a year cutoff value between 0 and 100 inclusive.	Check data passed to function.
-2630	Substring to end of field length=0 must have pad=NONE.	Check data passed to function.
-2640	Bad file.	Bad file name given for reading a field map from a file. Check file.
-2641	Incorrect version.	Incorrect file version in file containing field map. Check file.
-2642	Call to Management API failed due to attempt to add duplicate component.	Map already contains the output field. Check field mapping definition.
-2643	Call to NNFMgrCreateMapLink() failed due to attempt to link map to itself, directly or indirectly.	Recursive map link. Check the map relationships.

Code	Description	Response
-4011	Invalid NNFie file	Make sure file was generated by NNFie.
-5021	User does not have Update permission on specific component.	Owner must give user or PUBLIC Update permission for the component.
-5022	User does not have Delete permission on specific component.	User must have update permission and be the owner to delete the component.
-5505	Permission table entry cannot be created for specific component; indicates database failure.	Run Consistency Checker.

Appendix A
Character Sets

For blank entries, see the relevant EBCDIC code pages.

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.

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
000	00	NUL	NUL	Null	0000 0000
001	01	SCH	SOH	Start of Heading	0000 0001
002	02	STX	STX	Start of Text	0000 0010
003	03	ETX	ETX	End of Text	0000 0011
004	04	EOT	SEL	Select	0000 0100
005	05	ENO	HT	Horizontal Tab	0000 0101
006	06	ACK	RNL	Required New Line	0000 0110
007	07	BEL	DEL	Delete	0000 0111
008	08	BS	GE	Graphic Escape	0000 1000
009	09	HT	SPS	Superscript	0000 1001
010	0A	LF	RPT	Repeat	0000 1010
011	0B	VT	VT	Vertical Tab	0000 1011
012	0C	FF	FF	Form Feed	0000 1100

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
013	0D	CR	CR	Carriage Return	0000 1101
014	0E	SO	SO	Shift Out	0000 1110
015	0F	SI	SI	Shift In	0000 1111
016	10	DLE	DLE	Data Link Escape	0001 0000
017	11	DC1	DC1	Device Control 1	0001 0001
018	12	DC2	DC2	Device Control 2	0001 0010
019	13	DC3	DC3	Device Control 3	0001 0011
020	14	DC4	RES/ENP	Restore/Enable Presentation	0001 0100
021	15	NAK	NL	New Line	0001 0101
022	16	SYN	BS	Backspace	0001 0110
023	17	ETB	POC	Program-Operator Communication	0001 0111
024	18	CAN	CAN	Cancel	0001 1000
025	19	EM	EM	End of Medium	0001 1001
026	1A	SUB	UBS	Unit Backspace	0001 1010
027	1B	ESCAPE	CU1	Customer Use 1	0001 1011
028	1C	FS	IFS	Interchange File Separator	0001 1100
029	1D	GS	IGS	Interchange Group Separator	0001 1101
030	1E	RS	IRS	Interchange Record Separator	0001 1110

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
031	1F	US	IBT / IUS	Intermediate Transmission Block / Interchange Unit Separator	0001 1111
032	20	SPACE	DS	Digit Select	0010 0000
033	21	!	SOS	Start of Significance	0010 0001
034	22	“	FS	Field Separator	0010 0010
035	23	#	WUS	Word Underscore	0010 0011
036	24	$	BYP / INP	Bypass / Inhibit Presentation	0010 0100
037	25	%	LF	Line Feed	0010 0101
038	26	&	ETB	End of Transmission Block	0010 0110
039	27	‘	ESC	Escape	0010 0111
040	28	(	SA	Set Attribute	0010 1000
041	29	)	SFE	Start Field Extended	0010 1001
042	2A	*	SM / SW	Set Mode / Switch	0010 1010
043	2B	+	CSP	Control Sequence Prefix	0010 1011
044	2C	,	MFA	Modify Field Attribute	0010 1100
045	2D	-	ENQ	Enquiry	0010 1101
046	2E	.	ACK	Acknowledge	0010 1110
047	2F	/	BEL	Bell	0010 1111
048	30	0			0011 0000

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
049	31	1			0011 0001
050	32	2	SYN	Synchronous Idle	0011 0010
051	33	3	IR	Index Return	0011 0011
052	34	4	PP	Presentation Position	0011 0100
053	35	5	TRN	Transparent	0011 0101
054	36	6	NBS	Numeric Backspace	0011 0110
055	37	7	EOT	End of Transmission	0011 0111
056	38	8	SBS	Subscript	0011 1000
057	39	9	IT	Indent Tab	0011 1001
058	3A	:	RFF	Required Form Feed	0011 1010
059	3B	;	CU3	Customer Use 3	0011 1011
060	3C	<	DC4	Device Control 4	0011 1100
061	3D	=	NAK	Negative Acknowledge	0011 1101
062	3E	>			0011 1110
063	3F	?	SUB	Substitute	0011 1111
064	40	@	SP	Space	0100 0000
065	41	A	RSP		0100 0001
066	42	B			0100 0010
067	43	C			0100 0011
068	44	D			0100 0100
069	45	E			0100 0101

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
070	46	F			0100 0110
071	47	G			0100 0111
072	48	H			0100 1000
073	49	I			0100 1001
074	4A	J	¢		0100 1010
075	4B	K	.		0100 1011
076	4C	L	<		0100 1100
077	4D	M	(		0100 1101
078	4E	N	+		0100 1110
079	4F	O	|		0100 1111
080	50	P	&		0101 0000
081	51	Q			0101 0001
082	52	R			0101 0010
083	53	S			0101 0011
084	54	T			0101 0100
085	55	U			0101 0101
086	56	V			0101 0110
087	57	W			0101 0111
088	58	X			0101 1000
089	59	Y			0101 1001
090	5A	Z	!		0101 1010
091	5B	[	$		0101 1011

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
092	5C	\	*		0101 1100
093	5D	]	)		0101 1101
094	5E	^	;		0101 1110
095	5F	-	¬		0110 1111
096	60	‘	-		0110 0000
097	61	a	/		0110 0001
098	62	b			0110 0010
099	63	c			0110 0011
100	64	d			0110 0100
101	65	e			0110 0101
102	66	f			0110 0110
103	67	g			0110 0111
104	68	h			0110 1000
105	69	i			0110 1001
106	6A	j	|		0110 1010
107	6B	k	,		0110 1011
108	6C	l	%		0110 1100
109	6D	m	_		0110 1101
110	6E	n	>		0110 1110
111	6F	o	?		0110 1111
112	70	p			0111 0000
113	71	q			0111 0001

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
114	72	r			0111 0010
115	73	s			0111 0011
116	74	t			0111 0100
117	75	u			0111 0101
118	76	v			0111 0110
119	77	w			0111 0111
120	78	x			0111 1000
121	79	y			0111 1001
122	7A	z	:		0111 1010
123	7B	{	#		0111 1011
124	7C	|	@		0111 1100
125	7D	}	'		0111 1101
126	7E	~	=		0111 1110
127	7F	DEL	"		0111 1111
128	80	Ä	Ä		1000 0000
129	81	unused	a		1000 0001
130	82	,	b		1000 0010
131	83	*f*	c		1000 0011
132	84	„	d		1000 0100
133	85	…	e		1000 0101
134	86	†	f		1000 0110
135	87	‡	g		1000 0111

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
136	88	ˆ	h		1000 1000
137	89	‰	i		1000 1001
138	8A	Š			1000 1010
139	8B	‹			1000 1011
140	8C	Œ			1000 1100
141	8D	unused			1000 1101
142	8E	unused			1000 1110
143	8F	unused			1000 1111
144	90	unused			1001 0000
145	91	‘	j		1001 0001
146	92	‘	k		1001 0010
147	93	“	l		1001 0011
148	94	”	m		1001 0100
149	95	•	n		1001 0101
150	96	–	o		1001 0110
151	97	—	p		1001 0111
152	98	˜	q		1001 1000
153	99	™	r		1001 1001
154	9A	š			1001 1010
155	9B	›			1001 1011
156	9C	œ			1001 1100
157	9D	unused			1001 1101

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
158	9E	unused			1001 1110
159	9F	Ÿ			1001 1111
160	A0	nonbreaking space			1010 0000
161	A1	¡	~		1010 0001
162	A2	¢	s		1010 0010
163	A3	£	t		1010 0011
164	A4	¤	u		1010 0100
165	A5	¥	v		1010 0101
166	A6	¦	w		1010 0110
167	A7	§	x		1010 0111
168	A8	¨	y		1010 1000
169	A9	©	z		1010 1001
170	AA	ª			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	²			1011 0010

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
179	B3	³			1011 0011
180	B4	´			1011 0100
181	B5	µ			1011 0101
182	B6	¶			1011 0110
183	B7	·			1011 0111
184	B8	¸			1011 1000
185	B9	¹			1011 1001
186	BA	º			1011 1010
187	BB	»			1011 1011
188	BC	¼			1011 1100
189	BD	½			1011 1101
190	BE	¾			1011 1110
191	BF	¿			1011 1111
192	C0	À	{		1100 0000
193	C1	Á	A		1100 0001
194	C2	Â	B		1100 0010
195	C3	Ã	C		1100 0011
196	C4	Ä	D		1100 0100
197	C5	Å	E		1100 0101
198	C6	Æ	F		1100 0110
199	C7	Ç	G		1100 0111
200	C8	È	H		1100 1000

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
201	C9	É	I		1100 1001
202	CA	Ê	SHY		1100 1010
203	CB	Ë			1100 1011
204	CC	Ì			1100 1100
205	CD	Í			1100 1101
206	CE	Î			1100 1110
207	CF	Ï			1100 1111
208	D0	Ð	}		1101 0000
209	D1	Ñ	J		1101 0001
210	D2	Ò	K		1101 0010
211	D3	Ó	L		1101 0011
212	D4	Ô	M		1101 0100
213	D5	Õ	N		1101 0101
214	D6	Ö	O		1101 0110
215	D7	×	P		1101 0111
216	D8	Ø	Q		1101 1000
217	D9	Ù	R		1101 1001
218	DA	Ú			1101 1010
219	DB	Û			1101 1011
220	DC	Ü			1101 1100
221	DD	Ý			1101 1101
222	DE	Þ			1101 1110

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
223	DF	ß			1101 1111
224	E0	à	\		1110 0000
225	E1	á			1110 0001
226	E2	â	S		1110 0010
227	E3	ã	T		1110 0011
228	E4	ä	U		1110 0100
229	E5	å	V		1110 0101
230	E6	æ	W		1110 0110
231	E7	ç	X		1110 0111
232	E8	è	Y		1110 1000
233	E9	é	Z		1110 1001
234	EA	ê			1110 1010
235	EB	ë			1110 1011
236	EC	ì			1110 1100
237	ED	í			1110 1101
238	EE	î			1110 1110
239	EF	ï			1110 1111
240	F0	ð	0		1111 0000
241	F1	ñ	1		1111 0001
242	F2	ò	2		1111 0010
243	F3	ó	3		1111 0011
244	F4	ô	4		1111 0100

Decimal Value	Hex Value	ASCII Extended Value	EBCDIC Value	EBCDIC Description	Binary
245	F5	õ	5		1111 0101
246	F6	ö	6		1111 0110
247	F7	÷	7		1111 0111
248	F8	ø	8		1111 1000
249	F9	ù	9		1111 1001
250	FA	ú			1111 1010
251	FB	û			1111 1011
252	FC	ü			1111 1100
253	FD	ý			1111 1101
254	FE	þ			1111 1110
255	FF	ÿ	EO	Eight Ones	1111 1111

Appendix A

Appendix B
Data Types

Not Applicable

No data type is assumed.

String

A string of character data that is associated with a code set. For more information on conversion, see *Data Type Conversion* on page 472.

Numeric

A string of standard ASCII numeric characters.

Binary

The Binary data type is used to parse any value and transform that value to an ASCII representation of the value internally in New Era of Networks Formatter. The internal representation takes each byte of the input value and converts it to a readable form. An example of this is parsing a byte whose value is (hexadecimal) 0x9C and transforming that to the internal ASCII representation of 9C, which is the hexadecimal value 0x3943. If this value is used in an output format with the output control's data type set to String, the value placed in the message is ASCII 0x9C. If this value is again placed in an output message with the data type Binary, the ASCII value is not printable and occupies one byte with the value of (hexadecimal) 0x9C.

Conversely, an input value of ASCII 3B7A parsed with the String data type can be output using the Binary data type. The output value is (hexadecimal) 0x37BA and occupies 2 bytes in the output message. Valid characters that can be converted to Binary from the String data type are 0 through 9 and A through F. All other characters are invalid.

EBCDIC

A string of characters encoded using the EBCDIC (Extended Binary Coded Decimal Interchange Code) encoding that is 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).

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

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; all numbers are assumed to be positive.

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.

The following example illustrates how to generate a default value for an IBM Packed Integer:

The input data type is IBM Signed Packed Decimal, with a default ASCII value of -12345. The control is optional and there is no corresponding field in the input message, so New Era of Networks Formatter uses the default value,

converts it to IBM Signed Packed Decimal, and generates the following output: 12 34 5D. Each pair of numbers represents the two nibbles of a byte. The result is three bytes long.

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).

IBM Signed Zoned Integer

Data type on larger IBM computers used to represent integers. Each decimal digit is represented by a byte. The left nibble of each byte, except the last byte, is a hexadecimal F. The left nibble of the last byte is a hexadecimal C if the number is positive, and a hexadecimal D if the number is negative. The right nibble of each byte is the hexadecimal value of the digit. For example, 1234 is represented as F1 F2 F3 C4 (the number pairs show the hexadecimal values of the nibbles of each byte). -1234 is represented as F1 F2 F3 D4.

Little Endian 2

Two-byte integer where the bytes are ordered with the rightmost byte being the high order or most significant byte. For example, the hexadecimal number 0x0102 is stored as 02 01 (where the number pairs show the hexadecimal values of the nibbles of a byte).

Little Swap Endian 2

Two-byte integer where the two bytes are swapped with respect to a Little Endian 2 value. For example, the hexadecimal number 0x0102 is stored as 01 02.

Little Endian 4

Four-byte integer where two bytes of each word are swapped with respect to Little Endian 4. For example, the hexadecimal number 0x01020304 is stored as 03 04 01 02.

Little Swap Endian 4

Four-byte integer where the bytes are ordered with the rightmost byte being the high order or more significant byte. For example, the hexadecimal number 0x0102 is stored as 02 01, where the number pairs show the hexadecimal values of the nibbles as a byte.

Big Endian 2

Two-byte integer where the bytes are ordered with the leftmost byte being the high order or most significant byte. For example, the hexadecimal number 0x0102 is stored as 01 02 (where the number pairs show the hexadecimal values of the nibbles of a byte).

Big Swap Endian 2

Two-byte integer where the two bytes are swapped with respect to a Big Endian 2 value. For example, the hexadecimal number 0x0102 is stored as 02 01.

Big Endian 4

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

Four-byte integer where the two bytes of each word are swapped with respect to a Big Endian 4 value. For example, the hexadecimal number 0x01020304 is stored as 02 01 04 03.

Decimal, International

Data type where every third number left of the decimal point is preceded by a period. The decimal point is represented by a comma. Numbers right of the decimal point represent a fraction of one unit. For example, the number 12345.678 is represented as 12.345,678. Decimal international data types can contain negative values.

Decimal, U.S.

Data type where every third number left of the decimal point is preceded by a comma. The decimal point is represented by a period. Numbers right of the decimal point represent a fraction of one unit. For example, the number 12345.678 is represented as 12,345.678. Decimal US data types can contain negative values.

Unsigned Little Endian 2

Like Little Endian 2, except that the value is interpreted as an unsigned value.

Unsigned Little Swap Endian 2

Like Little Swap Endian 2, except that the value is interpreted as an unsigned value.

Unsigned Little Endian 4

Like Little Endian 4, except that the value is interpreted as an unsigned value.

Unsigned Little Swap Endian 4

Like Little Swap Endian 4, except that the value is interpreted as an unsigned value.

Unsigned Big Endian 2

Like Big Endian 2, except that the value is interpreted as an unsigned value.

Unsigned Big Swap Endian 2

Like Big Swap Endian 2, except that the value is interpreted as an unsigned value.

Unsigned Big Endian 4

Like Big Endian 4, except that the value is interpreted as an unsigned value.

Unsigned Big Swap Endian 4

Like Big Swap Endian 4, except that the value is interpreted as an unsigned value.

Date

Based on the international ISO-8601:1988 standard date notation: YYYYMMDD where YYYY represents the year in the 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 can be represented in Numeric, String, and EBCDIC base data types. For some data types, a minimum of four bytes is required.

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

Time

Based on the international ISO-8601:1988 standard time notation: HHMMSS, where HH represents the number of complete hours passed since midnight (00-23), MM is the number of minutes passed since the start of the hour (00-59), and SS is the number of seconds since the start of the minute (00-59). Times are represented in 24-hour format.

Times can be represented in Numeric, String, and EBCDIC base data types. For some data types, a minimum of four bytes is required.

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see the ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

Date and Time

Based on the international ISO-8601:1988 standard datetime notation: YYYYMMDDHHMMSS.

Combined dates and times can be represented in Numeric, String, and EBCDIC base data types. For some data types, a minimum of eight bytes is required.

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see the ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

Custom Date and Time

Custom Date and Time enables users to define formats for dates, times, and combined dates and times. Custom Date and Time can be represented in Numeric, String, and EBCDIC data types.

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more

information on supported code sets, see the ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

Date/Time format specifications are strings made up of the following character combinations:

String	Value Range	Description
YY	00-99	2-digit year
YYYY	1900-2xxx	4-digit year
MON	JAN-DEC	3-letter abbreviation for month name
MN	01-12	2-digit month
DD	01-31	2-digit day of month
HH	01-23	2-digit hour of day
MM	01-59	2-digit minute of hour
SS	01-59	2-digit second of minute
AM	none	AM indicator
PM	none	PM indicator
JJJ	001-366	3-digit day-of-year; Julian date

The date/time format specifiers can be in any order. For String and EBCDIC underlying data types, the specifiers can include spaces and delimiters. A format specification string cannot include both 2-digit and 4-digit strings, or both numeric months and abbreviate month names.

New Era of Networks Formatter defines several different date and time representations that can be parsed and output. The following is a list of the custom formats supplied with New Era of Networks Formatter:

MM/DD/YY	MN/DD/YY HH:MM	MNDDYY
DD/MN/YY	DD/MN/YY HH:MM	MNDDYYYY
MN/DD/YYYY	MN/DD/YY HH:MM:SS PM	DDMONYY
DD/MN/YYYY	DD/MN/YY HH:MM:SS PM	DDMONYYYY
DD-MON-YY	MN/DD/YY HH:MM:SS	MONYY
DD-MON-YYYY	DD/MN/YYHH:MM:SS	MONYYYY
MON-YY	HH:MM PM	MONDDYYYY
MON-YYYY	HH:MM	MNDDYYHHMM
MN/DD/YY HH:MM PM	HH:MM:SS PM	MNDDYYHHMMSS
DD/MN/YY HH:MM PM	HH:MM:SS	HHMMSS

Data Type Conversion

Converting to String Representation

The following table describes the source values for each data type and how the data is converted to an intermediate String representation.

Data Type	Source Data Value Range	Intermediate String Representation
Not Applicable	Any value, any length.	Source value is unchanged.
String	A string of characters using one of the supported code sets that is any length. For more information on code sets, set the Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms ***Installation Guide***.	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 converted to the string 0x0123AC, and the prefix 0x is prepended to the data.

Data Type	Source Data Value Range	Intermediate String Representation
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. Characters in the EBCDIC code set that are not in the native code set are converted to a native encoded space character. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
IBM Packed Integer	Maximum 16-byte value. Each nibble can contain the hexadecimal value 0-9. 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 0-9.	String that represents the number. The last nibble contains a hexadecimal C for positive numbers and a hexadecimal D for negative integers. 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.

Data Type	Source Data Value Range	Intermediate String Representation
IBM Zoned Integer	The left nibble of each byte is a hexadecimal F. The right nibble of each byte is a hexadecimal 0-9.	String that represents the number. For example, the zoned integer value F1 F2 F3 F4 F5 becomes 12345.
IBM Signed Zoned Integer	The left nibble of each byte 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 0-9.	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 (>= 0).

Data Type	Source Data Value Range	Intermediate String Representation
Decimal, International	A character string representing a number, where every third digit left of the decimal point is preceded by a period (.). The decimal point is represented by a comma (,). The value can be preceded by + or -. For example, the number 12345.678 is represented as 12.345,678. Numbers that contain no separators or a 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 (.). The value can be preceded by + or -. For example, the number 12345.678 is represented as 12,345.678. Numbers that contain no separators or a 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 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.

Data Type	Source Data Value Range	Intermediate String Representation
Date and Time	A 14-byte numeric string. Each character is in the range 0-9 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 0-9 that represents a valid time in the international ISO-8601:1988 standard datetime notation: HHMMSS. The base data type can 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 0-9 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.

Data Type	Source Data Value Range	Intermediate String Representation
Custom Date and Time	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 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.

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 New Era of Networks Formatter does not attempt to change the data between the input message and the output message.

String

A string of character data that is associated with a code set.

Output Data Type	Constraints
Not Applicable	The data remains unchanged between the input message and the output message.
String	The data type of the input message remains unchanged in the output message unless the two code sets differ. When the input code set is different than the output code set, a data conversion occurs.

Output Data Type	Constraints
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. The string is interpreted as the string encoded hexadecimal form of a binary field. For example, the string0123AC 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	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 4Big Swap Endian 4	Valid only if the string represents an integer and has a value that is within the range allowed for Endian 4 types.

Output Data Type	Constraints
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if the string represents 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.
Date and Time Custom Date and Time Time Date	Valid only if the string is in the form based on the international ISO-8601:1988 standard datetime notation: YYYYMNDDHHMMSS.

Numeric

A numeric string is a sequence of characters encoded in the native encoding for the machine on which New Era of Networks Formatter executes. A numeric string contains only the characters 0 to 9 (no + or -).

Output Data Type	Constraints
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).

Output Data Type	Constraints
EBCDIC Data	This is a valid conversion. Each ASCII character is converted to its EBCDIC equivalent NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
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.
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.

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

Binary

Binary indicates a sequence of binary characters.

Output Data Type	Constraints
Not Applicable	Same behavior as 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.

Output Data Type	Constraints
EBCDIC Data	Same behavior as String, except each character is an EBCDIC encoded character instead of a native encoded character. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
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.
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.

Note:
The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.

Output Data Type	Constraints
Not Applicable	Same behavior as String.
String	A valid conversion in which each EBCDIC character is converted to an ASCII equivalent takes place when the string data is compatible with the US EBCDIC code set. If a value in the US EBCDIC code set does not have an equivalent value in the native-encoded character set, it is 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).

Output Data Type	Constraints
EBCDIC Data	The data remains unchanged between the input message and the output message. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
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.

Output Data Type	Constraints
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if the string represents 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.

Output Data Type	Constraints
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.

Output Data Type	Constraints
EBCDIC Data	This is a valid conversion. The value, incorporating the implied decimal point, is converted to an EBCDIC-encoded string representing the number. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
IBM Packed Integer IBM Signed Packed Integer IBM Zoned Integer IBM Signed Zoned Integer	This is a valid conversion for all pairs of data types, as long as values in the source data type are in the range allowed for the target data type.
Little Endian 2 Little Swap Endian 2 Big Endian 2 Big Swap Endian 2	Valid only if the number is an integer and has a value within the range allowed for the Endian 2 types.
Unsigned Little Endian 2 Unsigned Little Swap Endian 2 Unsigned Big Endian 2 Unsigned Big Swap Endian 2	Valid only if the number is an integer and has a value within the range allowed for the Unsigned Endian 2 types.
Little Endian 4 Little Swap Endian 4 Big Endian 4 Big Swap Endian 4	Valid only if number is an integer and has a value within the range allowed for the Endian 4 types.

Output Data Type	Constraints
Unsigned Little Endian 4 Unsigned Little Swap Endian 4 Unsigned Big Endian 4 Unsigned Big Swap Endian 4	Valid only if 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.

Output Data Type	Constraints
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.

Output Data Type	Constraints
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.
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 4 Types

This is a 4-byte numeric type that includes: Little Endian 4, Little Swap Endian 4, Big Endian 4, Big Swap Endian 4, Unsigned Little Endian 4,

Unsigned Little Swap Endian 4, Unsigned Big Endian 4, and Unsigned Big Swap Endian 4.

Output Data Type	Constraints
Not Applicable	Same behavior as 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. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
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.

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

Decimal International and Decimal US

A Decimal International or Decimal US value is a string representing a number, where every third digit left of the decimal point is preceded by a comma (Decimal US) or a period (Decimal International). The decimal point is represented by a period (Decimal US) or a comma (Decimal International). A + or a - 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.

Output Data Type	Constraints
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.
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.

Output Data Type	Constraints
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. NOTE: The EBCDIC data type should not be used for new applications. It exists only to support previous user formats. If data needs to be encoded in EBCDIC, you must implement one of the supported EBCDIC code sets. For more information on supported code sets, see ***Rules and Formatter Extension for IBM ® WebSphere Message Broker for Multiplatforms User's Guide***.
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.

Output Data Type	Constraints
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).

Appendix C

Using Access Modes

Access modes determine how New Era of Networks Formatter accesses fields in the input message to generate fields in the output message. You can select output field access modes and associated input field names to tell New Era of Networks Formatter how to map fields from the input messages to fields in the output message.

When format maps are used, the access mode controls can be moved from the output format definition into the mapping layer.

Defining Access Modes

To use access modes effectively, you must understand the concepts of parent format, child format, nesting levels, sibling fields, and controlling field.

A repeating flat format Y contained within a compound format X is called a child format of X. The parent format is X. Fields in any nonrepeating component of the parent format are parent fields of the fields in the child format. The fields in the child format are child fields. A parent is also referred to as an ancestor.

A repeating child format and the fields within it have a nesting level one greater than the nesting level of the parent format, and any nonrepeating components of the parent format. The nesting level of the root format is 1.

If field A is contained within the same repeating component at the same nesting level as field B, they are sibling fields.

The following example illustrates these access mode concepts.

```
(
      (F0)
      (F1 F2)
      {
            (F3 F4 F5)
      }
)
```

F0, F1, and F2 are the parent fields of child fields F3, F4, and F5.

F0, F1, and F2 are at nesting level 1. F3, F4, and F5 are at nesting level 2.

F0, F1, and F2 are siblings, even though F1 and F2 are in a different flat format than F0. F3, F4, and F5 are siblings.

The controlling field is a field in a repeating format component that controls the number of repetitions of that component. Each time a repeating format is output, the next instance of the controlling field is output. The repetitions end with the last child of the controlling field instance of the parent format. If the parent format has no controlling field, the repetitions terminate with the last field instance from the input message. A repeating component should contain only one controlling field. Alternative components each contain a controlling field because alternatives are independent of each other.

When there are repeating segments, if the controlling field is missing, the repetitions stop.

Types of Access Modes

The following access modes are available in New Era of Networks Formatter:

Access Mode Types

Access Mode	Description
Not Applicable	Do not access a field in the input message. You must supply a default value for this mode.

Access Mode	Description
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. In a repeating format, this access mode behaves the same as 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. This mode is used when explicit control is required because the relationship of input data is not well defined.
Access using Relative Index	The first field in a repeating component 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." Because this mode is essentially making a guess, this access mode should not be used often; use a more explicit access mode instead. This mode is only included for backward compatibility.
Access nth Instance of Field	Access the nth instance of a field in the input message.You can specify which instance of a field will fill a specified value. The instance can be specified in the subscript field below the access mode field. The subscript is zero based, meaning that if you actually want the value of the third instance, you would type 2 as the subscript value (0, 1, 2). (When n=0, then get the first instance.)
Controlling Field	Marks this field as 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 stop with the last field instance from the input message. This field should be mapped to an input field that was mandatory on input. Do not use a default value or an infinite loop is created. There is only one per flat format.

Access Mode	Description
Access current instance	Access the same field instance as on 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 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. This access mode is similar to normal access in that it accesses the first instance where there is no controlling field or accesses the instance in the same repeating component as the current controlling field.

Using Access Modes

Guidelines for Using Access Modes in Simple Cases

The following guidelines refer to all non-repeating formats and most repeating formats.

In simple formats, output field access falls into three categories:

1. Data is generated within New Era of Networks Formatter instead of being mapped from the input message. For example, literals, that use an access mode of Not Applicable and should be mapped to the input field name NONE.
2. Field is non-repeating and maps directly from the input. Use the access mode Normal Access.
3. Field is repeating.

Because non-repeating formats only have fields from categories 1 and 2, every field would have an access mode of either Normal Access or Not Applicable. Whether the format is mandatory or optional is irrelevant.

For category 3, the access modes needed would depend on the nesting of the repeating formats and their relation to the nesting of the input format.

The only access modes needed for simple cases, for example, using the same nesting structure as the input, are Controlling Field and Access Sibling. Each level of nesting requires a controlling field that is mandatory in the input. all other fields, except fields that belong to category 1, have the access mode of Access Sibling.

Access Mode Examples

The examples in this section illustrate various combinations of access mode concepts. Each example shows the input format definition and the output format definition, followed by the input message and the resulting output message.

The following conventions are used in the format definitions. An indent indicates an additional level of a specific format definition.

- Fn: Represents a field with the name "Fn", such as F1 or F2.
- ( ): Items enclosed in parentheses represent field definitions within the same flat format. For example, (F1 F2) indicates a flat format with two fields, F1 and F2.
- { }: Items enclosed in braces indicate a repeating component. Items enclosed in braces are indented.
- / and \: Items enclosed in forward and backward slashes indicate an alternative format. In alternative formats, only one alternative applies for each single or repeating component.
- [ ]: Items enclosed in brackets indicate an optional component.

The following conventions are used in the input and output messages:

- ( ) Items enclosed in parentheses are field instances in the same repeating component. Nested sets of parentheses indicate repeating components within repeating components.
- Fn-K: Represents an instance of the field "Fn", such as F1-1.

Formatting Nonrepeating Messages into Nonrepeating Messages

When you format an input message with no repeating components into an output message with no repeating components, the input and output messages have the same basic structure. Normal Access is used because it accesses the first instance of the field. In this example, you can also use Access Current Instance or Access Nth Instance, where N=0, but this introduces unwarranted complexity. There is no Controlling Field because there are no repeating components.

Input Format Definition

```
(F0 F1 F2 F3)
```

Output Format Definition

```
(F0 - Normal Access
 F1 - Normal Access
 F2 - Normal Access
 F3 - Normal Access)
```

Input Message

```
(F0-1 F1-1 F2-1 F3-1)
```

Output Message

```
(F0-1 F1-1 F2-1 F3-1)
```

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.

Input Format Definition

The input format describes three levels of nesting in the input message.

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

Output Format Definition

You can use Controlling Field and Access Sibling Instance access modes to build the output format definition.

```
{
    (F0 - Controlling Field)
    {
        (F1 - Controlling Field)
        {
             (F2 - Controlling Field
              F3 - Access Sibling Instance
              F4 - Access Sibling Instance)
        }
    }
}
```

Input Message

```
(
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1
        )
        (
            F2-2 F3-2 F4-2
        )
```

```
            (
            F1-2
            (
                F2-3 F3-3 F4-3
            )
        )
    )
    (
        F0-2
        (
            F1-3
            (
                F2-4 F3-4 F4-4
            )
            (
                F2-5 F3-5 F4-5
            )
            (
            F1-4
            (
                F2-6 F3-6 F4-6
            )
        )
    )
```

Output Message

Each time field F0 is accessed, the next instance of F0 is output because F0 is a controlling field. F0 has no parent controlling field, so the iterations terminate with the last instance of F0 in the input message.

F1 is a controlling field at its nesting level, so each time F1 is accessed the next instance of F1 is output. The parent controlling field of F1 is F0. There are multiple iterating sequences of F1. A specific iterating sequence of F1 terminates with the last child F1 of the current instance of F0.

F2 is a controlling field at its nesting level, so each time F2 is accessed the next instance of F2 is output. The parent controlling field of F2 is F1. There are multiple iterating sequences of F2. A specific iterating sequence of F2 terminates with the last child F2 of the current instance of F1.

F3 and F4 are siblings of F2. When a new iteration of (F2 F3 F4) is output, the next instance of F2 is output because F2 is the controlling field. The F3 and F4 instances in the same repeating group as the current instance of F2 is output.

```
(
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1
        )
        (
            F2-2 F3-2 F4-2
        )
        (
        F1-2
        (
            F2-3 F3-3 F4-3
        )
    )
)
(
    F0-2
    (
        F1-3
        (
            F2-4 F3-4 F4-4
        )
        (
            F2-5 F3-5 F4-5
        )
        (
        F1-4
        (
            F2-6 F3-6 F4-6
        )
    )
)
```

Outputting the Same Field Twice in a Repeating Component

To format an input message with repeating components into an output message where a particular field is output more than once in each repetition, you can use combinations of Controlling Field, Access Sibling Instance, and Access Current Instance.

Input Format Definition

The input format describes three levels of nesting in an input message.

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

Output Format Definition

F2 is the repeating field.

```
{
    (F0 - Controlling Field)
    {
        (F1 - Controlling Field)
        {
            (F2 - Controlling Field
             F3 - Access Sibling Instance
             F4 - Access Sibling Instance
             F2 - Access Current Instance)
             }
        }
    }
}
```

Input Message

```
(
```

```
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1
        )
        (
            F2-2 F3-2 F4-2
        )
        (
        F1-2
        (
            F2-3 F3-3 F4-3
        )
    )
)
(
    F0-2
    (
        F1-3
        (
            F2-4 F3-4 F4-4
        )
        (
            F2-5 F3-5 F4-5
        )
        (
        F1-4
        (
            F2-6 F3-6 F4-6
        )
    )
)
```

Output Message

F1, F2, and F3 increment as described in the preceding example. *Formatting Nested Messages into Nested Messages with a Similar Structure* on page 500F3 and F4 are siblings of F2, and the instances that belong with the current F2 are output with F2. The second F2 in the definition has the access mode Access Current Instance. The same instance of F2 is accessed as was last accessed.

```
(
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1 F2-1
        )
        (
            F2-2 F3-2 F4-2 F2-2
        )
        (
        F1-2
        (
            F2-3 F3-3 F4-3 F2-3
        )
    )
)
(
    F0-2
    (
        F1-3
        (
            F2-4 F3-4 F4-4 F2-4
        )
        (
            F2-5 F3-5 F4-5 F2-5
        )
        (
        F1-4
        (
            F2-6 F3-6 F4-6 F2-6
        )
    )
)
```

Formatting a Nested Format into a Nested Format with a Missing Level

To format an input message with nested repeating components into an output message missing one of the intermediate nesting levels in the input

format, you can use combinations of the access modes Controlling Field and Access Sibling Instance.

Input Format Definition

The input format describes three levels of nesting in an input message.

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

Output Format Definition

To eliminate the nesting level containing field F1, use Controlling Field and Access Sibling Instance access modes.

```
{
    (F0 - Controlling Field)
    {
        (F2 - Controlling Field
         F3 - Access Sibling Instance
         F4 - Access Sibling Instance)
    }
}
```

Input Message

```
(
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1
        )
        (
            F2-2 F3-2 F4-2
        )
```

```
            (
            F1-2
            (
                F2-3 F3-3 F4-3
            )
        )
    )
    (
        F0-2
        (
            F1-3
            (
                F2-4 F3-4 F4-4
            )
            (
                F2-5 F3-5 F4-5
            )
            (
            F1-4
            (
                F2-6 F3-6 F4-6
            )
        )
    )
```

Output Message

In preceding examples, the parent of the controlling field F2 was the controlling field F1 at the next lower level of nesting. In this example, F1 is not included in the output format definition. The parent controlling field is F0. F0 is actually the grandparent of F2. The iterations of F2 in each repeating sequence of F2s ends with the last grandchild of F0.

```
(
    F0-1
    (
        F2-1 F3-1 F4-1
    )
    (
        F2-2 F3-2 F4-2
    )
    (
```

```
        F2-3 F3-3 F4-3
    )
)
(
    F0-2
    (
        F2-4 F3-4 F4-4
    )
    (
        F2-5 F3-5 F4-5
    )
    (
        F2-6 F3-6 F4-6
    )
)
```

Formatting a Nested Format into a Flattened Format

To flatten a nested input message so the parent fields are output with each child field instance, use combinations of Controlling Field, Access Parent Instance, and Access Sibling Instance.

Input Format Definition

The input format describes three levels of nesting in an input message.

```
{
    (F0)
    {
        (F1)
        {
            (F2 F3 F4)
        }
    }
}
```

Output Format Definition

To eliminate the nesting level containing field F1, use Controlling Field and Access Sibling Instance.

```
{
    (F0 - Access Parent Instance
     F1 - Access Parent Instance
     F2 - Controlling Field
     F3 - Access Sibling Instance
     F4 - Access Sibling Instance)
}
```

Input Message

```
(
    F0-1
    (
        F1-1
        (
            F2-1 F3-1 F4-1
        )
        (
            F2-2 F3-2 F4-2
        )
        (
        F1-2
        (
            F2-3 F3-3 F4-3
        )
    )
)
(
    F0-2
    (
        F1-3
        (
            F2-4 F3-4 F4-4
        )
        (
            F2-5 F3-5 F4-5
        )
        (
```

```
        F1-4
        (
            F2-6 F3-6 F4-6
        )
    )
)
```

Output Message

The controlling field is F2. It has no parent controlling field, so the iterations terminate with the last instance of F2. F0 is the grandparent, and F1 is the parent of F2 in the input format definition. Selecting the access mode Access Parent Instance for F0 and F1 tells New Era of Networks Formatter to access the immediate ancestor instance (parent, grandparent, and so on) of the current controlling field instance from the input message. F3 and F4 are siblings. The instances of F3 and F4 from the same repeating component as the current instance of F2 are output with F2.

```
(
    F0-1 F1-1 F2-1 F3-1 F4-1
)
    F0-1 F1-1 F2-2 F3-2 F4-2
)
    F0-1 F1-2 F2-2 F3-3 F4-3
)
    F0-2 F1-3 F2-4 F3-4 F4-4
)
    F0-2 F1-4 F2-5 F3-5 F4-5
)
    F0-2 F1-4 F2-6 F3-6 F4-6
)
```

Formatting an Alternative with Overlapping Field Names

To handle alternative output formats, use combinations of Controlling Field and Access Sibling Instance.

Input Format Definition

```
{
     (F0)
     {
          (
               /
                    (F1 F2)
                    (
                         /
                              (F3)
                         \
                         /
                              (F4 F5)
                         \
                    )
                    {
                         (F6)
                    }
               \
               /
                    (F1)
                    (
                       /
                              (F8)
                       \
                       /
                              (F9)
                      \
                    )
                    {
                         (F6)
                    }
               \
          )
     }
}
```

Output Format Definition

The following output format definition has two alternatives at nesting level 2: the alternative that contains F1, F2, F3, F4, F5, and F6; and the alternative that

contains F1, F8, F9, and F6. Each of those alternatives has three components: a single flat format, an alternative format, and a repeating flat format.

F1 is output for each repetition of the first alternative, so F1 is the controlling field. Either F3 or (F4 F5) appears with F1. F3, F4, and F5 are at the same nesting level as F1. To output the instances of F3, F4, and F5 that go with the current instance of F1, use Access Sibling Instance as the access mode for these fields.

The repeating component F6 is at the next lower nesting level from the nesting level of F1. Because there is only one field, F6 has its own Controlling Field access mode. Iterations of F6 terminate with the last child of the current parent controlling field instance F1.

The second alternative at nesting level 2 contains F1, F8, F9, and F6. F1 in the second alternative is also a controlling field. Alternatives are independent of each other, so each alternative can have its own controlling fields.

F8 and F9 are siblings of the F1 instances that go with F8 and F9, not the F1 instances that go with F3, F4, and F5. The instances of F6 that are children of the second alternative F1 instances should be output with this parent controlling instance.

```
{
    (F0)
    {
        (
            /
                (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)
                    }
                \
            )
        }
}
```

Input Message (Example 1)

The following sample message matches the input format definition.

```
(
    F0-1
    (
        F1-1 F8-1
        (
             F6-1
        )
    )
    (
        F1-2 F9-1
        (
             F6-2
        )
        (
             F6-3
        )
    )
    (
        F1-3 F2-1 F4-1 F5-1
        (
             F6-4
```

```
          )
          (
               F6-5
          )
     )
)
(
     F0-2
     (
          F1-4 F2-2 F3-1
          (
               F6-6
          )
          (
               F6-7
          )
     )
     (
          F1-5 F8-2
          (
               F6-8
          )
     )
     (
          F1-6 F2-3 F4-2 F5-2
          (
               F6-9
          )
          (
               F6-10
          )
     )
)

I
```

Input Message (Example 2)

```
Field Name       F0    F1    F2    F3    F4    F5    F6    F1
F8    F9    F6
Instance of       1
1     1               1
Each field in
2             1       2
The message
3
                                    3      1               1
1      4

5
                         2      4      2       1
6

7
                          5
2                 8
                                    6      3               2
2       9

10
```

Output Message

```
(
    F0-1
    (
        F1-1 F8-1
        (
             F6-1
        )
    )
    (
        F1-2 F9-1
        (
             F6-2
        )
        (
             F6-3
        )
```

```
        )
        (
            F1-3 F2-1 F4-1 F5-1
            (
                 F6-4
            )
            (
                 F6-5
            )
        )
    )
    (
        F0-2
        (
            F1-4 F2-2 F3-1
            (
                 F6-6
            )
            (
                 F6-7
            )
        )
        (
            F1-5 F8-2
            (
                 F6-8
            )
        )
        (
            F1-6 F2-3 F4-2 F5-2
            (
                 F6-9
            )
            (
                 F6-10
            )
        )
    )
```

Formatting a Floating Alternative with Overlapping Field Names

The following example illustrates an input format definition for a format with floating alternative components. The alternative components are considered floating, because for each repetition of the alternative, there is no field that is always output with the alternative.

Input Format Definition

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

Output Format Definition

There is no anchoring field for each alternative, so a controlling field must be designated for each alternative.

```
{
    /
        (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)
    \
}
```

Input Message

```
(
    F1-1 F2-1 F3-1
)
(
    F1-2 F2-2
)
(
    F2-3 F3-2
)
(
    F1-3 F2-4
)
(
    F1-4 F2-5 F3-3
)
(
    F2-6 F3-4
)
(
    F1-5 F2-7
)
```

Output Message

```
(
    F1-1 F2-1 F3-1
)
(
    F1-2 F2-2
)
(
    F2-3 F3-2
)
(
    F1-3 F2-4
)
(
    F1-4 F2-5 F3-3
)
(
    F2-6 F3-4
```

```
)
(
    F1-5 F2-7
)
```

Formatting an Output Message with Optional Repeating Components

The following example illustrates an output format definition that contains an optional component.

Input Format Definition

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

Output Format Definition

F3 appears in each repetition of the format, so you can use it as the controlling field. You can also designate F4 or F5 as the controlling field. F1 and F2 are optional. If F1 and F2 do appear, the instances of F1 and F2 that belong with the current instance of F3 are output

```
{
    [ (F1 - Access Sibling Instance
       F2 - Access Sibling Instance) ]
      (F3 - Controlling Field
       F4 - Access Sibling Instance
       F5 - Access Sibling Instance)
}
```

Input Message

```
(
    (
        F1-1 F2-1
    )
    (
        F3-1 F4-1 F5-1
```

```
    )
(
)
    (
        F3-2 F4-2 F5-2
    )
)
(
    (
        F1-3 F2-3
    )
    (
        F3-3 F4-3 F5-3
    )
)
```

Output Message

The output message has the same structure and contents.

```
(
    (
        F1-1 F2-1
    )
    (
        F3-1 F4-1 F5-1
    )
(
)
    (
        F3-2 F4-2 F5-2
    )
)
(
    (
        F1-3 F2-3
    )
    (
        F3-3 F4-3 F5-3
    )
)
```

Formatting an Output Message with Boolean Fields in Repeating Components

Boolean controls are used to evaluate a field value; however, the field itself is not output in the message. The output control types Field= and Field Exists are Boolean controls.

Input Format Definition

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

Output Format Definition

In the following output format definition, F1 has a Boolean control of Field= "X". F1 is designated Access Sibling Instance because the evaluation is performed on the instance of F1 in that repetition. The sibling of F1 is itself. Access Current Instance has the same result. Do not use Controlling Field for this instance of F1 because F1 would increment twice for the same repetition.

```
{
    (F1 (Boolean) - Access Sibling Instance
     F1 - Controlling Field
     F2 - Access sibling instance
     F3 - Access sibling instance)
}
```

Input Message

```
(
    F1-1 ("X") F2-1 F3-1
)
(
    F1-2 ("Y") F2-2 F3-2
)
(
    F1-3 ("X") F2-3 F3-3
)
(
    F1-4 ("X") F2-4 F3-5
```

```
)
(
    F1-5 ("Z") F2-5 F3-5
)
```

Output Message

```
(
    F1-1 ("X") F2-1 F3-1
)
(
    F1-3 ("X") F2-3 F3-3
)
(
    F1-4 ("X") F2-4 F3-4
)
```

Using Access nth Instance of Field

The following example illustrates an output format definition that can select a specific instance of a repetition for use in the output. The output definition will produce a repeating sequence of data for some fields and a fixed instance of another field.

Input Format Definition

```
{
      (F0 F1)
}
```

Output Format Definition

```
{
      (
      F0 - Access nth Instance, Subscript = 1
      F1 - Controlling Field
      )
}
```

Input Message

```
(F0-1 F1-1)
(F0-2 F1-2)
```

```
(F0-3 F1-3)
```

Output Message

```
(F0-2 F1-1)
(F0-2 F1-2)
(F0-2 F1-3)
```

Appendix D

Hierarchy IDs

ID	Name
1	Node
2	Application Group
3	Message Type
4	Rule
5	Subscription
6	Filler 6
7	Filler 7
8	Filler 8
9	Filler 9
10	Filler 10
11	Filler 11
12	Filler 12
13	Filler13
14	Filler 14
15	Filler 15
16	Filler 16
17	Filler 17
18	Filler 18

ID	Name
19	Plug-in
20	Field
21	Literal
22	User-Defined Type
23	Output Operation
24	Input Control
25	Output Control
26	Output Operation Collection
27	Format
28	Format Collection
29	Format Map
30	Maplink
31	Substitute
32	PrefixorPostfix
33	Default
34	Length
35	Substring
36	UserExit
37	MathExpression
38	Trim
39	Filler 39
40	Filler 40
41	Filler 41

ID	Name
42	Filler 42
43	Filler 43

Appendix E
Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

may vary significantly. Some measurements may have been made on development-level systems and there is not guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information (http://www.ibm.com/legal/copytrade.shtml).

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

Java is a registered trademark of Oracle and/or its affiliates.

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

Y

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

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book only and the way in which the information is presented.

To request additional publications or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail:

  IBM United Kingdom Laboratories
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN

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

Electronically, use the appropriate network ID:

- IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
- IBMLink: HURSLEY(IDRCF)

- Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic number to which your comment applies
- Your name / address / telephone number / fax number / network ID